The Why3 platform

Version 0.88.2, December 2017

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

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

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

Foreword

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

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

Availability

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

Why3 is 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 5 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, Stefan Berghofer, Sylvie Boldo, Martin Clochard, Simon Cruanes, Léon Gondelman, Johannes Kanig, Stéphane Lescuyer, David Mentré, Simão Melo de Sousa, Benjamin Monate, Thi-Minh-Tuyen Nguyen, Mário Pereira, Asma Tafat, Piotr Trojanek.

Contents

Part I

Tutorial

Chapter 1

Getting Started

1.1 Hello Proofs

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

```
theory HelloProof "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 HelloProof and labeled 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 6.3 for a more complete description.

The GUI is launched on the file above as follows.

```
why3 ide hello_proof.why
```

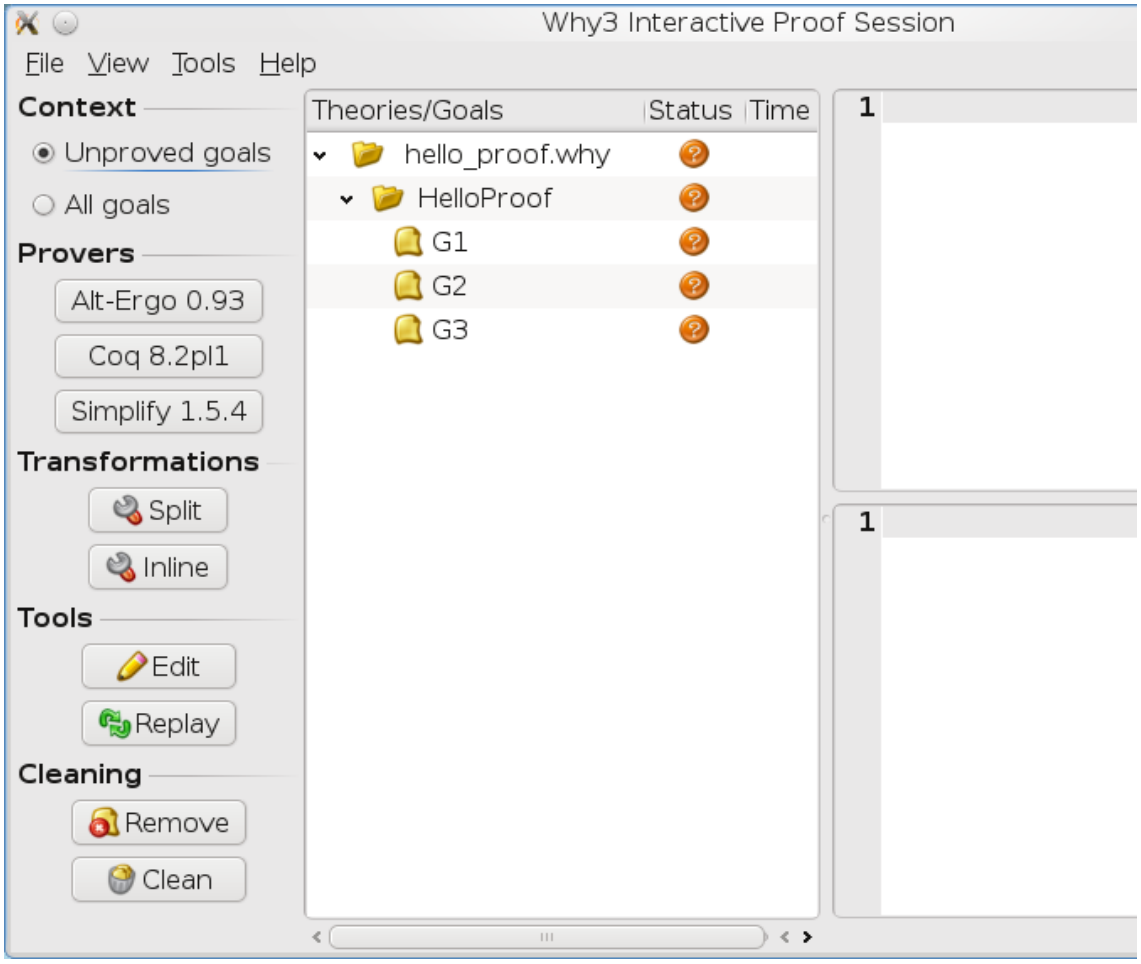

Figure 1.1: The GUI when started the very first time

When the GUI is started for the first time, you should get a window that 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 computer.[1] Three provers were detected, in this case, these are Alt-Ergo [4], Coq [2] and Simplify [5].

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.

1.2.1 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

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

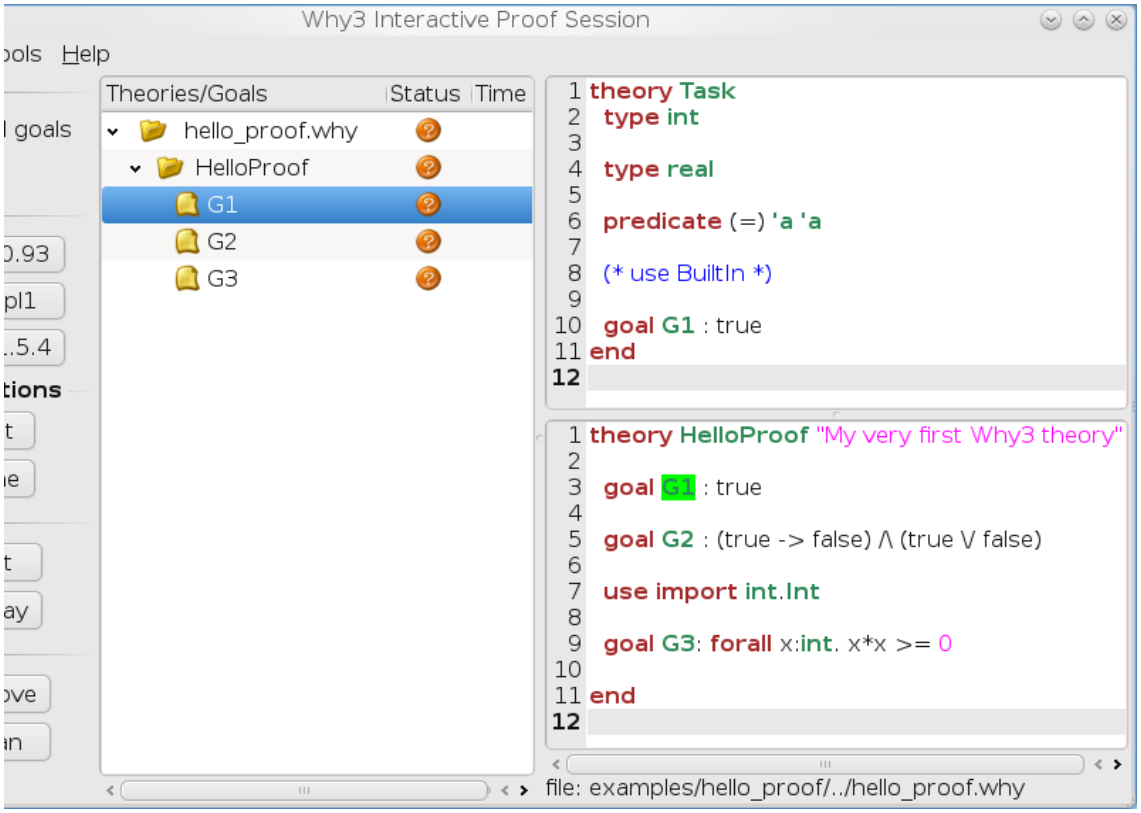

Figure 1.2: The GUI with goal G1 selected

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

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. Goal G_3 should be proved now, but not G_2.

1.2.2 Applying transformations

Instead of calling a prover on a goal, you can apply a transformation to it. Since G_2 is a conjunction, a possibility is to split it into subgoals. You can do that by 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 6.3 for details on how to configure this). You get now a regular Coq file to fill in,

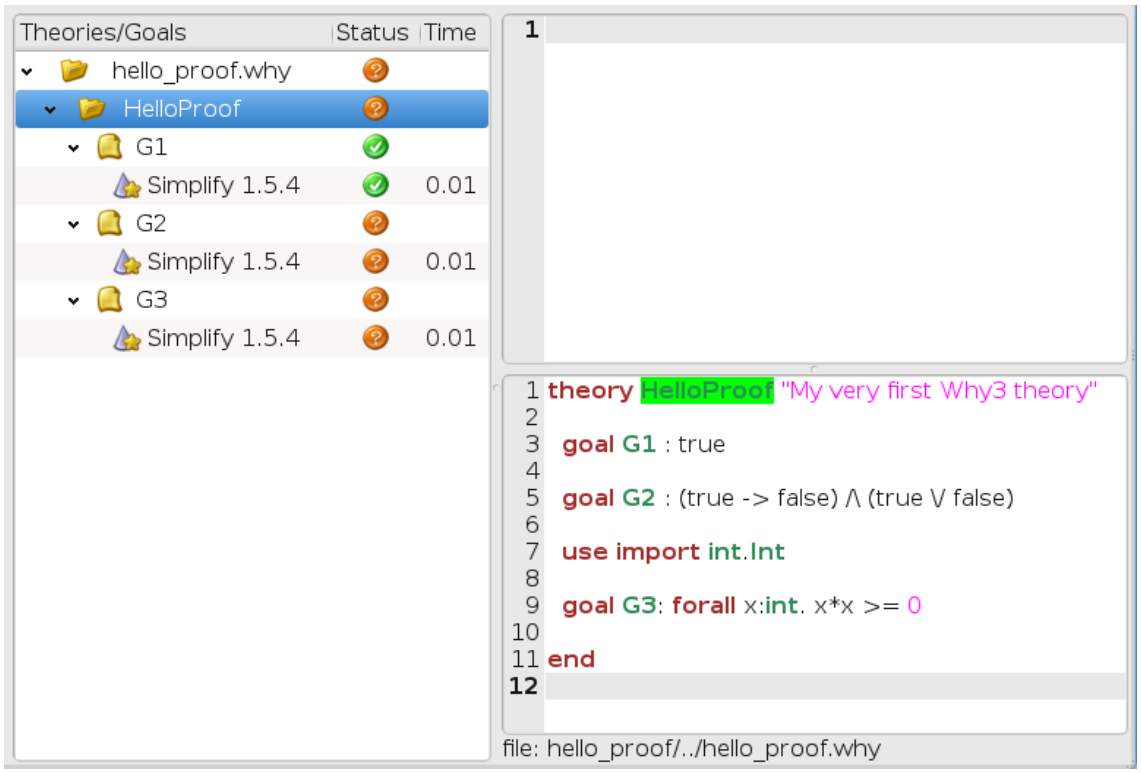

Figure 1.3: The GUI after Simplify prover is run on each goal

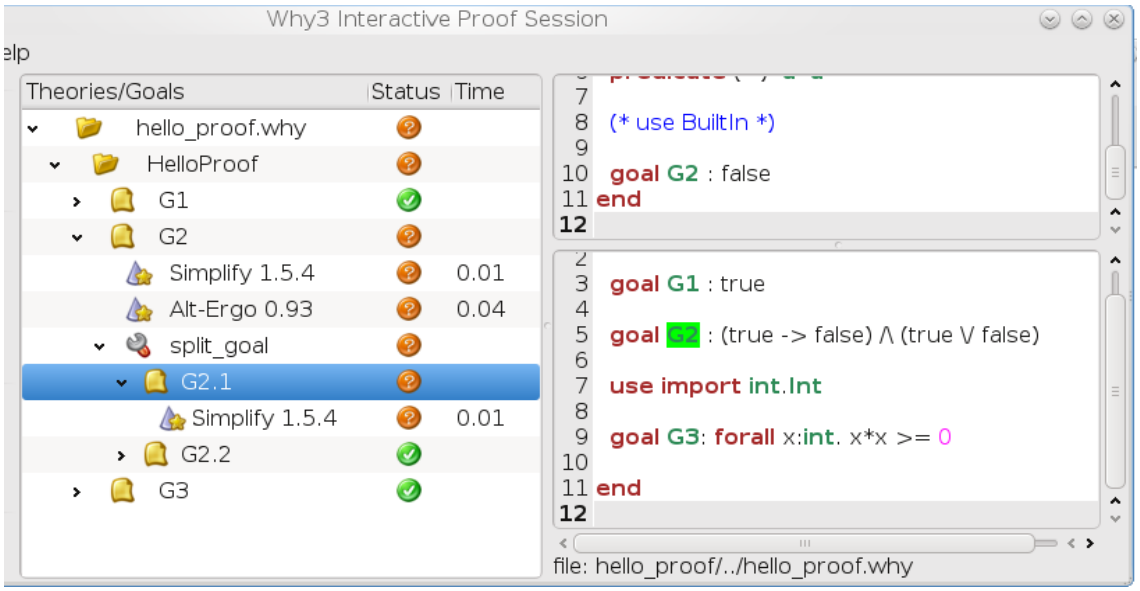

Figure 1.4: The GUI after splitting goal G_2 and collapsing proved goals

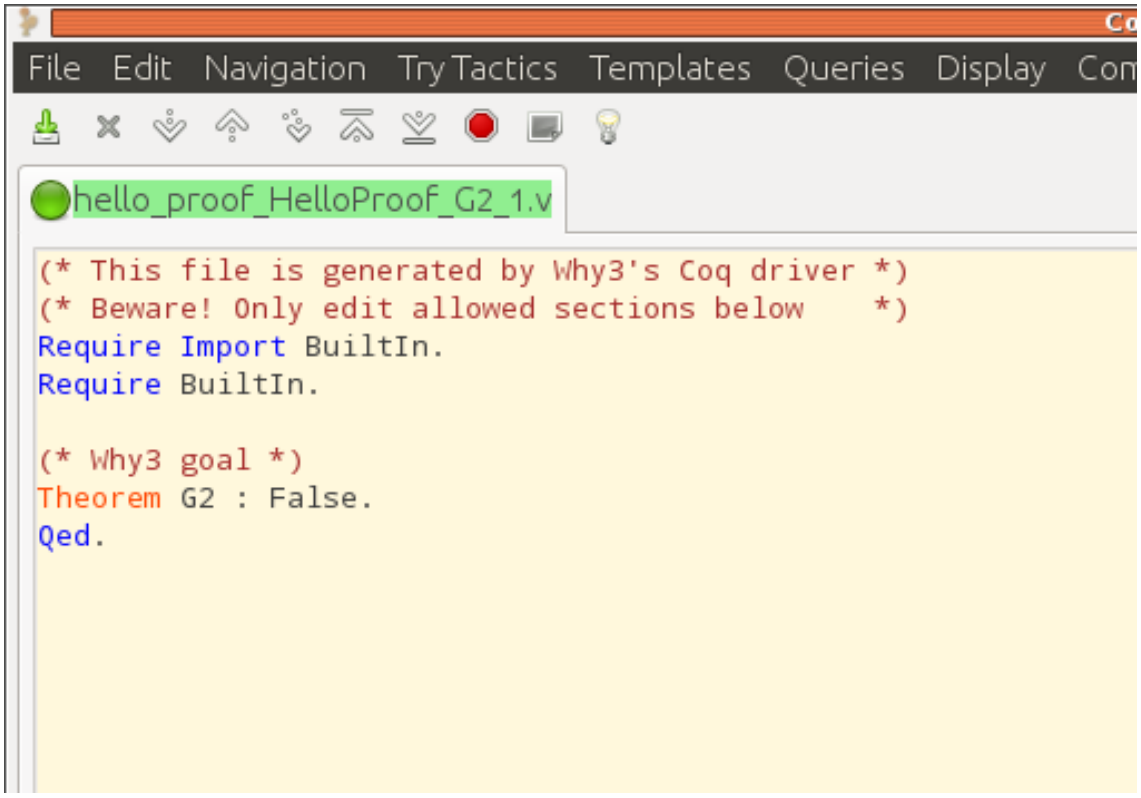

Figure 1.5: CoqIDE on subgoal 1 of G_2

as shown on Figure 1.5. Please be mindful of the comments of this file. They indicate where Why3 expects you to fill the blanks. Note that the comments themselves should not be removed, as they are needed to properly regenerate the file when the goal is changed. See Section 9.3 for more details.

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

1.2.3 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 us assume we change the goal G_2 by replacing the first occurrence of true by false, *e.g.*

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

We can 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.

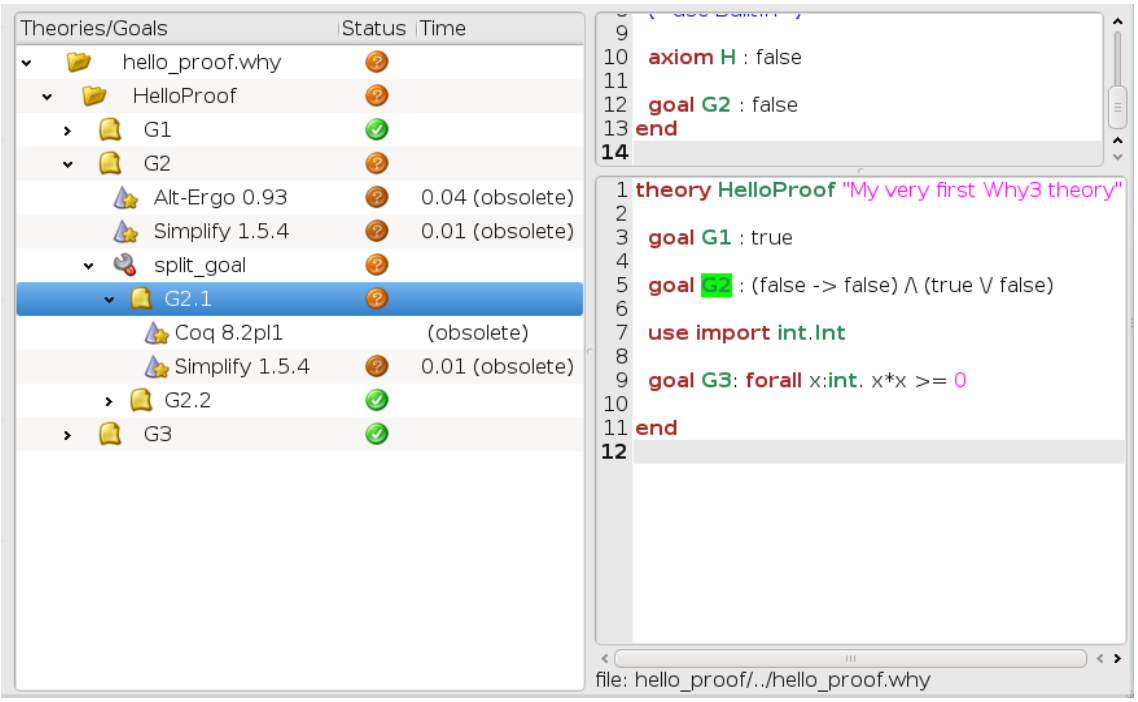

Figure 1.6: File reloaded after modifying goal G_2

1.2.4 Replaying obsolete proofs

Instead of pushing a prover's button to rerun its proofs, you can *replay* the existing but obsolete proof attempts, 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 `replay` command (see Section 6.5) For example, running the replayer on the `hello_proof` example is as follows (assuming G_2 still is `(true -> false) /\ (true \/ false)`).

```
$ why3 replay hello_proof
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 differences were detected between the current run and the run stored in the XML file. The tree above reminds us that G_2 is not proved.

1.2.5 Cleaning

You may want to clean some 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 `prove` command makes it possible to check the validity of goals with external provers, in batch mode. This section presents the basic use of this tool. Refer to Section 6.2 for a more complete description of this tool and all its command-line options.

The very first time you want to use Why3, 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):

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

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

```
> why3 --list-provers
Known provers:
  alt-ergo (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 [4], Coq [2] and Simplify [5].

Let us assume that 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 prove -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 the Why3 GUI, the command-line tool does not save the proof attempts or applied transformations in a database.

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

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

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

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

Known splitting transformations:
  [...]
  split_goal
  split_intro
```

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

```
> why3 prove -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 10.5 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.

2.1 Example 1: Lists

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` $\times$ `list 'a` $\rightarrow$ `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

```
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

an instance of the defined predicate. Moreover, the defined predicate may only occur in positive positions in the antecedent. For example, a clause:

```
  | 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.

2.2 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

```
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)

cloned theory, possibly instantiating some of its abstract (*i.e.* declared but not defined) symbols.

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 10.4).

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 `O.t` of `SortedGen` and the default order on integers for predicate `O.(<=)`. 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 O` 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 `O.` 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 O`" can be equivalently written as:

```
namespace import O
  clone export Order
end
```

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

2.3 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 code given below is available in the source distribution in directory `examples/logic/einstein.why`.

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;

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

- The yellow house's owner smokes Dunhill;
- In the house in the center lives someone who drinks milk;
- The Norwegian lives in the first house;
- The man who smokes Blends lives next to the one who has cats;
- The man who owns a horse lives next to the one who smokes Dunhills;
- The man who smokes Blue Masters drinks beer;
- The German smokes Prince;
- The Norwegian lives next to the blue house;
- The man who smokes Blends has a neighbour who drinks water.

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

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

```
theory Bijection
  type t
  type u

  function of t : u
  function to_ u : t

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

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

```
theory Einstein "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 WhyML Programming Language

This chapter describes the WhyML programming language. A WhyML 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` and/or `ghost`.
- In an algebraic type declaration (this includes record types), an invariant can be specified.
- There are programming constructs with no counterpart in the logic:
  - mutable field assignment;
  - sequence;
  - loops;
  - exceptions;
  - local and anonymous functions;
  - ghost parameters and ghost code;
  - annotations: pre- and postconditions, assertions, loop invariants.
- A program 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 `why3`. For instance

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

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

```
> why3 prove -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 6 for more details regarding command lines.

As an introduction to WhyML, we use the five problems from the VSTTE 2010 verification competition [10]. The source code for all these examples is contained in Why3's distribution, in sub-directory `examples/`.

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 WhyML's standard library, so we import the corresponding modules, with a similar declaration:

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

Modules `Ref` and `Array` respectively provide a type `ref 'a` for references and a type `array 'a` for arrays, together with useful operations and traditional syntax. They are loaded from the WhyML files `ref.mlw` and `array.mlw` in the standard library. Why3 reports an error when it finds a theory and a module with the same name in the standard library, or when it finds a theory declared in a `.mlw` file and in a `.why` file with the same name.

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

The first 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). The second precondition expresses 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. The same postcondition can be written in another form, doing the pattern matching immediately:

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

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

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

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

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

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

```
  (!sum, !max)
```

This completes the code for function `max_sum`. As such, it cannot be proved correct, since the loop is still lacking a loop invariant. In this case, the loop invariant is as simple as `!sum <= i * !max`, since the postcondition only requires 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 `why3`, 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:

```
module MaxAndSum

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

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

end
```

Figure 3.1: Solution for VSTTE'10 competition problem 1

> 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 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:

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

Here we chose to have array `b` as argument; returning a freshly allocated array would be equally simple. The whole module is given in Figure 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 import option.Option
  use export list.List
  use export list.Length
  use export list.Nth
```

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

```
module InvertingAnInjection

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

end
```

Figure 3.2: Solution for VSTTE'10 competition problem 2

```
  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`.

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

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

end
```

Figure 3.3: Solution for VSTTE'10 competition problem 3

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

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

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

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

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

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

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

Solving the problem is simply a matter of calling `search` with 0 as first argument. The code is given Figure 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 functions `hd` and `tl` over lists.

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

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

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

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

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

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

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

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

3.4 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 array.Array
```

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

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

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

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

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

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

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

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

  exception Inconsistent int

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

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

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 /\
    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`:

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

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

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

We place a code mark `'Init` immediately at the beginning of the program body 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 }
    invariant { forall b:array int.  length b = n -> is_board b n ->
      eq_board board b pos -> 0 <= b[pos] < i -> not (solution b n) }
```

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

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

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

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

This second part of the solution is given Figure 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.

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* [8] for more details.)

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

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

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

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

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; }
  invariant {
    length self.front = self.lenf >= length self.rear = self.lenr }
```

The type definition is accompanied with an invariant — a logical property imposed on any value of the type. Why3 assumes that any `queue` passed as an argument to a program function satisfies the invariant and it produces a proof obligation every time a `queue` is created or modified in a program.

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

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

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

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

end
```

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

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

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

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

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

That the argument `q` satisfies the type invariant is implicitly assumed. The type invariant 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 implementation 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)
  requires { lf = length f /\ lr = length r }
  ensures  { sequence result = f ++ reverse r }
= if lf >= lr then
    { front = f; lenf = lf; rear = r; lenr = lr }
  else
    let f = f ++ reverse r in
    { front = f; lenf = lf + lr; rear = Nil; lenr = 0 }
```

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

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

```
  end
```

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)
  ensures { sequence result = sequence q ++ Cons x Nil }
= create q.front q.lenf (Cons x q.rear) (q.lenr + 1)
```

The code is given Figure 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; }
  invariant {
    length self.front = self.lenf >= length self.rear = self.lenr }

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

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

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

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

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

  let enqueue (x: 'a) (q: queue 'a)
    ensures { sequence result = sequence q ++ Cons x Nil }
  = create q.front q.lenf (Cons x q.rear) (q.lenr + 1)
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-0.88.2/.

We assume the reader has a fair knowledge of the OCaml language. Notice that the Why3 library must be installed, see Section 5.3. The OCaml code given below is available in the source distribution in directory `examples/use_api/` together with a few other examples.

4.1 Building Propositional Formulas

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

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

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

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

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

As expected, the output is as follows.

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

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

4.2 Building Tasks

Let us see how we can call a prover to prove a formula. As said in previous chapters, a prover must be given a task, so we need to build tasks from our formulas. Task can be build incrementally from an empty task by adding declaration to it, using the functions `add_*_decl` of module `Task`. For the formula $true \vee false$ 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 that lemmas are not allowed in tasks and can only be used in theories.

Once a task is built, it can be printed.

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

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

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

Execution of our OCaml program now outputs:

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

task 2 is:
theory Task
  predicate A

  predicate B

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

4.3 Calling External Provers

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

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

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

```
type prover =
    { prover_name : string; (* "Alt-Ergo" *)
      prover_version : string; (* "2.95" *)
      prover_altern : string; (* "special" *)
    }
```

The map `provers` provides 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
    Whyconf.prover_by_id config "alt-ergo"
  with Whyconf.ProverNotFound _ ->
    eprintf "Prover alt-ergo not installed or not configured@.";
    exit 0
```

We could also get a specific version with :

```
let alt_ergo : Whyconf.config_prover =
  try
    let prover = {Whyconf.prover_name = "Alt-Ergo";
                  prover_version = "0.92.3";
                  prover_altern = ""} in
    Whyconf.Mprover.find prover 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 (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 "@[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 `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 7.4).

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

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

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

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

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

4.5 Building Quantified Formulas

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

```
let zero : Term.term =
  Term.t_const (Number.ConstInt (Number.int_const_dec "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

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

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

Creation of a theory named `My_theory` is done by

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

First let us add formula 1 above as a goal:

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

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

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

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

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

```
(* [use th1 th2] insert the equivalent of a "use import th2" in
  theory th1 under construction *)
let use th1 th2 =
  let name = th2.Theory.th_name in
  Theory.close_namespace
    (Theory.use_export
      (Theory.open_namespace th1 name.Ident.id_string) th2) true
```

Addition of formula 3 is then

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

Addition of goal 4 is nothing more complex:

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

Finally, we close our theory under construction as follows.

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

We can inspect what we did by printing that theory:

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

which outputs

```
theory is:
theory My_theory
  (* use BuiltIn *)

  goal goal1 : true \/ false

  predicate A

  predicate B

  goal goal2 : A /\ B -> A

  (* use int.Int *)

  goal goal3 : (2 + 2) = 4

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

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

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

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

We can check our generated tasks by printing them:

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

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

4.7 Applying Transformations

[TO BE COMPLETED]

4.8 Writing New Functions on Terms

[TO BE COMPLETED]

4.9 Proof Sessions

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

4.10 ML Programs

There are two ways for building WhyML programs from OCaml. The first is to build untyped syntax trees for such WhyML programs, and then call the Why3 typing procedure to build typed declarations. The second way is to directly build typed programs using smart constructors that check well-typedness at each step.

The first approach, building untyped trees and then typing them, is examplified in file `examples/use_api/mlw_tree.ml` of the distribution. The second approach is examplified in file `examples/use_api/mlw.ml`. The first approach is significantly simpler to do since the internal typing mechanism using regions remains implicit, whereas when one uses the second approach one should care about such typing. On the other hand, the second approach is more "efficient" in the sense that no intermediate form needs to be built in memory.

Part II

Reference Manual

Chapter 5

Compilation, Installation

In short, installation proceeds as follows.

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

5.1 Installation Instructions from Source Distribution

After unpacking the distribution, go to the newly created directory `why3-0.88.2`. Compilation must start with a configuration phase which is run as

`./configure`

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

- The Objective Caml compiler, version 3.11.2 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 of the Why3 tools, additional OCaml libraries are needed:

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

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

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

- For `why3 bench`, the OCaml bindings of the sqlite3 library are needed. 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

If you want to use the specific Coq features, *i.e.* the Coq tactic (Section 9.3.1) and Coq realizations (Section 9.2), then Coq has to be installed before Why3. Look at the summary printed at the end of the configuration script to check if Coq has been detected properly. Similarly, for using PVS (Section 9.5) or Isabelle (Section 9.4) to discharge proofs, PVS and Isabelle must be installed before Why3. You should check that those proof assistants are correctly detected by the configure script.

When configuration is finished, you can compile Why3.

```
make
```

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

```
make install
```

Installation can be tested as follows:

1. install some external provers (see Section 5.4 below)
2. run `why3 config --detect`
3. run some examples from the distribution, *e.g.* you should obtain the following:

```
$ cd examples
$ why3 replay logic/scottish-private-club
Opening session... done
Progress: 4/4
 1/1
Everything OK.
$ why3 replay programs/same_fringe
Opening session... done
Progress: 12/12
 3/3
Everything OK.
```

5.2 Local Use, Without Installation

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

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

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

5.3 Installation of the Why3 API

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

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

5.4 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 automatic provers, *e.g.* SMT solvers, before compiling and installing Why3.

For installation of external provers, please refer to the specific section about provers on the Web page http://why3.lri.fr/.

For configuring Why3 to use the provers, follow instructions given in Section 6.1.

5.5 Multiple Versions of the Same Prover

Why3 is able to use several versions of the same prover, *e.g.* it can use both CVC3 2.2 and CVC3 2.4.1 at the same time. The automatic detection of provers looks for typical names for their executable command, *e.g.* `cvc3` for CVC3. However, if you install several version of the same prover it is likely that you would use specialized executable names, such as `cvc3-2.2` or `cvc3-2.4.1`. To allow the Why3 detection process to recognize these, you can use the option `--add-prover` with the `config` command, *e.g.*

```
why3 config --detect --add-prover cvc3-2.4 /usr/local/bin/cvc3-2.4.1
```

the first argument (here `cvc3-2.4`) must be one of the class of provers known in the file `provers-detection-data.conf` typically located in `/usr/local/share/why3` after installation. See Appendix 10.2 for details.

5.6 Session Update after Prover Upgrade

If you happen to upgrade a prover, *e.g.* installing CVC3 2.4.1 in place of CVC3 2.2, then the proof sessions formerly recorded will still refer to the old version of the prover. If you open one such a session with the GUI, and replay the proofs, you will be asked to choose between 3 options:

- Keep the former proofs as they are. They will be marked as "archived".
- Upgrade the former proofs to an installed prover (typically a upgraded version). The corresponding proof attempts will become attached to this new prover, and marked as obsolete, to make their replay mandatory.
- Copy the former proofs to an installed prover. This is a combination of the actions above: each proof attempt is duplicated, one with the former prover marked as archived, and one for the new prover marked as obsolete.

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

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

Outside the GUI, the prover upgrades are handled as follows. The `replay` command will just ignore proof attempts marked as archived. Conversely, a non-archived proof attempt with a non-existent prover will be reported as a replay failure. The `session` command performs move or copy operations on proof attempts in a fine-grained way, using filters, as detailed in Section 6.6.

Chapter 6

Reference Manuals for the Why3 Tools

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

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

The following commands are available:

`bench` produces benchmarks.

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

`doc` produces HTML versions of Why3 source codes.

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

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

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

`prove` reads Why3 and WhyML input files and calls provers, on the command-line.

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

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

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

`wc` gives some token statistics about Why3 and WhyML source codes.

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

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

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

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

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

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

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

`--list-debug-flags` lists known debug flags.

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

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

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

6.1 The `config` Command

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

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

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

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

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

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

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

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

6.2 The `prove` Command

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

The `prove` command executes the following steps:

1. Parse the command line and report errors if needed.

2. Read the configuration file using the priority defined in Section 10.3.

3. Load the plugins mentioned in the configuration. It will not stop if some plugin fails to load.

4. Parse and typecheck the given files using the correct parser in order to obtain a set of Why3 theories for each file. It uses the filename extension or the `--format` option to choose among the available parsers. `why3 --list-formats` lists the registered parsers. WhyML modules are turned into theories containing verification conditions as goals.

5. Extract the selected goals inside each of the selected theories into tasks. The goals and theories are selected using options `-G/--goal` and `-T/--theory`. Option `-T/--theory` applies to the previous file appearing on the command line. Option `-G/--goal` applies to the previous theory appearing on the command line. If no theories are selected in a file, then every theory is considered as selected. If no goals are selected in a theory, then every goal is considered as selected.

6. Apply the transformations requested with `-a/--apply-transform` in their order of appearance on the command line. `why3 --list-transforms` lists the known transformations; plugins can add more of them.

7. Apply the driver selected with the `-D/--driver` option, or the driver of the prover selected with the `-P/--prover` option. `why3 --list-provers` lists the known provers, *i.e.* the ones that appear in the configuration file.

8. If option `-P/--prover` is given, call the selected prover on each generated task and print the results. If option `-D/--driver` is given, print each generated task using the format specified in the selected driver.

6.2.1 Prover Results

The provers can give the following output:

Valid The goal is proved in the given context.

Unknown The prover has stopped its search.

Timeout The prover has reached the time limit.

Failure An error has occurred.

Invalid The prover knows the goal cannot be proved.

6.2.2 Additional Options

`--get-ce` activates the generation of a potential counter-example when a proof does not succeed (experimental).

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

6.3 The `ide` Command

The basic usage of the GUI is described by the tutorial of Section 1.2. There are no specific command-line options, apart from the common options detailed in introduction to this chapter. However at least one anonymous argument must be specified on the command line. More precisely, the first anonymous argument must be the directory of the session. If the directory does not exist, it is created. The other arguments should be existing files that are going to be added to the session. For convenience, if there is only one anonymous argument, it can be an existing file and in this case the session directory is obtained by removing the extension from the file name.

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

6.3.1 Session

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

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

Invalid The task is invalid.

Timeout the prover exceeded the time limit.

OufOfMemory The prover exceeded the memory limit.

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

Failure The prover reported a failure.

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

Additionally, a proof attempt can have the following attributes:

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

archived The proof attempt is not useful anymore; it is kept for history; no Why3 tools will select it by default. Section 5.6 shows an example of this usage.

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

6.3.2 Left toolbar actions

Context presents 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.

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

Split splits the current goal into subgoals if it is a conjunction of two or more goals. It corresponds to the `split_goal_wp` transformation.

Inline inlines the definitions in the conclusion of the goal. It corresponds to the `introduce_premises` transformation follwoed by `inline_goal`.

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

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

A more detailed description of strategies is given in Section 10.6, as well as a description on how to design strategies of your own.

Provers provide a button for each detected prover. Clicking on such a button starts the corresponding prover on the selected goal(s).

Tools section provides a set of specific action that are typically performed on the selected goal(s). :

Edit starts an editor on the selected task.

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

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

Replay replays all the 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

Interrupt cancels all the proof attempts currently scheduled but not yet started.

6.3.3 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 saves 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 that 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 proof.

Non-splitting transformation applies one of the available transformations, as listed in Section 10.5.

Splitting transformation is the same as above, but for splitting transformations, *i.e.* those that can generate several sub-goals.

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

6.3.4 Preferences Dialog

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

General Settings tab allows one to set various general settings.

- the limits set on resource usages:
  - the time limit given to provers, in seconds
  - the memory given to provers, in megabytes
  - the maximal number of simultaneous provers allowed to run in parallel

  By default, modification of any of these settings has effect only for the current run of the GUI. A checkbox allows you to save these settings also for future sessions.
- a few display settings:
  - introduce premises: if selected, the goal of the task shown in top-right window is displayed after introduction of universally quantified variables and implications, *e.g.* a goal of the form $\forall x : t.P \rightarrow Q$ is displayed as

$$\frac{\begin{array}{l} x : t \\ H : P \end{array}}{Q}$$

  - show labels in formulas
  - show source locations in formulas
  - show time limit for each proof
- 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 saved automatically, you must use menu File/Save session
  - ask whether to save: on exit, a popup window asks whether you want to save or not.

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

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

Provers tab allows to select which of the installed provers one wants to see as buttons in the left toolbar.

Uninstalled Provers tab presents all the decision previously taken for missing provers, as described in Section 5.6. You can remove any recorded decision by clicking on it.

6.3.5 Displaying Counterexamples

When the selected prover finds (counterexample) model, it is possible to display parts of this model in the terms of the original Why3 input. Currently, this is supported for CVC4 prover version 1.5 and Z3.

To display the counterexample in Why3 IDE, the counterexample model generation must be enabled in File -/> Preferences -/> get counter-example. After running the prover and clicking on the prover result in, the counterexample can be displayed in the tab Counter-example. The counterexample is displayed with the original Why3 code in comments. Counterexample values for Why3 source code elements at given line are displayed in a comment at the line below. An alternative way how to display a counterexample is to use the option `--get-ce` of the `prove` command.

Why3 source code elements that should be a part of counterexample must be explicitly marked with `"model"` label. The following example shows a Why3 theory with some terms annotated with the `model` label and the generated counterexample in comments:

```
theory T

  use import int.Int

  goal g_no_lab_ex : forall x:int. x >= 42 -> x + 3 <= 50

  goal g_lab_ex : forall x "model":int. ("model" x >= 42) ->
    ("model" "model_trace:x+3<=50" x + 3 <= 50)

  goal computation_ex : forall x1 "model" x2 "model" x3 "model" .
  (* x1 = 1; x2 = 1; x3 = 1 *)
  ("model" "model_trace: x1 + 1 = 2" x1 + 1 = 2) ->
  (*  x1 + 1 = 2 = true *)
  ("model" x2 + 1 = 2) ->
  (* (= (+ x2 1) 2) = true *)
  ("model" x3 + 1 = 2) ->
  (* (= (+ x3 1) 2) = true *)
```

```
("model" x1 + x2 + x3  = 5)
(* (= (+ (+ x1 x2) x3) 5) = false *)
```

To display counterexample values in assertions the term being asserted must be labeled with the label `"model_vc"`. To display counterexample values in postconditions, the term in the postcondition must be labeled with the label `"model_vc_post"`. The following example shows a counterexample of a function with postcondition:

```
let incr_ex ( x "model" : ref int ) (y "model" : ref int): unit
(* x = -2; y = -2 *)
ensures { !x = old !x + 2 + !y }
=
y := !y + 1;
(* y = -1 *)
x := !x + 1;
(* x = -1 *)
x := !x + 1
(* x = 0 *)
```

It is also possible to rename counterexample elements using the lable `"model_trace:"`. The following example shows the use of renaming a counterexample element in order to print the counterexample in infix notation instead of default prefix notation:

```
goal g_lab_ex : forall x "model":int. ("model" x >= 42) ->
(* x = 48; (<= 42 x) = true *)
  ("model" "model_trace:x+3<=50" x + 3 <= 50)
  (* x+3<=50 = false *)
```

Renaming counterexample elements is in particular useful when Why3 is used as intermediate language and to show names of counterexample elements in the source language instead of showing names of Why3 elements. Note that if the counterexample element is of a record type, it is also possible to rename names of the record fields by putting the label `model_trace:` to definitions of record fields. For example:

```
type r = {f "model_trace:.F" :int; g "model_trace:G" :bool}
```

When a prover is queried for a counterexample value of a term of an abstract type=, an internal representation of the value is get. To get the concrete representation, the term must be marked with the label `"model_projected"` and a projection function from the abstract type to a concrete type must be defined, marked as a projection using the meta `"model_projection"`, and inlining of this function must be disabled using the meta `"inline : no"`. The following example shows a counterexample of an abstract value:

```
theory A

  use import int.Int

  type byte
  function to_rep byte : int
  predicate in_range (x : int) = -128 <= x <= 127
  axiom range_axiom : forall x:byte.
    in_range (to_rep x)
  meta "model_projection" function to_rep
```

```
meta "inline : no" function to_rep

goal abstr : forall x "model_projected" :byte. to_rep x >= 42 -> to_rep x
+ 3 <= 50
(* x = 48 *)
```

More examples of using counterexample feature of Why3 can be found in the file examples/tests/cvc4-models.mlw and examples/tests/cvc4-models.mlw. More information how counterexamples in Why3 works can be found in [7].

6.3.6 Additional Command-Line Options

The `ide` command also accepts the following options described for the command `prove` in Section 6.2.2.

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

6.4 The `bench` Command

The `bench` command adds a scheduler on top of the Why3 library. It is designed to compare various components of automatic proofs: automatic provers, transformations, definitions of a theory. For that purpose, it tries to prove predefined goals using each component to compare. Various formats can be used as outputs:

`csv` the simpler and more informative format; the results are represented in an array; the rows correspond to the compared components, the columns correspond to the result (Valid, Unknown, Timeout, Failure, Invalid) and the CPU time taken in seconds.

`average` it 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/misc/prgbench.rc` is an example of such a file. More generally, a bench configuration file looks like

```
[probs "myprobs"]
   file = "examples/mygoal.why" #relatives to the rc file
   file = "examples/myprogram.mlw"
   theory = "myprogram.T"
   goal = "mygoal.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`. A `tools` section defines a set of components to compare. A `probs` section defines a set of goals on which to compare some components. A `bench` section defines which components to compare using which goals. It refers by name to some sections `tools` and `probs` defined in the same file. The order of the definitions is irrelevant. Notice that one can use `loadpath` in a `tools` section to compare different axiomatizations.

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

```
why3 bench -B path_to_my_bench.rc
```

6.5 The `replay` Command

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

The tool is invoked in a terminal or a script using

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

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

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

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

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

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

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

Options are:

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

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

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

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

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

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

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

`none` Do not run the smoke detector.

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

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

becomes

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

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

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

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

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

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

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

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

6.6 The `session` Command

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

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

The available subcommands are as follows:

`info` prints informations and statistics about sessions.

`latex` outputs session contents in LaTeX format.

`html` outputs session contents in HTML format.

`mod` modifies some of the proofs, selected by a filter.

`copy` duplicates some of the proofs, selected by a filter.

`copy-archive` same as copy but also archives the original proofs.

`rm` removes some of the proofs, selected by a filter.

The first three commands do not modify the sessions, whereas the last four modify them. Only the proof attempts recorded are modified. No prover is called on the modified or created proof attempts, and consequently the proof status is always marked as obsolete.

6.6.1 Command `info`

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

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

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

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

`--tree` prints the structure of the session as a tree in ASCII, as detailed below.

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

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

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

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

Session Tree The hierarchical structure of the session is printed as a tree in ASCII. The files, theories, goals are marked with a question mark `?`, if they are not verified. A proof is usually said to be verified if the proof result is `valid` and the proof is not obsolete. However here specially we separate these two properties. On the one hand if the proof suffers from an internal failure we mark it with an exclamation mark `!`, otherwise if it is not valid we mark it with a question mark `?`, finally if it is valid we add nothing. On the other hand if the proof is obsolete we mark it with an `O` and if it is archived we mark it with an `A`.

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

```
hello_proof---../hello_proof.why?---HelloProof?-+-G3-+-Simplify (1.5.4)?
                                                 |    `-Alt-Ergo (0.94)
                                                 |-G2?-+-split_goal?-+-G2.2-+-Simplify (1.5.4)
                                                 |     |             |      `-Alt-Ergo (0.94)
                                                 |     |             `-G2.1?-+-Coq (8.3pl4)?
                                                 |     |                     |-Simplify (1.5.4)?
                                                 |     |                     `-Alt-Ergo (0.94)?
                                                 |     |-Simplify (1.5.4)?
                                                 |     `-Alt-Ergo (0.94)?
                                                 `-G1---Simplify (1.5.4)
```

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

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

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

```
== Number of goals ==
  total: 5  proved: 3

== Goals not proved ==
  +-- file ../hello_proof.why
    +-- theory HelloProof
      +-- goal G2
        +-- transformation split_goal
          +-- goal G2.1

== Goals proved by only one prover ==
  +-- file ../hello_proof.why
    +-- theory HelloProof
      +-- goal G1: Simplify (1.5.4) (0.00)
      +-- goal G3: Alt-Ergo (0.94) (0.00)

== Statistics per prover: number of proofs, time (minimum/maximum/average) in seconds ==
  Alt-Ergo (0.94)      :   2   0.00   0.00   0.00
  Simplify (1.5.4)     :   2   0.00   0.00   0.00
```

6.6.2 Command `latex`

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

The specific options are

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

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

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

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

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

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

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

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

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

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

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

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

Figure 6.1: Sample macros for the LaTeX command

Proof obligations		Alt-Ergo (0.94)	Coq (8.3pl4)	Simplify (1.5.4)
lemma G1				0.00
lemma G2		0.00		0.00
	lemma 1.	0.00	0.43	0.00
	lemma 2.	0.00		0.00
lemma G3		0.00		0.00

Figure 6.2: LaTeX table produced for the HelloProof example (style 1)

Figure 6.1 suggests some definitions for these macros, while Figures 6.2 and 6.3 show the tables obtained from the HelloProof example of Section 1, respectively with style 1 and 2.

6.6.3 Command `html`

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

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

The style 'table' outputs the contents of the session as a table, similar to the LaTeX output above. Figure 6.4 is the HTML table produced for the 'HelloProof' example, as typically shown in a Web browser. The gray cells filled with `--` just mean that the prover was not run on the corresponding goal. Green background means the result was "Valid",

Proof obligations	Alt-Ergo (0.94)	Coq (8.3pl4)	Simplify (1.5.4)
lemma `G1`			0.00
lemma `G2`	0.00		0.00
transformation `split_goal`			
subgoal 1	0.00	0.43	0.00
subgoal 2	0.00		0.00
lemma `G3`	0.00		0.00

Figure 6.3: LaTeX table produced for the HelloProof example (style 2)

Why3 Proof Results for Project "hello_proof"

Theory "HelloProof": not fully verified

Obligations	Alt-Ergo (0.94)	Coq (8.3pl4)	Simplify (1.5.4)
G1	---	---	0.00
G2	0.00	---	0.00
split_goal			
1.	0.00	0.43	0.00
2.	0.00	---	0.00
G3	0.00	---	0.00

Figure 6.4: HTML table produced for the HelloProof example

other cases are in orange background. The red background for a goal means that the goal was not proved.

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

Specific options for this command are as follows.

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

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

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

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

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

6.6.4 Commands modifying the proof attempts

The commands `mod`, `copy`, `copy-archive`, and `rm`, share the same set of options for selecting the proof attempts to work on:

`--filter-prover <prover>` selects only the proof attempt with the given prover. This option can be specified multiple times in order to select all the proofs that corresponds to any of the given provers.

`--filter-verified yes` selects only the proofs that are valid and not obsolete, while option `--filter-verified no` selects the ones that are not verified. `--filter-verified all`, the default, does not perform such a selection.

`--filter-verified-goal yes` restricts the selection to the proofs of verified goals (that does not mean that the proof is verified). Same for the other cases `no` and `all`.

`--filter-archived yes` restricts the selection to the proofs that are archived. Same for the other cases `no` and `all` except the default is `no`.

The commands `mod`, `copy`, and `copy-archive`, share the same set of options to specify the modification. The command `mod` modifies directly the proof attempt, `copy` copies the proof attempt before doing the modification, `copy-archive` marks the original proof attempt as archived. The options are:

`--set-obsolete` marks the selected proofs as obsolete.

`--set-archived` marks the selected proofs as archived.

`--unset-archived` removes the archived attribute from the selected proofs.

`--to-prover <prover>` modifies the prover, for example `--to-prover Alt-Ergo,0.94`. A conflict arises if a proof with this prover already exists. In this case, you can choose between four behaviors:

- replace the proof (`-f`, `--force`);
- do not replace the proof (`-n`, `--never`);
- replace the proof unless it is verified (valid and not obsolete) (`-c`, `--conservative`); this is the default;
- ask the user each time the case arises (`-i`, `--interactive`).

The command `rm` removes the selected proof attempts. The options `--interactive`, `--force`, and `--conservative`, can also be used to respectively ask before each suppression, suppress all the selected proofs (default), and remove only the proofs that are not verified. The macro option `--clean` corresponds to `--filter-verified-goal --conservative` and removes the proof attempts that are not verified but which correspond to verified goals.

The commands of this section do not accept by default to modify an obsolete session (as defined in 6.3.1). You need to add the option `-F` to force this behavior.

6.7 The `doc` Command

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

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

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

Options

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

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

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

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

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

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

Typesetting textual comments Some constructs are interpreted:

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

  1-6 a heading of level 1 to 6 respectively

  h raw HTML

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

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

6.8 The `execute` Command

Why3 can symbolically execute programs written using the WhyML language (extension `.mlw`). See also Section 8.1.

6.9 The `extract` Command

Why3 can extract programs written using the WhyML language (extension `.mlw`) to OCaml. See also Section 8.2.

6.10 The `realize` Command

Why3 can produce skeleton files for proof assistants that, once filled, realize the given theories. See also Section 9.2.

6.11 The `wc` Command

Why3 can give some token statistics about Why3 and WhyML source codes.

Chapter 7

Language Reference

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

7.1 Lexical Conventions

Lexical conventions are common to Why3 and WhyML.

7.1.1 Comments

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

7.1.2 Strings

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

7.1.3 Identifiers

Identifiers are non-empty sequences of characters among letters, digits, the underscore character and the quote character. They cannot start with a digit or a quote.

lalpha	::=	`a - z` | `_`
ualpha	::=	`A - Z`
alpha	::=	*lalpha* | *ualpha*
suffix	::=	(*alpha* | *digit* | `'`)*
lident	::=	*lalpha* *suffix*
uident	::=	*ualpha* *suffix*
ident	::=	*lident* | *uident*
lqualid	::=	*lident* | *uqualid* `.` *lident*
uqualid	::=	*uident* | *uqualid* `.` *uident*
qualid	::=	*ident* | *uqualid* `.` *ident*
digit-or-us	::=	`0 - 9` | `_`
alpha-no-us	::=	`a - z` | `A - Z`
suffix-nq	::=	(*alpha-no-us* | `'`* *digit-or-us*)* `'`*
lident-nq	::=	*lalpha* *suffix-nq*
uident-nq	::=	*ualpha* *suffix-nq*
ident-nq	::=	*lident-nq* | *uident-nq*

The syntax distinguishes identifiers that start with a lowercase or an uppercase letter (resp. *lident* and *uident*), and similarly, lowercase and uppercase qualified identifiers.

The restricted classes of identifiers denoted by *lident-nq* and *uident-nq* correspond to identifiers where the quote character cannot be followed by a letter. Identifiers where a quote is followed by a letter are reserved and cannot be used as identifier for declarations introduced by the user (see Section 7.2.4).

7.1.4 Constants

The syntax for constants is given in Figure 7.1. 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.

7.1.5 Operators

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

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

Figure 7.1: Syntax for constants.

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

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*).

7.1.6 Labels

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

label	::=	*string*
	|	`#` *filename* *digit*$^+$ *digit*$^+$ *digit*$^+$ `#`
filename	::=	*string*

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

7.2 The Why3 Language

7.2.1 Terms

The syntax for terms is given in Figure 7.2. The various constructs have the following priorities and associativities, from lowest to greatest priority:

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

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

7.2.2 Type Expressions

The syntax for type expressions is the following:

type	::=	*lqualid* *type**	type symbol
	|	`'` *lident*	type variable
	|	`()`	empty tuple type
	|	`(` *type* (`,` *type*)$^+$ `)`	tuple type
	|	`(` *type* `)`	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`).

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

Figure 7.2: Syntax for terms.

7.2.3 Formulas

The syntax for formulas is given Figure 7.3. The various constructs have the following priorities and associativities, from lowest to greatest priority:

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

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

Notice that there are two symbols for the conjunction: `/\` and `&&`, and similarly for disjunction. They are logically equivalent, but may be treated slightly differently by some transformations. For instance, `split` transforms the goal `A /\ B` into subgoals `A` and `B`, whereas it transforms `A && B` into subgoals `A` and `A -> B`. Similarly, it transforms `not (A || B)` into subgoals `not A` and `not ((not A) /\ B)`. The `by`/`so` connectives are proof indications. They are logically equivalent to their first argument, but may affect the result of some transformations. For instance, the `split_goal` transformations interpret those connectives as introduction of logical cuts (see 10.5.5 for details).

7.2.4 Theories

The syntax for theories is given on Figure 7.4 and 7.5.

Algebraic types

TO BE COMPLETED

Record types

TO BE COMPLETED

Range types

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

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

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

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

Figure 7.3: Syntax for formulas.

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

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

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

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

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

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

theory	::=	`theory` *uident-nq* *label** *decl** `end`
decl	::=	`type` *type-decl* (`with` *type-decl*)*
	|	`constant` *constant-decl*
	|	`function` *function-decl* (`with` *logic-decl*)*
	|	`predicate` *predicate-decl* (`with` *logic-decl*)*
	|	`inductive` *inductive-decl* (`with` *inductive-decl*)*
	|	`coinductive` *inductive-decl* (`with` *inductive-decl*)*
	|	`axiom` *ident-nq* `:` *formula*
	|	`lemma` *ident-nq* `:` *formula*
	|	`goal` *ident-nq* `:` *formula*
	|	`use` *imp-exp* *tqualid* (`as` *uident*)$^?$
	|	`clone` *imp-exp* *tqualid* (`as` *uident*)$^?$ *subst*$^?$
	|	`namespace` `import`$^?$ *uident-nq* *decl** `end`
logic-decl	::=	*function-decl*
	|	*predicate-decl*
constant-decl	::=	*lident-nq* *label** `:` *type*
	|	*lident-nq* *label** `:` *type* `=` *term*
function-decl	::=	*lident-nq* *label** *type-param** `:` *type*
	|	*lident-nq* *label** *type-param** `:` *type* `=` *term*
predicate-decl	::=	*lident-nq* *label** *type-param**
	|	*lident-nq* *label** *type-param** `=` *formula*
inductive-decl	::=	*lident-nq* *label** *type-param** `=` `|`$^?$ *ind-case* (`|` *ind-case*)*
ind-case	::=	*ident-nq* *label** `:` *formula*
imp-exp	::=	(`import` | `export`)$^?$
subst	::=	`with` (`,` *subst-elt*)$^+$
subst-elt	::=	`type` *lqualid* `=` *lqualid*
	|	`function` *lqualid* `=` *lqualid*
	|	`predicate` *lqualid* `=` *lqualid*
	|	`namespace` (*uqualid* | `.`) `=` (*uqualid* | `.`)
	|	`lemma` *qualid*
	|	`goal` *qualid*
tqualid	::=	*uident* | *ident* (`.` *ident*)* `.` *uident*
type-decl	::=	*lident-nq* *label** (`'` *lident-nq* *label**)* *type-defn*

Figure 7.4: Syntax for theories (part 1).

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

Figure 7.5: Syntax for theories (part 2).

Floating-point Types

A declaration of the form `type f = < float eb sb >` defines a type of floating-point numbers as specified by the IEEE-754 standard [1]. Here the literal `eb` represents the number of bits in the exponent and the literal `sb` the number of bits in the significand (including the hidden bit). Note that in order to make such a declaration the theory `real.Real` must be imported.

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

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

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

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

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

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

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

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

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

7.2.5 Files

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

file	::=	*theory**

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

Figure 7.6: Specification clauses in programs.

7.3 The WhyML Language

7.3.1 Specification

The syntax for specification clauses in programs is given in Figure 7.6. Within specifications, terms are extended with new constructs `old` and `at`:

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

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

7.3.2 Expressions

The syntax for program expressions is given in Figure 7.7 and Figure 7.8.

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

7.3.3 Modules

The syntax for modules is given in Figure 7.9. Any declaration which is accepted in a

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

Figure 7.7: Syntax for program expressions (part 1).

rec-defn	::=	*fun-defn* (`with` *fun-defn*)*
fun-defn	::=	`ghost`? *lident* *label** *fun-body*
fun-body	::=	*binder*+ (`:` *type*)? *spec** `=` *spec** *expr*
binder	::=	`ghost`? *lident* *label** | *param*
param	::=	`(` (`ghost`? *lident* *label**)+ `:` *type* `)`

Figure 7.8: Syntax for program expressions (part 2).

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

Figure 7.9: Syntax for modules.

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

7.3.4 Files

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

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

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

7.4 The Why3 Standard Library

The Why3 standard library provides general-purpose theories and modules, to be used in logic and/or programs. It can be browsed on-line at http://why3.lri.fr/stdlib/. Each file contains one or several theories and/or modules. To `use` or `clone` a theory/module `T` from file `file`, use the syntax `file.T`, since `file` is available in Why3's default load path. For instance, the theory of integers and the module of references are imported as follows:

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

Chapter 8

Executing WhyML Programs

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

Let us consider the program in Figure 3.1 on page 30 that computes the maximum and the sum of an array of integers. Let us assume it is contained in a file `maxsum.mlw`.

8.1 Interpreting WhyML Code

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

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

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

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

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

8.2 Compiling WhyML to OCaml

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

```
> mkdir dir
> why3 extract -D ocaml64 maxsum.mlw -o dir
```

The `extract` command requires the name of a driver, which indicates how theories/modules from the Why3 standard library are translated to OCaml. Here we assume a 64-bit architecture and thus we pass `ocaml64`. On a 32-bit architecture, we would use `ocaml32`

instead. Extraction also requires a target directory to be specified using option `-o`. Here we pass a freshly created directory `dir`.

Directory `dir` is now populated with a bunch of OCaml files, among which we find a file `maxsum__MaxAndSum.ml` containing the OCaml code for functions `max_sum` and `test`.

To compile it, we create a file `main.ml` containing a call to `test`, that is, for example,

```
let (s,m) = test () in
Format.printf "sum=%s, max=%s@."
  (Why3__BigInt.to_string s) (Why3__BigInt.to_string m)
```

and we pass both files `maxsum__MaxAndSum.ml` and `main.ml` to the OCaml compiler:

```
> cd dir
> ocamlopt zarith.cmxa why3extract.cmxa maxsum__MaxAndSum.ml main.ml
```

OCaml code extracted from Why3 must be linked with the library `why3extract.cmxa` that is shipped with Why3. It is typically stored in subdirectory `why3` of the OCaml standard library. Depending on the way Why3 was installed, it depends either on library `nums.cmxa` or `zarith.cmxa` for big integers. Above we assumed the latter. It is likely that additional options `-I` must be passed to the OCaml compiler for libraries `zarith.cmxa` and `why3extract.cmxa` to be found. For instance, it could be

```
> ocamlopt -I `ocamlfind query zarith` zarith.cmxa \
           -I `why3 --print-libdir`/why3 why3extract.cmxa \
           ...
```

To make the compilation process easier, one can write a `Makefile` which can include informations about Why3 configuration as follows.

```
WHY3SHARE=$(shell why3 --print-datadir)

include $(WHY3SHARE)/Makefile.config

maxsum:
        ocamlopt $(INCLUDE) $(BIGINTLIB).cmxa why3extract.cmxa \
                -o maxsum maxsum__MaxAndSum.ml main.ml
```

Chapter 9

Interactive Proof Assistants

9.1 Using an Interactive Proof Assistant to Discharge Goals

Instead of calling an automated theorem prover to discharge a goal, Why3 offers the possibility to call an interactive theorem prover instead. In that case, the interaction is decomposed into two distinct phases:

- Edition of a proof script for the goal, typically inside a proof editor provided by the external interactive theorem prover;
- Replay of an existing proof script.

An example of such an interaction is given in the tutorial section 1.2.

Some proof assistants offer more than one possible editor, *e.g.* a choice between the use of a dedicated editor and the use of the Emacs editor and the ProofGeneral mode. Selection of the preferred mode can be made in `why3ide` preferences, under the "Editors" tab.

9.2 Theory Realizations

Given a Why3 theory, one can use a proof assistant to make a *realization* of this theory, that is to provide definitions for some of its uninterpreted symbols and proofs for some of its axioms. This way, one can show the consistency of an axiomatized theory and/or make a connection to an existing library (of the proof assistant) to ease some proofs.

9.2.1 Generating a realization

Generating the skeleton for a theory is done by passing to the `realize` command a driver suitable for realizations, the names of the theories to realize, and a target directory.

```
why3 realize -D path/to/drivers/prover-realize.drv
             -T env_path.theory_name -o path/to/target/dir/
```

The theory is looked into the files from the environment, *e.g.* the standard library. If the theory is stored in a different location, option `-L` should be used.

The name of the generated file is inferred from the theory name. If the target directory already contains a file with the same name, Why3 extracts all the parts that it assumes to be user-edited and merges them in the generated file.

Note that Why3 does not track dependencies between realizations and theories, so a realization will become outdated if the corresponding theory is modified. It is up to the user to handle such dependencies, for instance using a `Makefile`.

9.2.2 Using realizations inside proofs

If a theory has been realized, the Why3 printer for the corresponding prover will no longer output declarations for that theory but instead simply put a directive to load the realization. In order to tell the printer that a given theory is realized, one has to add a meta declaration in the corresponding theory section of the driver.

```
theory env_path.theory_name
  meta "realized_theory" "env_path.theory_name", "optional_naming"
end
```

The first parameter is the theory name for Why3. The second parameter, if not empty, provides a name to be used inside generated scripts to point to the realization, in case the default name is not suitable for the interactive prover.

9.2.3 Shipping libraries of realizations

While modifying an existing driver file might be sufficient for local use, it does not scale well when the realizations are to be shipped to other users. Instead, one should create two additional files: a configuration file that indicates how to modify paths, provers, and editors, and a driver file that contains only the needed `meta "realized_theory"` declarations. The configuration file should be as follows.

```
[main]
loadpath="path/to/theories"

[prover_modifiers]
name="Coq"
option="-R path/to/vo/files Logical_directory"
driver="path/to/file/with/meta.drv"

[editor_modifiers coqide]
option="-R path/to/vo/files Logical_directory"

[editor_modifiers proofgeneral-coq]
option="--eval \"(setq coq-load-path (cons '(\\\"path/to/vo/files\\\" \
  \\\"Logical_directory\\\") coq-load-path))\""
```

This configuration file can be passed to Why3 thanks to the `--extra-config` option.

9.3 Coq

This section describes the content of the Coq files generated by Why3 for both proof obligations and theory realizations. When reading a Coq script, Why3 is guided by the presence of empty lines to split the script, so the user should refrain from removing empty lines around generated parts or adding empty lines inside them.

1. The header of the file contains all the library inclusions required by the driver file. Any user-made changes to this part will be lost when the file is regenerated by Why3. This part ends at the first empty line.

2. Abstract logic symbols are assumed with the vernacular directive `Parameter`. Axioms are assumed with the `Axiom` directive. When regenerating a script, Why3 assumes that all such symbols have been generated by a previous run. As a consequence, the user should not introduce new symbols with these two directives, as they would be lost.

3. Definitions of functions and inductive types in theories are printed in a block that starts with `(* Why3 assumption *)`. This comment should not be removed; otherwise Why3 will assume that the definition is user-made.

4. Finally, proof obligations and symbols to be realized are introduced by `(* Why3 goal *)`. The user is supposed to fill the script after the statement. Why3 assumes that the user-made part extends up to `Qed`, `Admitted`, `Save`, or `Defined`, whichever comes first. In the case of definitions, the original statement can be replaced by a `Notation` directive, in order to ease the usage of already defined symbols. Why3 also recognizes `Variable` and `Hypothesis` and preserves them; they should be used in conjunction with Coq's `Section` mechanism to realize theories that still need some abstract symbols and axioms.

Currently, the parser for Coq scripts is rather naive and does not know much about comments. For instance, Why3 can easily be confused by some terminating directive like `Qed` that would be present in a comment.

9.3.1 Coq Tactic

Why3 provides a Coq tactic to call external theorem provers as oracles.

Installation

You need Coq version 8.4 or greater. If this is the case, Why3's configuration detects it, then compiles and installs the Coq tactic. The Coq tactic is installed in

why3-lib-dir`/coq-tactic/`

where *why3-lib-dir* is Why3's library directory, as reported by `why3 --print-libdir`. This directory is automatically added to Coq's load path if you are calling Coq via Why3 (from `why3 ide`, `why3 replay`, etc.). If you are calling Coq by yourself, you need to add this directory to Coq's load path, either using Coq's command line option `-I` or by adding

`Add LoadPath "`*why3-lib-dir*`/coq-tactic/".`

to your `~/.coqrc` resource file.

Usage

The Coq tactic is called `why3` and is used as follows:

`why3 "`*prover-name*`"` [`timelimit` *n*].

The string *prover-name* identifies one of the automated theorem provers supported by Why3, as reported by `why3 --list-provers` (interactive provers excluded). The current goal is then translated to Why3's logic and the prover is called. If it reports the goal to be valid, then Coq's `admit` tactic is used to assume the goal. The prover is called with a time limit in seconds as given by Why3's configuration file (see Section 10.3). A different value may be given using the `timelimit` keyword.

Error messages.

The following errors may be reported by the Coq tactic.

`Not a first order goal` The Coq goal could not be translated to Why3's logic.

`Timeout` There was no answer from the prover within the given time limit.

`Don't know` The prover stopped without validating the goal.

`Invalid` The prover stopped, reporting the goal to be invalid.

`Failure` The prover failed. Depending on the message that follows, you may want to file a bug report, either to the Why3 developers or to the prover developers.

9.4 Isabelle/HOL

When using Isabelle from Why3, files generated from Why3 theories and goals are stored in a dedicated XML format. Those files should not be edited. Instead, the proofs must be completed in a file with the same name and extension `.thy`. This is the file that is opened when using "Edit" action in `why3 ide`.

9.4.1 Installation

You need version Isabelle2016-1 or Isabelle2017. Former versions are not supported. We assume below that your version is 2017, please replace 2017 by 2016-1 otherwise.

Isabelle must be installed before compiling Why3. After compilation and installation of Why3, you must manually add the path

```
<Why3 lib dir>/isabelle
```

into either the user file

```
.isabelle/Isabelle2017/etc/components
```

or the system-wide file

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

9.4.2 Usage

The most convenient way to call Isabelle for discharging a Why3 goal is to start the Isabelle/jedit interface in server mode. In this mode, one must start the server once, before launching `why3 ide`, using

```
isabelle why3_jedit
```

Then, inside a `why3 ide` session, any use of "Edit" will transfer the file to the already opened instance of jEdit. When the proof is completed, the user must send back the edited proof to `why3 ide` by closing the opened buffer, typically by hitting `Ctrl-w`.

9.4.3 Realizations

Realizations must be designed in some `.thy` as follows. The realization file corresponding to some Why3 file `f.why` should have the following form.

```
theory Why3_f
imports Why3_Setup
begin

section {* realization of theory T *}

why3_open "f/T.xml"

why3_vc <some lemma>
<proof>

why3_vc <some other lemma> by proof

[...]

why3_end
```

See directory `lib/isabelle` for examples.

9.5 PVS

9.5.1 Installation

You need version 6.0.

9.5.2 Usage

When a PVS file is regenerated, the old version is split into chunks, according to blank lines. Chunks corresponding to Why3 declarations are identified with a comment starting with `% Why3`, *e.g.*

```
% Why3 f
f(x: int) : int
```

Other chunks are considered to be user PVS declarations. Thus a comment such as `% Why3 f` must not be removed; otherwise, there will be two declarations for `f` in the next version of the file (one being regenerated and another one considered to be a user-edited chunk).

9.5.3 Realization

The user is allowed to perform the following actions on a PVS realization:

- give a definition to an uninterpreted symbol (type, function, or predicate symbol), by adding an equal sign (`=`) and a right-hand side to the definition. When the declaration is regenerated, the left-hand side is updated and the right-hand side is reprinted as is. In particular, the names of a function or predicate arguments should

not be modified. In addition, the `MACRO` keyword may be inserted and it will be kept in further generations.

- turn an axiom into a lemma, that is to replace the PVS keyword `AXIOM` with either `LEMMA` or `THEOREM`.
- insert anything between generated declarations, such as a lemma, an extra definition for the purpose of a proof, an extra `IMPORTING` command, etc. Do not forget to surround these extra declarations with blank lines.

Why3 makes some effort to merge new declarations with old ones and with user chunks. If it happens that some chunks could not be merged, they are appended at the end of the file, in comments.

Chapter 10

Technical Informations

10.1 Structure of Session Files

The proof session state is stored in an XML file named `<dir>/why3session.xml`, where `<dir>` is the directory of the project. The XML file follows the DTD given in `share/why3session.dtd` and reproduced below.

```
<!ELEMENT why3session (prover*, file*)>
<!ATTLIST why3session shape_version CDATA #IMPLIED>

<!ELEMENT prover EMPTY>
<!ATTLIST prover id CDATA #REQUIRED>
<!ATTLIST prover name CDATA #REQUIRED>
<!ATTLIST prover version CDATA #REQUIRED>
<!ATTLIST prover alternative CDATA #IMPLIED>
<!ATTLIST prover timelimit CDATA #IMPLIED>
<!ATTLIST prover memlimit CDATA #IMPLIED>
<!ATTLIST prover steplimit CDATA #IMPLIED>

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

<!ELEMENT theory (label*,goal*)>
<!ATTLIST theory name CDATA #REQUIRED>
<!ATTLIST theory expanded CDATA #IMPLIED>
<!ATTLIST theory verified CDATA #IMPLIED>
<!ATTLIST theory sum CDATA #IMPLIED>
<!ATTLIST theory locfile CDATA #IMPLIED>
<!ATTLIST theory loclnum CDATA #IMPLIED>
<!ATTLIST theory loccnumb CDATA #IMPLIED>
<!ATTLIST theory loccnume CDATA #IMPLIED>

<!ELEMENT goal (label*, proof*, transf*, metas*)>
<!ATTLIST goal name CDATA #REQUIRED>
<!ATTLIST goal expl CDATA #IMPLIED>
<!ATTLIST goal sum CDATA #IMPLIED>
```

```
<!ATTLIST goal shape CDATA #IMPLIED>
<!ATTLIST goal proved CDATA #IMPLIED>
<!ATTLIST goal expanded CDATA #IMPLIED>
<!ATTLIST goal locfile CDATA #IMPLIED>
<!ATTLIST goal loclnum CDATA #IMPLIED>
<!ATTLIST goal loccnumb CDATA #IMPLIED>
<!ATTLIST goal loccnume CDATA #IMPLIED>

<!ELEMENT proof (result|undone|internalfailure|unedited)>
<!ATTLIST proof prover CDATA #REQUIRED>
<!ATTLIST proof timelimit CDATA #IMPLIED>
<!ATTLIST proof memlimit CDATA #IMPLIED>
<!ATTLIST proof steplimit CDATA #IMPLIED>
<!ATTLIST proof edited CDATA #IMPLIED>
<!ATTLIST proof obsolete CDATA #IMPLIED>
<!ATTLIST proof archived CDATA #IMPLIED>

<!ELEMENT result EMPTY>
<!ATTLIST result status (valid|invalid|unknown|timeout|outofmemory|steplimitexceeded|fail
<!ATTLIST result time CDATA #IMPLIED>
<!ATTLIST result steps CDATA #IMPLIED>

<!ELEMENT undone EMPTY>
<!ELEMENT unedited EMPTY>

<!ELEMENT internalfailure EMPTY>
<!ATTLIST internalfailure reason CDATA #REQUIRED>

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

<!ELEMENT label EMPTY>
<!ATTLIST label name CDATA #REQUIRED>

<!ELEMENT metas (ts_pos*,ls_pos*,pr_pos*,meta*,goal)>
<!ATTLIST metas expanded CDATA #IMPLIED>
<!ATTLIST metas proved CDATA #IMPLIED>

<!ELEMENT ts_pos (ip_library*,ip_qualid+)>
<!ATTLIST ts_pos name CDATA #REQUIRED>
<!ATTLIST ts_pos arity CDATA #REQUIRED>
<!ATTLIST ts_pos id CDATA #REQUIRED>
<!ATTLIST ts_pos ip_theory CDATA #REQUIRED>

<!ELEMENT ls_pos (ip_library*,ip_qualid+)>
<!ATTLIST ls_pos name CDATA #REQUIRED>
<!ATTLIST ls_pos id CDATA #REQUIRED>
<!ATTLIST ls_pos ip_theory CDATA #REQUIRED>
```

```
<!ELEMENT pr_pos (ip_library*,ip_qualid+)>
<!ATTLIST pr_pos name CDATA #REQUIRED>
<!ATTLIST pr_pos id CDATA #REQUIRED>
<!ATTLIST pr_pos ip_theory CDATA #REQUIRED>

<!ELEMENT ip_library EMPTY>
<!ATTLIST ip_library name CDATA #REQUIRED>

<!ELEMENT ip_qualid EMPTY>
<!ATTLIST ip_qualid name CDATA #REQUIRED>

<!ELEMENT meta (meta_arg_ty*, meta_arg_ts*, meta_arg_ls*,
         meta_arg_pr*, meta_arg_str*, meta_arg_int*)>
<!ATTLIST meta name CDATA #REQUIRED>

<!ELEMENT meta_args_ty (ty_var|ty_app)>

<!ELEMENT ty_var EMPTY>
<!ATTLIST ty_var id CDATA #REQUIRED>

<!ELEMENT ty_app (ty_var*,ty_app*)>
<!ATTLIST ty_app id CDATA #REQUIRED>

<!ELEMENT meta_arg_ts EMPTY>
<!ATTLIST meta_arg_ts id CDATA #REQUIRED>

<!ELEMENT meta_arg_ls EMPTY>
<!ATTLIST meta_arg_ls id CDATA #REQUIRED>

<!ELEMENT meta_arg_pr EMPTY>
<!ATTLIST meta_arg_pr id CDATA #REQUIRED>

<!ELEMENT meta_arg_str EMPTY>
<!ATTLIST meta_arg_str val CDATA #REQUIRED>

<!ELEMENT meta_arg_int EMPTY>
<!ATTLIST meta_arg_int val CDATA #REQUIRED>
```

10.2 Prover Detection

The data configuration for the automatic detection of installed provers is stored in the file `provers-detection-data.conf` typically located in directory `/usr/local/share/why3` after installation. The content of this file is reproduced below.

```
[ATP alt-ergo]
name = "Alt-Ergo"
exec = "alt-ergo"
exec = "alt-ergo-1.30"
exec = "alt-ergo-1.01"
version_switch = "-version"
```

```
version_regexp = "^\\([0-9.]+\\)$"
version_ok = "1.30"
version_old = "1.01"
command = "%e -timelimit %t %f"
command_steps = "%e -steps-bound %S %f"
driver = "alt_ergo"
editor = "altgr-ergo"
use_at_auto_level = 1

[ATP alt-ergo]
name = "Alt-Ergo"
exec = "alt-ergo"
exec = "alt-ergo-0.99.1"
exec = "alt-ergo-0.95.2"
version_switch = "-version"
version_regexp = "^\\([0-9.]+\\)$"
version_old = "0.99.1"
version_old = "0.95.2"
command = "%e -no-rm-eq-existential -timelimit %t %f"
command_steps = "%e -no-rm-eq-existential -steps-bound %S %f"
driver = "alt_ergo"
editor = "altgr-ergo"

[ATP alt-ergo-prv]
name = "Alt-Ergo"
exec = "alt-ergo"
exec = "alt-ergo-1.20.prv"
exec = "alt-ergo-1.10.prv"
exec = "alt-ergo-1.00.prv"
version_switch = "-version"
version_regexp = "^\\([0-9.]+\\(-dev\\|prv\\)?\\)$"
version_old = "1.20.prv"
version_old = "1.10.prv"
version_old = "1.00.prv"
command = "%e -timelimit %t %f"
command_steps = "%e -steps-bound %S %f"
driver = "alt_ergo"
editor = "altgr-ergo"

# CVC4 version 1.5
[ATP cvc4-15]
name = "CVC4"
exec = "cvc4"
exec = "cvc4-1.5"
version_switch = "--version"
version_regexp = "This is CVC4 version \\([^ \n\r]+\\)"
version_ok = "1.5"
driver = "cvc4_15"
# --random-seed=42 is not needed as soon as --random-freq=0.0 by default
# to try: --inst-when=full-last-call
command = "%e --tlimit-per=%t000 --lang=smt2 %f"
command_steps = "%e --stats --rlimit=%S --lang=smt2 %f"
use_at_auto_level = 1

# CVC4 version 1.4, using SMTLIB fixed-size bitvectors
[ATP cvc4]
name = "CVC4"
exec = "cvc4"
exec = "cvc4-1.4"
version_switch = "--version"
version_regexp = "This is CVC4 version \\([^ \n\r]+\\)"
```

```
version_old = "1.4"
driver = "cvc4_14"
# --random-seed=42 is not needed as soon as --random-freq=0.0 by default
# to try: --inst-when=full-last-call
# --rlimit=%S : cvc4 1.4 DOES NOT accept -1 as argument
# cvc4 1.4 does not print steps used in --stats anyway
command = "%e --tlimit-per=%t000 --lang=smt2 %f"
use_at_auto_level = 1

# CVC4 version 1.4, not using SMTLIB bitvectors
[ATP cvc4]
name = "CVC4"
alternative = "noBV"
exec = "cvc4"
exec = "cvc4-1.4"
version_switch = "--version"
version_regexp = "This is CVC4 version \\([^ \n\r]+\\)"
version_old = "1.4"
driver = "cvc4"
# --random-seed=42 is not needed as soon as --random-freq=0.0 by default
# to try: --inst-when=full-last-call
# --rlimit=%S : cvc4 1.4 DOES NOT accept -1 as argument
# cvc4 1.4 does not print steps used in --stats anyway
command = "%e --tlimit-per=%t000 --lang=smt2 %f"


# CVC4 version 1.0 to 1.3
[ATP cvc4]
name = "CVC4"
exec = "cvc4"
exec = "cvc4-1.3"
exec = "cvc4-1.2"
exec = "cvc4-1.1"
exec = "cvc4-1.0"
version_switch = "--version"
version_regexp = "This is CVC4 version \\([^ \n\r]+\\)"
version_old = "1.3"
version_old = "1.2"
version_old = "1.1"
version_old = "1.0"
driver = "cvc4"
command = "%e --lang=smt2 %f"

# Psyche version 2.x
[ATP psyche]
name = "Psyche"
exec = "psyche"
exec = "psyche-2.02"
version_switch = "-version"
version_regexp = "\\([^ \n\r]+\\)"
version_ok = "2.0"
driver = "psyche"
command = "%e -gplugin dpll_wl %f"

# CVC3 versions 2.4.x
[ATP cvc3]
name = "CVC3"
exec = "cvc3"
exec = "cvc3-2.4.1"
exec = "cvc3-2.4"
version_switch = "-version"
```

```
version_regexp = "This is CVC3 version \\([^ \n]+\\)"
version_ok = "2.4.1"
version_old = "2.4"
# the -timeout option is unreliable in CVC3 2.4.1
command = "%e -seed 42 %f"
driver = "cvc3"

# CVC3 versions 2.x
[ATP cvc3]
name = "CVC3"
exec = "cvc3"
exec = "cvc3-2.2"
exec = "cvc3-2.1"
version_switch = "-version"
version_regexp = "This is CVC3 version \\([^ \n]+\\)"
version_old = "2.2"
version_old = "2.1"
command = "%e -seed 42 -timeout %t %f"
driver = "cvc3"

[ATP yices]
name = "Yices"
exec = "yices"
exec = "yices-1.0.38"
version_switch = "--version"
version_regexp = "\\([^ \n]+\\)"
version_ok = "1.0.38"
version_old = "^1\.0\.3[0-7]$"
version_old = "^1\.0\.2[5-9]$"
version_old = "^1\.0\.2[0-4]$"
version_old = "^1\.0\.1\.*$"
command = "%e"
driver = "yices"

[ATP yices-smt2]
name = "Yices"
exec = "yices-smt2"
exec = "yices-smt2-2.3.0"
version_switch = "--version"
version_regexp = "^Yices \\([^ \n]+\\)$"
version_ok = "2.3.0"
command = "%e"
driver = "yices-smt2"

[ATP eprover]
name = "Eprover"
exec = "eprover"
exec = "eprover-2.0"
exec = "eprover-1.9.1"
exec = "eprover-1.9"
exec = "eprover-1.8"
exec = "eprover-1.7"
exec = "eprover-1.6"
exec = "eprover-1.5"
exec = "eprover-1.4"
version_switch = "--version"
version_regexp = "E \\([-0-9.]+\\) [^\n]+"
version_ok = "2.0"
version_old = "1.9.1-001"
version_old = "1.9"
version_old = "1.8-001"
```

```
version_old = "1.7"
version_old = "1.6"
version_old = "1.5"
version_old = "1.4"
command = "%e -s -R -xAuto -tAuto --cpu-limit=%t --tstp-in %f"
driver = "eprover"
use_at_auto_level = 2

[ATP gappa]
name = "Gappa"
exec = "gappa"
exec = "gappa-1.3.2"
exec = "gappa-1.3.0"
exec = "gappa-1.2.2"
exec = "gappa-1.2.0"
exec = "gappa-1.1.1"
exec = "gappa-1.1.0"
exec = "gappa-1.0.0"
exec = "gappa-0.16.1"
exec = "gappa-0.14.1"
version_switch = "--version"
version_regexp = "Gappa \\([^ \n]*\\)"
version_ok = "^1\.[0-3]\..+$"
version_old = "^0\.1[1-8]\..+$"
command = "%e -Eprecision=70"
driver = "gappa"

[ATP mathsat]
name = "MathSAT5"
exec = "mathsat"
exec = "mathsat-5.2.2"
version_switch = "-version"
version_regexp = "MathSAT5 version \\([^ \n]+\\)"
version_ok = "5.2.2"
command = "%e -input=smt2 -model -random_seed=80"
driver = "mathsat"

[ATP simplify]
name = "Simplify"
exec = "Simplify"
exec = "simplify"
exec = "Simplify-1.5.4"
exec = "Simplify-1.5.5"
version_switch = "-version"
version_regexp = "Simplify version \\([^ \n,]+\\)"
version_old = "1.5.5"
version_old = "1.5.4"
command = "%e %f"
driver = "simplify"

[ATP metis]
name = "Metis"
exec = "metis"
version_switch = "-v"
version_regexp = "metis \\([^ \n,]+\\)"
version_ok = "2.3"
command = "%e --time-limit %t %f"
driver = "metis"

[ATP metitarski]
name = "MetiTarski"
```

```
exec = "metit"
exec = "metit-2.4"
exec = "metit-2.2"
version_switch = "-v"
version_regexp = "MetiTarski \\([^ \n,]+\\)"
version_ok = "2.4"
version_old = "2.2"
command = "%e --time %t %f"
driver = "metitarski"

[ATP polypaver]
name = "PolyPaver"
exec = "polypaver"
exec = "polypaver-0.3"
version_switch = "--version"
version_regexp = "PolyPaver \\([0-9.]+\\) (c)"
version_ok = "0.3"
command = "%e -d 2 -m 10 --time=%t %f"
driver = "polypaver"

[ATP spass]
name = "Spass"
exec = "SPASS"
exec = "SPASS-3.7"
version_switch = " | grep 'SPASS V'"
version_regexp = "SPASS V \\([^ \n\t]+\\)"
version_ok = "3.7"
command = "%e -TPTP -PGiven=0 -PProblem=0 -TimeLimit=%t %f"
driver = "spass"
use_at_auto_level = 2

[ATP spass]
name = "Spass"
exec = "SPASS"
exec = "SPASS-3.8ds"
version_switch = " | grep 'SPASS[^ \\n\\t]* V'"
version_regexp = "SPASS[^ \n\t]* V \\([^ \n\t]+\\)"
version_ok = "3.8ds"
command = "%e -Isabelle=1 -PGiven=0 -TimeLimit=%t %f"
driver = "spass_types"
use_at_auto_level = 2

[ATP vampire]
name = "Vampire"
exec = "vampire"
exec = "vampire-0.6"
version_switch = "--version"
version_regexp = "Vampire \\([0-9.]+\\)"
command = "%e -t %t"
driver = "vampire"
version_ok = "0.6"

[ATP princess]
name = "Princess"
exec = "princess"
exec = "princess-2015-12-07"
# version_switch = "-h"
# version_regexp = "(CASC version \\([0-9-]+\\))"
version_regexp = "(release \\([0-9-]+\\))"
command = "%e -timeout=%t %f"
driver = "princess"
```

```
# version_ok = "2013-05-13"
version_ok = "2015-12-07"

[ATP beagle]
name = "Beagle"
exec = "beagle"
exec = "beagle-0.4.1"
# version_switch = "-h"
version_regexp = "version \\([0-9.]+\\)"
command = "%e %f"
driver = "beagle"
version_ok = "0.4.1"

[ATP verit]
name = "veriT"
exec = "veriT"
exec = "veriT-201410"
version_switch = "--version"
version_regexp = "version \\([^ \n\r]+\\)"
command = "%e --disable-print-success %f"
driver = "verit"
version_ok = "201410"

[ATP verit]
name = "veriT"
exec = "veriT"
exec = "veriT-201310"
version_switch = "--version"
version_regexp = "version \\([^ \n\r]+\\)"
command = "%e --disable-print-success --enable-simp \
--enable-unit-simp --enable-simp-sym --enable-unit-subst-simp --enable-bclause %f"
driver = "verit"
version_old = "201310"

# Z3 >= 4.4.0, with BV support
[ATP z3]
name = "Z3"
exec = "z3"
exec = "z3-4.5.0"
exec = "z3-4.4.1"
exec = "z3-4.4.0"
version_switch = "-version"
version_regexp = "Z3 version \\([^ \n\r]+\\)"
version_ok = "4.5.0"
version_ok = "4.4.1"
version_ok = "4.4.0"
driver = "z3_440"
command = "%e -smt2 sat.random_seed=42 nlsat.randomize=false smt.random_seed=42 %f"
command_steps = "%e -smt2 sat.random_seed=42 nlsat.randomize=false smt.random_seed=42 memory_max_alloc_count=
use_at_auto_level = 1

# Z3 >= 4.4.0, without BV support
[ATP z3]
name = "Z3"
alternative = "noBV"
exec = "z3"
exec = "z3-4.5.0"
exec = "z3-4.4.1"
exec = "z3-4.4.0"
version_switch = "-version"
version_regexp = "Z3 version \\([^ \n\r]+\\)"
```

```
version_ok = "4.5.0"
version_ok = "4.4.1"
version_ok = "4.4.0"
driver = "z3_432"
command = "%e -smt2 sat.random_seed=42 nlsat.randomize=false smt.random_seed=42 %f"
command_steps = "%e -smt2 sat.random_seed=42 nlsat.randomize=false smt.random_seed=42 memory_max_alloc_count=

# Z3 4.3.2 does not support option global option -rs anymore.
# use settings given by "z3 -p" instead
# Z3 4.3.2 supports Datatypes
[ATP z3]
name = "Z3"
exec = "z3-4.3.2"
version_switch = "-version"
version_regexp = "Z3 version \\([^ \n\r]+\\)"
version_ok = "4.3.2"
driver = "z3_432"
command = "%e -smt2 sat.random_seed=42 nlsat.randomize=false smt.random_seed=42 %f"
command_steps = "%e -smt2 sat.random_seed=42 nlsat.randomize=false smt.random_seed=42 memory_max_alloc_count=

[ATP z3]
name = "Z3"
exec = "z3"
exec = "z3-4.3.1"
exec = "z3-4.3.0"
exec = "z3-4.2"
exec = "z3-4.1.2"
exec = "z3-4.1.1"
exec = "z3-4.0"
version_switch = "-version"
version_regexp = "Z3 version \\([^ \n\r]+\\)"
version_old = "4.3.1"
version_old = "4.3.0"
version_old = "4.2"
version_old = "4.1.2"
version_old = "4.1.1"
version_old = "4.0"
driver = "z3"
command = "%e -smt2 -rs:42 %f"

[ATP z3]
name = "Z3"
exec = "z3"
exec = "z3-3.2"
exec = "z3-3.1"
exec = "z3-3.0"
version_switch = "-version"
version_regexp = "Z3 version \\([^ \n\r]+\\)"
version_old = "3.2"
version_old = "3.1"
version_old = "3.0"
driver = "z3"
# the -T is unreliable in Z3 3.2
command = "%e -smt2 -rs:42 %f"

[ATP z3]
name = "Z3"
exec = "z3"
exec = "z3-2.19"
exec = "z3-2.18"
exec = "z3-2.17"
```

```
exec = "z3-2.16"
version_switch = "-version"
version_regexp = "Z3 version \\([^ \n\r]+\\)"
version_old = "^2\.2.+$"
version_old = "^2\.1[6-9]$"
driver = "z3"
command = "%e -smt2 -rs:42 \
PHASE_SELECTION=0 \
RESTART_STRATEGY=0 \
RESTART_FACTOR=1.5 \
QI_EAGER_THRESHOLD=100 \
ARITH_RANDOM_INITIAL_VALUE=true \
SORT_AND_OR=false \
CASE_SPLIT=3 \
DELAY_UNITS=true \
DELAY_UNITS_THRESHOLD=16 \
%f"
#Other Parameters given by Nikolaj Bjorner
#BV_REFLECT=true #arith?
#MODEL_PARTIAL=true
#MODEL_VALUE_COMPLETION=false
#MODEL_HIDE_UNUSED_PARTITIONS=false
#MODEL_V1=true
#ASYNC_COMMANDS=false
#NNF_SK_HACK=true

[ATP z3]
name = "Z3"
exec = "z3"
exec = "z3-2.2"
exec = "z3-2.1"
exec = "z3-1.3"
version_switch = "-version"
version_regexp = "Z3 version \\([^ \n\r]+\\)"
version_old = "^2\.1[0-5]$"
version_old = "^2\.[0-9]$"
version_old = "1.3"
command = "%e -smt %f"
driver = "z3_smtv1"

[ATP zenon]
name = "Zenon"
exec = "zenon"
exec = "zenon-0.8.0"
exec = "zenon-0.7.1"
version_switch = "-v"
version_regexp = "zenon version \\([^ \n\t]+\\)"
version_ok = "0.8.0"
version_ok = "0.7.1"
command = "%e -p0 -itptp -max-size %mM -max-time %ts %f"
driver = "zenon"

[ATP zenon_modulo]
name = "Zenon Modulo"
exec = "zenon_modulo"
version_switch = "-v"
version_regexp = "zenon_modulo version \\([0-9.]+\\)"
version_ok = "0.4.1"
command = "%e -p0 -itptp -max-size %mM -max-time %ts %f"
driver = "zenon_modulo"
```

```
[ATP iprover]
name = "iProver"
exec = "iprover"
exec = "iprover-0.8.1"
version_switch = " | grep iProver"
version_regexp = "iProver v\\([^ \n\t]+\\)"
version_ok = "0.8.1"
command = "%e --fof true --out_options none \
--time_out_virtual %t --clausifier /usr/bin/env --clausifier_options \
\"eprover --cnf --tstp-format \" %f"
driver = "iprover"

[ATP mathematica]
name = "Mathematica"
exec = "math"
version_switch = "-run \"Exit[]\""
version_regexp = "Mathematica \\([0-9.]+\\)"
version_ok = "9.0"
version_ok = "8.0"
version_ok = "7.0"
command = "%e -noprompt"
driver = "mathematica"

# Coq 8.6: do not limit memory
[ITP coq]
name = "Coq"
compile_time_support = true
exec = "coqtop -batch"
version_switch = "-v"
version_regexp = "The Coq Proof Assistant, version \\([^ \n]+\\)"
version_ok = "8.7.0"
version_ok = "8.6.1"
version_ok = "8.6"
command = "%e -I %l/coq-tactic -R %l/coq-tactic Why3 -R %l/coq Why3 -l %f"
driver = "coq"
editor = "coqide"

# Coq 8.5: do not limit memory
[ITP coq]
name = "Coq"
compile_time_support = true
exec = "coqtop -batch"
version_switch = "-v"
version_regexp = "The Coq Proof Assistant, version \\([^ \n]+\\)"
version_ok = "8.5pl3"
version_ok = "8.5pl2"
version_ok = "8.5pl1"
version_ok = "8.5"
command = "%e -R %l/coq-tactic Why3 -R %l/coq Why3 -l %f"
driver = "coq"
editor = "coqide"

[ITP coq]
name = "Coq"
compile_time_support = true
exec = "coqtop -batch"
version_switch = "-v"
version_regexp = "The Coq Proof Assistant, version \\([^ \n]+\\)"
version_ok = "^8\.4pl[1-6]$"
version_ok = "8.4"
command = "%e -R %l/coq-tactic Why3 -R %l/coq Why3 -l %f"
```

```
driver = "coq"
editor = "coqide"

[ITP pvs]
name = "PVS"
compile_time_support = true
exec = "pvs"
version_switch = "-version"
version_regexp = "PVS Version \\([^ \n]+\\)"
version_ok = "6.0"
version_bad = "^[0-5]\..+$"
command = "%l/why3-call-pvs %l proveit -f %f"
driver = "pvs"
in_place = true
editor = "pvs"

[ITP isabelle]
name = "Isabelle"
exec = "isabelle"
version_switch = "version"
version_regexp = "Isabelle\\([0-9]+\\(-[0-9]+\\)?\\)"
version_ok = "2016-1"
version_bad = "2017"
version_bad = "2016"
command = "%e why3 -b %f"
driver = "isabelle2016-1"
in_place = true
editor = "isabelle-jedit"

[ITP isabelle]
name = "Isabelle"
exec = "isabelle"
version_switch = "version"
version_regexp = "Isabelle\\([0-9]+\\(-[0-9]+\\)?\\)"
version_ok = "2017"
version_bad = "2016-1"
version_bad = "2016"
command = "%e why3 -b %f"
driver = "isabelle2017"
in_place = true
editor = "isabelle-jedit"

[editor pvs]
name = "PVS"
command = "%l/why3-call-pvs %l pvs %f"

[editor coqide]
name = "CoqIDE"
command = "coqide -I %l/coq-tactic -R %l/coq-tactic Why3 -R %l/coq Why3 %f"

[editor proofgeneral-coq]
name = "Emacs/ProofGeneral/Coq"
command = "emacs --eval \"(setq coq-load-path '((\\\"%l/coq-tactic\\\" \\\"Why3\\\") \
(\\\"%l/coq\\\" \\\"Why3\\\")))\" %f"

[editor isabelle-jedit]
name = "Isabelle/jEdit"
command = "isabelle why3 -i %f"

[editor altgr-ergo]
name = "AltGr-Ergo"
```

```
command = "altgr-ergo %f"

[shortcut shortcut1]
name="Alt-Ergo"
shortcut="altergo"
```

10.3 The `why3.conf` Configuration File

One can use a custom configuration file. The Why3 tools look for it in the following order:

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

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

The configuration file is a human-readable text file, which consists of association pairs arranged in sections. Figure 10.1 shows an example of configuration file.

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.

Sections contain associations `key=value`. A value is either an integer (*e.g.* `-555`), a boolean (`true`, `false`), or a string (*e.g.* `"emacs"`). Some specific keys can be attributed multiple values and are thus allowed to occur several times inside a given section. In that case, the relative order of these associations matter.

10.4 Drivers for External Provers

Drivers for external provers are readable files from directory `drivers`. Experimented users can modify them to change the way the external provers are called, in particular which transformations are applied to goals.

[TO BE COMPLETED LATER]

10.5 Transformations

This section documents the available transformations. We first describe the most important ones, and then we provide a quick documentation of the others, first the non-splitting ones, *e.g.* those which produce exactly one goal as result, and the others which produce any number of goals.

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

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

```
[main]
loadpath = "/usr/local/share/why3/theories"
loadpath = "/usr/local/share/why3/modules"
magic = 14
memlimit = 0
plugin = "/usr/local/lib/why3/plugins/tptp"
plugin = "/usr/local/lib/why3/plugins/genequlin"
plugin = "/usr/local/lib/why3/plugins/hypothesis_selection"
running_provers_max = 4
timelimit = 2

[ide]
default_editor = "editor %f"
error_color = "orange"
goal_color = "gold"
iconset = "fatcow"
intro_premises = true
premise_color = "chartreuse"
print_labels = false
print_locs = false
print_time_limit = false
saving_policy = 2
task_height = 404
tree_width = 512
verbose = 0
window_height = 1173
window_width = 1024

[prover]
command = "'why3-cpulimit' 0 %m -s coqtop -batch -I %l/coq-tactic -R %l/coq Why3 -l %f"
driver = "/usr/local/share/why3/drivers/coq.drv"
editor = "coqide"
in_place = false
interactive = true
name = "Coq"
shortcut = "coq"
version = "8.3pl4"

[prover]
command = "'why3-cpulimit' %t %m -s alt-ergo %f"
driver = "/usr/local/share/why3/drivers/alt_ergo_0.93.drv"
editor = ""
in_place = false
interactive = false
name = "Alt-Ergo"
shortcut = "altergo"
shortcut = "alt-ergo"
version = "0.93.1"

[editor coqide]
command = "coqide -I %l/coq-tactic -R %l/coq Why3 %f"
name = "CoqIDE"
```

Figure 10.1: Sample `why3.conf` file

10.5.1 Inlining definitions

Those transformations generally amount to replace some applications of function or predicate symbols with its definition.

inline_trivial expands and removes definitions of the form

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

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

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

inline_all expands all non-recursive definitions.

10.5.2 Induction Transformations

induction_ty_lex performs structural, lexicographic induction on goals involving universally quantified variables of algebraic data types, such as lists, trees, etc. For instance, it transforms the following goal

```
goal G: forall l: list 'a. length l >= 0
```

into this one:

```
goal G :
  forall l:list 'a.
     match l with
     | Nil -> length l >= 0
     | Cons a l1 -> length l1 >= 0 -> length l >= 0
     end
```

When induction can be applied to several variables, the transformation picks one heuristically. A label `"induction"` can be used to force induction over one particular variable, *e.g.* with

```
goal G: forall l1 "induction" l2 l3: list 'a.
        l1 ++ (l2 ++ l3) = (l1 ++ l2) ++ l3
```

induction will be applied on `l1`. If this label is attached to several variables, a lexicographic induction is performed on these variables, from left to right.

10.5.3 Simplification by Computation

These transformations simplify the goal by applying several kinds of simplification, described below. The transformations differ only by the kind of rules they apply:

compute_in_goal aggressively applies all known computation/simplification rules.

compute_specified performs rewriting using only built-in operators and user-provided rules.

The kinds of simplification are as follows.

- Computations with built-in symbols, *e.g.* operations on integers, when applied to explicit constants, are evaluated. This includes evaluation of equality when a decision can be made (on integer constants, on constructors of algebraic data types), Boolean evaluation, simplification of pattern-matching/conditional expression, extraction of record fields, and beta-reduction. At best, these computations reduce the goal to `true` and the transformations thus does not produce any sub-goal. For example, a goal like `6*7=42` is solved by those transformations.

- Unfolding of definitions, as done by `inline_goal`. Transformation `compute_in_goal` unfolds all definitions, including recursive ones. For `compute_specified`, the user can enable unfolding of a specific logic symbol by attaching the meta `rewrite_def` to the symbol.

```
function sqr (x:int) : int = x * x
meta "rewrite_def" function sqr
```

- Rewriting using axioms or lemmas declared as rewrite rules. When an axiom (or a lemma) has one of the forms

```
axiom a: forall ... t1 = t2
```

or

```
axiom a: forall ... f1 <-> f2
```

then the user can declare

```
meta "rewrite" prop a
```

to turn this axiom into a rewrite rule. Rewriting is always done from left to right. Beware that there is no check for termination nor for confluence of the set of rewrite rules declared.

Instead of using a meta, it is possible to declare an axiom as a rewrite rule by adding the label `"rewrite"` on the axiom name or on the axiom itself, e.g.:

```
axiom a "rewrite": forall ... t1 = t2
lemma b: "rewrite" forall ... f1 <-> f2
```

The second form allows some form of local rewriting, e.g.

```
lemma l: forall x y. ("rewrite" x = y) -> f x = f y
```

can be proved by `introduce_premises` followed by `"compute_specified"`.

Bound on the number of reductions The computations performed by these transformations can take an arbitrarily large number of steps, or even not terminate. For this reason, the number of steps is bounded by a maximal value, which is set by default to 1000. This value can be increased by another meta, *e.g.*

```
meta "compute_max_steps" 1_000_000
```

When this upper limit is reached, a warning is issued, and the partly-reduced goal is returned as the result of the transformation.

10.5.4 Other Non-Splitting Transformations

eliminate_algebraic replaces algebraic data types by first-order definitions [9].

eliminate_builtin removes definitions of symbols that are declared as builtin in the driver, *i.e.* with a "syntax" rule.

eliminate_definition_func replaces all function definitions with axioms.

eliminate_definition_pred replaces all predicate definitions with axioms.

eliminate_definition applies both transformations above.

eliminate_mutual_recursion replaces mutually recursive definitions with axioms.

eliminate_recursion replaces all recursive definitions with axioms.

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

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

eliminate_if applies both transformations above.

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

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

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

eliminate_let applies both transformations above.

encoding_smt encodes polymorphic types into monomorphic types [3].

encoding_tptp encodes theories into unsorted logic.

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

simplify_array automatically rewrites the task using the lemma `Select_eq` of theory `map.Map`.

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

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

```
function f : ... = ... g ...
with g : ... = e
```

becomes

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

if f does not occur in e.

simplify_trivial_quantification simplifies quantifications of the form

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

into

```
P(t)
```

when x does not occur in t. More generally, this simplification is applied whenever $x = t$ or $t = x$ appears in negative position.

simplify_trivial_quantification_in_goal is the same as above but it applies only in the goal.

split_premise replaces axioms in conjunctive form by an equivalent collection of axioms. In absence of case analysis labels (see `split_goal` for details), the number of axiom generated per initial axiom is linear in the size of that initial axiom.

split_premise_full is similar to `split_premise`, but it also converts the axioms to conjunctive normal form. The number of axioms generated per initial axiom may be exponential in the size of the initial axiom.

10.5.5 Other Splitting Transformations

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

split_goal changes conjunctive goals into the corresponding set of subgoals. In absence of case analysis labels, the number of subgoals generated is linear in the size of the initial goal.

Behavior on asymmetric connectives and by/so The transformation treats specially asymmetric and `by`/`so` connectives. Asymmetric conjunction `A && B` in goal position is handled as syntactic sugar for `A /\ (A -> B)`. The conclusion of the first subgoal can then be used to prove the second one.

Asymmetric disjunction `A || B` in hypothesis position is handled as syntactic sugar for `A \/ ((not A) /\ B)`. In particular, a case analysis on such hypothesis would give the negation of the first hypothesis in the second case.

The `by` connective is treated as a proof indication. In hypothesis position, `A by B` is treated as if it were syntactic sugar for its regular interpretation `A`. In goal position, it is treated as if `B` was an intermediate step for proving `A`. `A by B` is then replaced by `B` and the transformation also generates a side-condition subgoal `B -> A` representing the logical cut.

Although splitting stops at disjunctive points like symmetric disjunction and left-hand sides of implications, the occurrences of the `by` connective are not restricted. For instance:

- Splitting

```
goal G : (A by B) && C
```

generates the subgoals

```
goal G1 : B
goal G2 : A -> C
goal G3 : B -> A (* side-condition *)
```

- Splitting

```
goal G : (A by B) \/ (C by D)
```

generates

```
goal G1 : B \/ D
goal G2 : B -> A (* side-condition *)
goal G3 : D -> C (* side-condition *)
```

- Splitting

```
goal G : (A by B) || (C by D)
```

generates

```
goal G1 : B || D
goal G2 : B -> A        (* side-condition *)
goal G3 : B || (D -> C) (* side-condition *)
```

Note that due to the asymmetric disjunction, the disjunction is kept in the second side-condition subgoal.

- Splitting

```
goal G : exists x. P x by x = 42
```

generates

```
goal G1 : exists x. x = 42
goal G2 : forall x. x = 42 -> P x (* side-condition *)
```

Note that in the side-condition subgoal, the context is universally closed.

The `so` connective plays a similar role in hypothesis position, as it serves as a consequence indication. In goal position, `A so B` is treated as if it were syntactic sugar for its regular interpretation `A`. In hypothesis position, it is treated as if both `A` and `B` were true because `B` is a consequence of `A`. `A so B` is replaced by `A /\ B` and the transformation also generates a side-condition subgoal `A -> B` corresponding to the consequence relation between formula.

As with the `by` connective, occurrences of `so` are unrestricted. For instance:

- Splitting

```
goal G : (((A so B) \/ C) -> D) && E
```

generates

```
goal G1 : ((A /\ B) \/ C) -> D
goal G2 : (A \/ C -> D) -> E
goal G3 : A -> B              (* side-condition *)
```

- Splitting

```
goal G : A by exists x. P x so Q x so R x by T x
(* reads: A by (exists x. P x so (Q x so (R x by T x))) *)
```

generates

```
goal G1 : exists x. P x
goal G2 : forall x. P x -> Q x                (* side-condition *)
goal G3 : forall x. P x -> Q x -> T x         (* side-condition *)
goal G4 : forall x. P x -> Q x -> T x -> R x  (* side-condition *)
goal G5 : (exists x. P x /\ Q x /\ R x) -> A  (* side-condition *)
```

In natural language, this corresponds to the following proof scheme for `A`: There exists a `x` for which `P` holds. Then, for that witness `Q` and `R` also holds. The last one holds because `T` holds as well. And from those three conditions on `x`, we can deduce `A`.

Labels controlling the transformation The transformations in the split family can be controlled by using labels on formulas.

The label `"stop_split"` can be used to block the splitting of a formula. The label is removed after blocking, so applying the transformation a second time will split the formula. This is can be used to decompose the splitting process in several steps. Also, if a formula with this label is found in non-goal position, its `by`/`so` proof indication will be erased by the transformation. In a sense, formulas tagged by `"stop_split"` are handled as if they were local lemmas.

The label `"case_split"` can be used to force case analysis on hypotheses. For instance, applying `split_goal` on

```
goal G : ("case_split" A \/ B) -> C
```

generates the subgoals

```
goal G1 : A -> C
goal G2 : B -> C
```

Without the label, the transformation does nothing because undesired case analysis may easily lead to an exponential blow-up.

Note that the precise behavior of splitting transformations in presence of the `"case_split"` label is not yet specified and is likely to change in future versions.

split_all performs both `split_premise` and `split_goal`.

split_intro performs both `split_goal` and `introduce_premises`.

split_goal_full has a behavior similar to `split_goal`, but also converts the goal to conjunctive normal form. The number of subgoals generated may be exponential in the size of the initial goal.

split_all_full performs both `split_premise` and `split_goal_full`.

10.6 Proof Strategies

As seen in Section 6.3, the IDE provides a few buttons that trigger the run of simple proof strategies on the selected goals. Proof strategies can be defined using a basic assembly-style language, and put into the Why3 configuration file. The commands of this basic language are:

- `c` *p t m* calls the prover *p* with a time limit *t* and memory limit *m*. On success, the strategy ends, it continues to next line otherwise
- `t` *n lab* applies the transformation *n*. On success, the strategy continues to label *lab*, and is applied to each generated sub-goals. It continues to next line otherwise.
- `g` *lab* inconditionally jumps to label *lab*
- *lab*`:` declares the label *lab*. The default label `exit` allows to stop the program.

To examplify this basic programming language, we give below the default strategies that are attached to the default buttons of the IDE, assuming that the provers Alt-Ergo 1.30, CVC4 1.5 and Z3 4.5.0 were detected by the `why3 config --detect` command

Split is bound to the 1-line strategy

```
t split_goal_wp exit
```

Auto level 0 is bound to

```
c Z3,4.5.0, 1 1000
c Alt-Ergo,1.30, 1 1000
c CVC4,1.5, 1 1000
```

The three provers are tried for a time limit of 1 second and memory limit of 1 Gb, each in turn. This is a perfect strategy for a first attempt to discharge a new goal.

Auto level 1 is bound to

```
start:
c Z3,4.5.0, 1 1000
c Alt-Ergo,1.30, 1 1000
c CVC4,1.5, 1 1000
t split_goal_wp start
c Z3,4.5.0, 10 4000
c Alt-Ergo,1.30, 10 4000
c CVC4,1.5, 10 4000
```

The three provers are first tried for a time limit of 1 second and memory limit of 1 Gb, each in turn. If none of them succeed, a split is attempted. If the split works then the same strategy is retried on each sub-goals. If the split does not succeed, the provers are tried again with a larger limits.

Auto level 2 is bound to

```
start:
c Z3,4.5.0, 1 1000
c Eprover,2.0, 1 1000
c Spass,3.7, 1 1000
c Alt-Ergo,1.30, 1 1000
c CVC4,1.5, 1 1000
t split_goal_wp start
c Z3,4.5.0, 5 2000
c Eprover,2.0, 5 2000
c Spass,3.7, 5 2000
c Alt-Ergo,1.30, 5 2000
c CVC4,1.5, 5 2000
t introduce_premises afterintro
afterintro:
t inline_goal afterinline
g trylongertime
afterinline:
t split_goal_wp start
trylongertime:
c Z3,4.5.0, 30 4000
c Eprover,2.0, 30 4000
c Spass,3.7, 30 4000
c Alt-Ergo,1.30, 30 4000
c CVC4,1.5, 30 4000
```

Notice that now 5 provers are used. The provers are first tried for a time limit of 1 second and memory limit of 1 Gb, each in turn. If none of them succeed, a split is attempted. If the split works then the same strategy is retried on each sub-goals. If the split does not succeed, the prover are tried again with limits of 5 s and 2 Gb. If all fail, we attempt the transformation of introduction of premises in the context, followed by an inlining of the definitions in the goals. We then attempt a split again, if the split succeeds, we restart from the beginning, if it fails then provers are tried again with 30s and 4 Gb.

Part III

Appendix

Appendix A

Release Notes

A.1 Release Notes for version 0.80: syntax changes w.r.t. 0.73

The syntax of WhyML programs changed in release 0.80. The table in Figure A.1 summarizes the changes.

version 0.73	**version 0.80**
`type t = {| field : int |}`	`type t = { field : int }`
`{| field = 5 |}`	`{ field = 5 }`
`use import module M`	`use import M`

`let rec f (x:int) (y:int) : t`
`  variant { t } with rel =`
`  { P }`
`  e`
`  { Q }`
`  | Exc1 -> { R1 }`
`  | Exc2 n -> { R2 }`
	`let rec f (x:int) (y:int) : t`
	`  variant { t with rel }`
	`  requires { P }`
	`  ensures { Q }`
	`  raises { Exc1 -> R1`
	`  | Exc2 n -> R2 }`
	`  = e`

`val f (x:int) (y:int) :`
`  { P }`
`  t`
`  writes a b`
`  { Q }`
`  | Exc1 -> { R1 }`
`  | Exc2 n -> { R2 }`
	`val f (x:int) (y:int) : t`
	`  requires { P }`
	`  writes { a, b }`
	`  ensures { Q }`
	`  raises { Exc1 -> R1`
	`  | Exc2 n -> R2 }`

`val f : x:int -> y:int ->`
`  { P }`
`  t`
`  writes a b`
`  { Q }`
`  | Exc1 -> { R1 }`
`  | Exc2 n -> { R2 }`
	`val f (x y:int) : t`
	`  requires { P }`
	`  writes { a, b }`
	`  ensures { Q }`
	`  raises { Exc1 -> R1`
	`  | Exc2 n -> R2 }`

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

Figure A.1: Syntax changes from version 0.73 to version 0.80

A.2 Summary of Changes w.r.t. Why 2

The main new features with respect to Why 2.xx are the following.

1. Completely redesigned input syntax for logic declarations
   - new syntax for terms and formulas
   - enumerated and algebraic data types, pattern matching
   - recursive definitions of logic functions and predicates, with termination checking
   - inductive definitions of predicates
   - declarations are structured in components called "theories", which can be reused and instantiated
2. More generic handling of goals and lemmas to prove
   - concept of proof task
   - generic concept of task transformation
   - generic approach for communicating with external provers
3. Source code organized as a library with a documented API, to allow access to Why3 features programmatically.
4. GUI with new features with respect to the former GWhy
   - session save and restore
   - prover calls in parallel
   - splitting, and more generally applying task transformations, on demand
   - ability to edit proofs for interactive provers (Coq only for the moment) on any subtask
5. Extensible architecture via plugins
   - users can define new transformations
   - users can add connections to additional provers

Bibliography

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

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

[3] 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. doi:10.1145/1512464.1512466.

[4] S. Conchon and E. Contejean. The Alt-Ergo automatic theorem prover, 2008. URL: http://alt-ergo.lri.fr/.

[5] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005. doi:10.1145/1066100.1066102.

[6] 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. doi:10.1007/978-3-540-73368-3_21.

[7] D. Hauzar, C. Marché, and Y. Moy. Counterexamples from proof failures in SPARK. In R. De Nicola and E. Kühn, editors, *Software Engineering and Formal Methods*, Lecture Notes in Computer Science, pages 215–233, Vienna, Austria, 2016. doi:10.1007/978-3-319-41591-8_15.

[8] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[9] A. Paskevich. Algebraic types and pattern matching in the logical language of the Why verification platform. Technical Report 7128, INRIA, 2009. URL: http://hal.inria.fr/inria-00439232.

[10] N. Shankar and P. Mueller. Verified Software: Theories, Tools and Experiments (VSTTE'10). Software Verification Competition, August 2010. URL: http://www.macs.hw.ac.uk/vstte10/Competition.html.

List of Figures

Index