The WHY verification tool

Tutorial and Reference Manual

Version 2.04

Jean-Christophe Filliâtre

August 2, 2007

Contents

Foreword

A computer program is telling the machine *how* to compute some output from its input. It is not giving the meaning of this computation, nor the reason *why* the program is correct. In the best case, comments inserted in the code would give part of this information. A formal method gives you the ability to tell the machine both *how* to compute and *why* it is correct; hence the name of this tool, Why.

This manual is organized as follows. Chapter 1 gives an overview of Why, illustrating all features with one-line examples. Chapter 2 is a reference manual.

License

Availability

The Why tool is available from `http://why.lri.fr/`, in source and binary formats, together with this documentation and many examples.

Contact

There is a mailing list for Why, which is mainly used to announce the releases. See `http://lists.lri.fr/mailman/listinfo/why`.

Bug reports can be sent to `filliatr@lri.fr`.

Chapter 1

Quick tutorial

1.1 Principles

Basically, the Why tool takes annotated programs written in a very simple imperative programming language of its own, produces verification conditions and sends them to existing provers (proof assistants such as Coq, PVS, etc. or automatic provers such as Simplify, CVC Lite, etc.), as illustrated below:

$$\begin{array}{c}\text{annotated programs}\\ \downarrow\\ \textsf{Why}\\ \downarrow\\ \text{verification conditions} = \text{first-order formulae}\\ \downarrow\\ \textsf{Why}\\ \downarrow\\ \text{external provers}\end{array}$$

Simplify speaking, Why may be seen as an implementation of Dijkstra's weakeast preconditions. Its key advantages are the following:

- It is a verification condition generator (VCG) for a language specifically designed for the interpretation of existing programming languages. It includes the usual constructs of imperative programming and logical annotations (pre- and postconditions, loop invariants and variants, intermediate assertions) but also exceptions, recursive functions and polymorphism.

- The Why tool allows the declaration of logical *models* (types, functions, predicates and axioms) that can be used in programs and annotations. Thus it is a highly generic VCG. If one wants to model some kind of bounded precision arithmetic, for instance, it is as simple as declaring a new type and the corresponding operations (which may be defined later on the prover side, or left axiomatized).

- The Why tool supports a wide set of existing provers (see Figure 1.1). Thus verification conditions can be discharged by several provers independently. One may

	Prover	Command-line option
Proof assistants	Coq	--coq
	PVS	--pvs
	Isabelle/HOL	--isabelle
	HOL 4	--hol4
	HOL Light	--hol-light
	Mizar	--mizar
Automatic theorem provers	Ergo	--why
	Simplify	--simplify
	CVC Lite	--cvcl
	haRVey	--harvey
	Zenon	--zenon
	SMTlib (Yices, CVC 3)	--smtlib

Figure 1.1: Supported provers

for instance use an interactive proof assistant to focus only on VC that are not automatically discharged by some (or any) automatic prover. Actually, the Why tool can be simply used as a front-end to all these provers, independently of its VCG mechanism. Then it is nothing more than a common first-order logic "syntax" for these provers.

1.2 A trivial example

Here is a small example of Why input code:

```
logic min: int, int -> int
parameter r: int ref
let f (n:int) = r := min !r n { r <= r@ }
```

This code declares a function symbol `min` and gives its arity. Whatever the status of this function is on the prover side (primitive, user-defined, axiomatized, etc.), it simply needs to be declared in order to be used in the following of the code. The next line declares a parameter, that is a value that is not defined but simply *assumed* to exist i.e. to belong to the environment. Here the parameter has name `r` and is an integer reference (Why's concrete syntax is very close to Ocaml's syntax). The third line defines a function `f` taking a integer `n` as argument (the type has to be given since there is no type inference in Why) and assigning to `r` the value of `min !r n`. The function `f` has no precondition and a postcondition expressing that the final value of `r` is smaller than its initial value. The current value of a reference `x` is directly denoted by `x` within annotations (not `!x`) and within postconditions `x@` is the notation for the value of `x` in the prestate (i.e. at the precondition point).

Let us assume the three lines code above to be in file `test.why`. Then we can produce the proof obligations for this program, to be verified with Coq, using the following command line:

```
why --coq test.why
```

A Coq file test_why.v is produced which contains the statement of a single proof obligation, which looks like

```
Lemma f_po_1 :
  forall (n: Z),
  forall (r: Z),
  forall (r0: Z),
  forall (HW_1: r0 = (min r n)),
  r0 <= r.
Proof.
(* FILL PROOF HERE *)
Save.
```

The proof itself has to be filled in by the user. If the Why input code is modified and Why run again, only the statement of the proof obligation will be updated and the remaining of the file (including the proof) will be left unmodified. Assuming that min is adequately defined in Coq, the proof above is trivial.

Trying an automatic decision procedure instead of Coq is as easy as running Why with a different command line option. For instance, to use Simplify [?], we type in

```
why --simplify test.why
```

A Simplify input file test_why.sx is produced. But Simplify is not able to discharge the proof obligation, since the meaning of min is unknown for Simplify:

```
Simplify test_why.sx
...
1: Invalid
```

The user can edit the header of test_why.sx to insert an axiom for min. Alternatively, this axiom can be inserted directly in the Why input code:

```
logic min: int, int -> int
axiom min_ax: forall x,y:int. min(x,y) <= x
parameter r: int ref
let f (n:int) = {} r := min !r n { r <= r@ }
```

This way this axiom will be replicated in any prover selected by the user. When using Coq, it is even possible to prove this axiom, though it is not mandatory. With the addition of this axiom, Simplify is now able to discharge the proof obligation:

```
why --simplify test.why
Simplify test_why.sx
1: Valid.
```

1.3 A less trivial example: Dijkstra's Dutch flag

Dijkstra's Dutch flag is a classical algorithm which sorts an array where elements can have only three different values. Assuming that these values are the three colors blue, white and red, the algorithm restores the Dutch (or French :-) national flag within the array. This algorithm can be coded with a few lines of C, as follows:

```
typedef enum { BLUE, WHITE, RED } color;

void swap(int t[], int i, int j) { color c = t[i]; t[i] = t[j]; t[j] = c;}

void flag(int t[], int n) {
  int b = 0, i = 0, r = n;
  while (i < r) {
    switch (t[i]) {
    case BLUE: swap(t, b++, i++); break;
    case WHITE: i++; break;
    case RED: swap(t, --r, i); break;
    }
  }
}
```

We are going to show how to verify this algorithm—the *algorithm*, not the C code—using Why. First we introduce an abstract type `color` for the colors together with three values `blue`, `white` and `red`:

```
  type color

  logic blue : color
  logic white : color
  logic red : color
```

Such a new type is necessarily an *immutable* datatype. The only mutable values in Why are references (and they only contain immutable values).

Then we introduce another type `color_array` for purely applicative arrays containing colors:

```
  type color_array

  logic acc : color_array, int -> color
  logic upd : color_array, int, color -> color_array
```

To get the usual theory of applicative arrays, we can add the necessary axioms:

```
  axiom acc_upd_eq :
    forall t:color_array. forall i:int. forall c:color.
      acc(upd(t,i,c),i) = c
  axiom acc_upd_neq :
    forall t:color_array. forall i:int. forall j:int. forall c:color.
      j<>i -> acc(upd(t,i,c),j) = acc(t,j)
```

The program arrays will be references containing values of type color_array. In order to constraint accesses and updates to be performed within arrays bounds, we add a notion of array length and two "programs" get and set with adequate preconditions:

```
logic length : color_array -> int

axiom length_upd :
  forall t:color_array. forall i:int. forall c:color.
  length(upd(t,i,v)) = length(t)

parameter get :
  t:color_array ref -> i:int ->
    { 0<=i<length(t) } color reads t { result=acc(t,i) }

parameter set :
  t:color_array ref -> i:int -> c:color ->
    { 0<=i<length(t) } unit writes t { t=upd(t@,i,c) }
```

These two programs need not being defined (they are only here to insert assertions automatically), so we declare them as parameters. As illustrated above, a function type has named arguments, a pre- and a postcondition, a result type and an explicit declaration of its side-effects: the reads and writes keywords respectively declare the set of references possibly accessed and modified.

We are now in position to define the swap function:

```
let swap (t:color_array ref) (i:int) (j:int) =
  { 0 <= i < length(t) and 0 <= j < length(t) }
  let c = get t i in
  set t i (get t j);
  set t j c
  { t = upd(upd(t@,i,acc(t@,j)), j, acc(t@,i)) }
```

The precondition for swap states that the two indices i and j must lie within the array bounds and the postcondition is simply a rephrasing of the code on the model level i.e. on purely applicative arrays. Verifying the swap function is immediate.

Next we need to give the main function a specification. First, we need to express that the array only contains one of the three values blue, white and red. Indeed, nothing prevents the type color to be inhabitated with other values (there is no notion of inductive type in Why logic, since it is intended to be a common fragment of many tools, including many with no primitive notion of inductive types). So we define the following predicate is_color:

```
predicate is_color(c:color) = c=blue or c=white or c=red
```

Note that this predicate is given a *definition* in Why.

Second, we need to express the main function postcondition that is, for the final contents of the array, the property of being "sorted" but also the property of being a permutation of the initial contents of the array (a property usually neglected but clearly as important as the former). For this purpose, we introduce a predicate monochrome expressing that a set of successive elements is monochrome:

```
  predicate monochrome(t:color_array, i:int, j:int, c:color) =
    forall k:int. i<=k<j -> acc(t,k)=c
```

For the permutation property, we simply *declare* a predicate `permutation`:

```
  logic permutation : color_array, color_array, int, int -> prop
```

The intended meaning of `permutation(`t_1`,`t_2`,`l`,`r`)` is "the (multi-)sets of the elements of the sub-arrays $t_1[l..r]$ and $t_2[l..r]$ are the same".

To be able to write down the code, we still need to translate the `switch` statement into successive tests, and for this purpose we need to be able to decide equality of the type `color`. We can declare this ability with the following parameter:

```
  parameter eq_color  :
    c1:color -> c2:color -> {} bool { if result then c1=c2 else c1<>c2 }
```

Note that the meaning of `=` within annotations has nothing to do with a boolean function deciding equality that we could use in our programs.

We can now write the Why code for the main function:

```
let dutch_flag (t:color_array ref) (n:int) =
  { length(t) = n and forall k:int. 0 <= k < n -> is_color(acc(t,k)) }
  let b = ref 0 in
  let i = ref 0 in
  let r = ref n in
  while !i < !r do
     if (eq_color (get t !i) blue) then begin
       swap t !b !i;
       b := !b + 1;
       i := !i + 1
     end else if (eq_color (get t !i) white) then
       i := !i + 1
     else begin
       r := !r - 1;
       swap t !r !i
     end
  done
  { (exists b:int. exists r:int.
       monochrome(t,0,b,blue) and
       monochrome(t,b,r,white) and
       monochrome(t,r,n,red))
    and permutation(t,t@,0,n-1) }
```

As given above, the code cannot be proved correct, since a loop invariant is missing, and so is a termination argument. The loop invariant must maintain the current situation, which can be depicted as

0	b	i	r	n
BLUE	WHITE	...to do...	RED	

But the loop invariant must also maintain less obvious properties such as the invariance of the array length (which is obvious since we only performs `upd` operations over the array, but we need not to loose this property) and the permutation w.r.t. the initial array. The termination is trivially ensured since `r-i` decreases at each loop step and is bound by 0. Finally, the loop is annotated as follows:

```
...
init:
while !i < !r do
   { invariant 0 <= b <= i and i <= r <= n and
               monochrome(t,0,b,blue) and
               monochrome(t,b,i,white) and
               monochrome(t,r,n,red) and
               length(t) = n and
               (forall k:int. 0 <= k < n -> is_color(acc(t,k))) and
               permutation(t,t@init,0,n-1)
     variant r - i }
...
```

A label `init` has been introduced right before the loop in order to relate the current value of the array inside the loop (denoted `t`) to its value right before the loop (denoted as `t@init`) in the loop invariant.

To be able to perform the proof, we need to "define" the `permutation` predicate. The simplest way to do is to axiomatize it as an equivalence relation containing the transpositions within $[l..r]$:

```
axiom permut_refl : forall t: color_array. forall l,r:int.
  permutation(t,t,l,r)

axiom permut_swap : forall t:color_array. forall l,r,i,j:int.
  l <= i <= r -> l <= j <= r ->
  permutation(t, upd(upd(t,i,acc(t,j)), j, acc(t,i)), l, r)

axiom permut_sym : forall t1,t2:color_array. forall l,r:int.
  permutation(t1,t2,l,r) -> permutation(t2,t1,l,r)

axiom permut_trans : forall t1,t2,t3: color_array. forall l,r:int.
  permutation(t1,t2,l,r) -> permutation(t2,t3,l,r) ->
  permutation(t1,t3,l,r)
```

We have no way to state that this is the *smallest* such relation (this is not a first-order property) but this is not needed to show the correctness of the program. Note: the loop invariant part related to `is_color` seems redundant since we should be able to deduce it from the permutation property. But it would also require induction, and thus it is simpler to add this property to the loop invariant.

We can now proceed to the verification of the program with Simplify. We first produce the verification conditions in Simplify syntax:

```
why --simplify flag.why
```

Then we run Simplify on these obligations, using the tool dp (distributed with Why; see 1.5.2) that displays the results of the automatic decision procedures in a nice way:

```
dp flag_why.sx

(. = valid * = invalid # = timeout ! = failure)
flag_ax_why.sx: .......... (10/0/0)
total           :  10
valid           :  10 (100%)
invalid         :   0 (   0%)
timeout/failure :   0 (   0%)
```

As we see, the 10 verification conditions are discharged by Simplify. The source code of this example is part of the Why distribution, in `examples/misc/flag_ax.mlw`.

Variants. We describe here two variations around the Dutch flag example.

- **Definition vs. axiomatization:** Instead of axiomatizing some logical elements (such as the type `color` or the predicate `permutation` above), we can alternatively *define* them on the prover side and simply *declare* them in the Why source file. If we are using the Coq proof assistant, for instance, we could prefer defining the `color` type as an inductive data type, as follows:

  ```
  Inductive color : Set := blue : color | white : color | red : color.
  ```

  Then we can declare these external elements in the Why source file as follows:

  ```
  external type color
  external logic blue, white, red : color
  ```

  The `external` modifier indicates that nothing must be produced in the prover file for these declarations. Then we do not the `is_color` property anymore, since we are able to prove (once on the Coq side) that there is no other value in the `color` type than `blue`, `white` and `red`.

  Similarly, we could define the `permutation` predicate as a Coq inductive predicate, and declare it in the Why source file as follows:

  ```
  external logic permutation :
    color_array, color_array, int, int -> prop
  ```

- **Primitive arrays:** The Why tool actually provides a datatype of arrays, exactly in the way we have done it above i.e. an array is a reference on a purely applicative array. There is even a nice syntax for array operations: $t[e]$ stands for (`array_get` t e) and $t[e_1] := e_2$ stands for (`array_set` t e_1 e_2). This is part of the Why prelude, which is described on page 30.

1.4 Other features

This section quickly describes additional features that were not present in the example above.

1.4.1 Assertions

Assertions to be verified can be placed at any place inside code using the `assert` construct. Example:

```
begin
  x := 2 * !x;
  assert { even(x) };
  x := !x - 1
end
```

An assertion can also be placed right `after` some program expression using the postcondition construct:

```
let x = (fib 4) { even(result) } in x+x
```

Such assertions (before an expression with `assert` and after with the brackets notation) can be used to introduce *proof cuts*: first, they result in proof obligations at the corresponding program point and then they are available in the context of the following verification conditions. They are *transparent* i.e. they do not modify the information already available in the other verification conditions.

There is also an *opaque* variant of the postcondition construct, which is written with double brackets:

```
let x = (fib 4) {{ even(result) }} in x+x
```

In that case, we still need to prove the property `even(fib(4))` but, in the remaining of the program, we do not know that `x=fib(4)` anymore. Only the property `even(x)` is available.

1.4.2 Recursive functions

Recursive functions are introduced with the `let rec` construct. They are annotated with a result type (there is no type inference) and a variant to ensure termination. For instance, the famous Mac Carthy's 91 function can be written as follows:

```
let rec f91 (n:int) : int { variant max(0,101-n) } =
  if n <= 100 then
    (f91 (f91 (n + 11)))
  else
    n - 10
  { (n <= 100 and result = 91) or (n >= 101 and result = n - 10) }
```

1.4.3 Exceptions

New exceptions are declared with the `exception` declaration and may have arguments:

```
exception E
exception F of int
```

Exceptions are raised with the construct `raise`:

```
raise E
raise (F 1)
```

and caught with the construct `try ... with ... end`:

```
try f 0 with E -> 1 end
try f 0 with F x -> x end
```

It is not possible to catch an exception which is not possibly raised by the expression following the `try`.

Postconditions are extended to specify the behavior in case of uncaught exceptions:

```
raise E { false | E => true }
raise (F 1) { false | F => result = 1 }
```

In a postcondition regarding an exception, `result` refers to the value carried by the exception (if any). In postconditions, it is not possible to mention exceptions which are not possibly raised by the program.

In a function type, the exceptions possibly raised by the function must be given as part of its side-effects, using the `raises` keyword. For instance

```
parameter f : x:int -> {} int raises E { P(x,result) | E => Q(x) }
```

declares a function `f` taking an integer `x` as argument, returning an integer and possibly raising the exception `E`. In the latter case, `Q(x)` holds.

1.4.4 Unreachable code

The `absurd` construct can be used to denote an unreachable point in the code. Example:

```
{ 0 <= x <= 1 }
if x = 0 then
  ...
else if x = 1 then
  ...
else
  absurd
{ ... }
```

At such program points, the user will be asked to prove that the context is indeed absurd, which corresponds to the proof obligation `false`.

1.4.5 Non-deterministic programs

A non-deterministic program expression is a place-holder for a program expression that is not given but only specified. Such a construct is a program specification (a type, an effect and possibly some pre/postconditions) written between square brackets. For instance

```
[ {} int { even(result) } ]
```

stands for any program expression of type `int` that has the property of being even. Thus we can prove correct the following code:

```
(1 + [ {} int { even(result) } ]) { odd(result) }
```

This construct can be used to proceed by successive *refinements* or to write incomplete programs where only the meaningful parts are coded. With respect to verification, a non-deterministic program expression behaves like a call to a function parameter.

1.5 Tools

1.5.1 Graphical user interface (`gwhy`)

The graphical user interface `gwhy` is a tool to help the user in the process of calling the various automatic provers on the goals (in order to analyze the failures and to determine what is wrong in the programs, the specifications, or sometimes the provers). `gwhy` is similar to Why and is passed the same options and files. A screenshort is given Figure 1.2. On the left side appears the list of all goals (either from `goal` commands in the Why files or resulting from verification conditions generation). In front of each goal, the results for each automatic prover is displayed. Provers can be launched by clicking on their names at the top of their columns. A green bullet means a proved goal; a red bullet means an unproved goal (that may be either a "don't know" or an "invalid" answer from the prover); the scissors means a timeout from the prover (the timeout limit can be set in the bottom bar); finally, a "tools" icon means an unexpected failure in the prover call.

The top right window displays the currently selected goal. In this window, a right click on some identifier displays its location in the corresponding Why input file in the bottom right window.

1.5.2 Calling decision procedures (`dp`)

A small tool is provided to call the decision procedures (Simplify, Ergo, Yices, haRVey, Zenon and CVC Lite) on files containing several goals, with a timeout for each goal, and to display s short summary of the results. Here is an example:

```
% dp -timeout 5 swap0_why.sx peano_why.sx
(. = valid * = invalid # = timeout/failure)
swap0_why.sx: ....... (7/0/0)
peano_why.sx: ...........*...*..*.... (21/3/0)
total          :  31
valid          :  28 ( 90%)
```

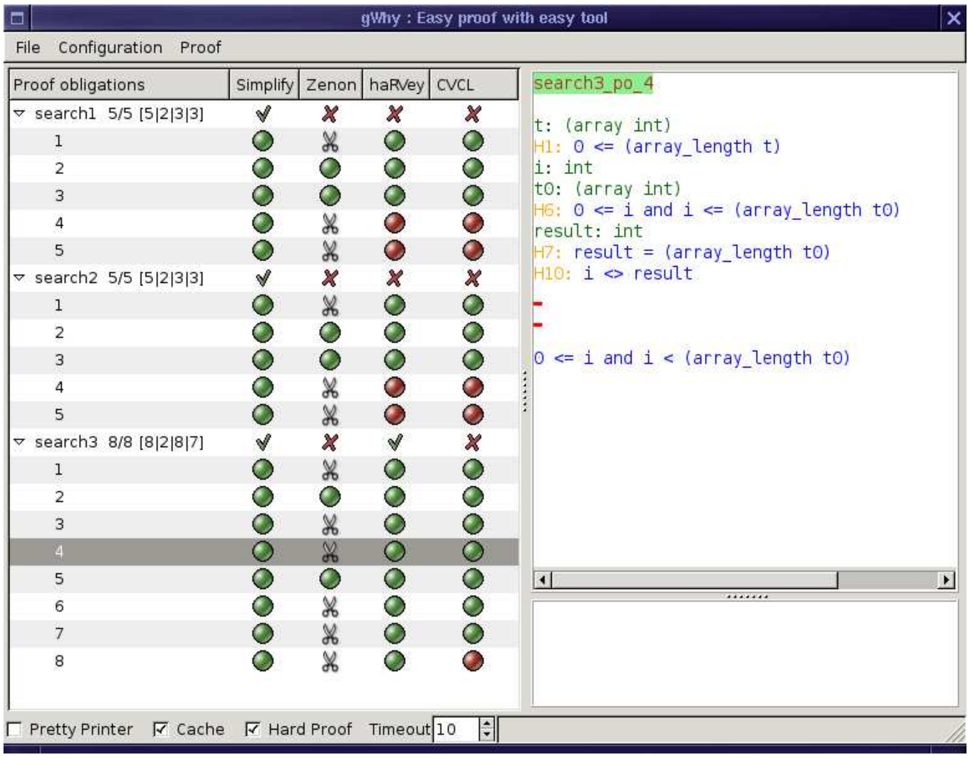

Figure 1.2: The `gwhy` graphical interface

```
invalid         :    3 ( 10%)
timeout/failure :    0 (  0%)
```

options:

`-batch` run in batch mode: no output is printed at all, but answer is given as the exit status. Notice that only the first goal is considered, so it is intended to be used only with single-goal files (see option `-?` of why)

1. 0: normal exit, prover said goal is valid
2. 1: dp error (syntax error, etc)
3. 2: prover answered 'invalid'
4. 3: prover answered 'unknown'
5. 4: prover was interrupted because of timeout
6. 5: prover failure: answer was not understood by dp

`-timeout <n>` sets the time limit to `n` seconds. Default is 10

1.5.3 Why obfuscator (`why-obfuscator`)

A tool is provided to obfuscate some Why files. It is invoked as follows:

`why-obfuscator` [`-o` *file*] [`-prefix` *p*] files...

It produces a Why file equivalent to the set of Why files given as arguments, where all identifiers are renamed and all comments removed. The output is printed on the standard output unless redirected with option `-o`. The identifiers are renamed as `x1`, `x2`, ... unless a different name prefix is set with option `-prefix`.

1.5.4 Why statistics (`why-stat`)

A tool is provided to display rough statistics on the goals contained in Why files. It is invoked as follows:

`why-stat` files...

It collects the symbols appearing in the conclusion of `goal` declarations, sums the number of goals with the same set of symbols and displays this multi-set starting with the most numerous elements. Here is a sample output:

```
8 acc,shift
2 valid
1 valid_range
```

1.5.5 Caml code generation (`why --ocaml`)

In order to run your Why code, true Caml code can be produced with option `--ocaml`:

```
why --ocaml foo.mlw
```

When this option is selected, there is no generation of proof obligations (Coq or PVS files are not produced or updated). The Caml code is written on standard output, unless redirected to some file with option `--output` (see Section 2.1).

Programs' annotations can be inserted as comments in the Caml code, with option `--ocaml-annot`. The default behavior is no annotation.

When the input file contains `parameter` declarations, the Caml code generated is a functor, with these parameters as arguments. For instance, the following input file

```
parameter x : int ref
let f (y:int) = x := !x + y
```

is translated into the following piece of Caml code:

```
module type Parameters = sig
  val x : int ref
end

module Make(P : Parameters) = struct
  open P
  let f = fun y (*:int*) -> x := !x + y
end
```

`external parameter` declarations are supposed to be realized by some external Caml code. However, it is possible to make them arguments of the functor too, with command line option `--ocaml-ext`.

1.5.6 Why to HTML converter (`why2html`)

A tool to convert Why input files to HTML is provided. Its usage is immediate:

```
why2html [-t title] files...
```

Each Why input file given on the command line, say *foo.mlw*, is translated into a HTML file *foo.mlw.html*. A title for the HTML page can be specified using command line option `-t`; otherwise the original file name is used.

1.5.7 Simplify to Why converter (`simplify2why`)

A tool to convert Simplify files to Why syntax is provided. Its usage is immediate:

```
simplify2why files...
```

Each Simplify input file given on the command line, say *foo.sx*, is translated into a Why file *foo.sx.why*. Since Simplify is untyped, all data in the Simplify input file are mapped to integers in the resulting Why file.

1.6 Installation

To compile Why, you need Objective Caml to be installed, in version at least 3.08. It can be fetched from `http://caml.inria.fr`. Then

1. Configure with `./configure`

   The directory for binaries defaults to `/usr/local/bin`; you can specify another directory with the `--bindir` option. Similarly, you can change the directory for the library with `--libdir` and for the man pages `--mandir`.

2. Compile with `make`.

3. Install with `make install`.

1.7 Examples

Many examples are delivered with Why in the subdirectory `examples/` of the distribution. They are also available on the Why web site, at `http://why.lri.fr/examples/`.

1.8 Verifying C and Java programs

C and Java programs can be verified using other tools in combination with Why. C programs can be verified using Caduceus (see `caduceus.lri.fr/`) and Java programs using Krakatoa (see `krakatoa.lri.fr`).

Chapter 2

Reference manual

2.1 Command line

Why is invoked as a batch compiler, given a list of input files:

$$\texttt{why}\ [\mathit{options}]\ \mathit{file}_1 \cdots \mathit{file}_n$$

If no file is given, then standard input is processed and output is named with the prefix `out_why` (and a suffix depending of the selected prover). The full list of command line options can be obtained with

```
why --help
```

2.2 Syntax of input files

2.2.1 Lexical conventions

Comments are opened with `(*`, closed with `*)` and can be nested.

Identifiers are made of letters, digits, the underscore character `_` and the single quote `'`, starting with a letter. Additionally, they can be qualified by a label (another identifier), using the `@` notation.

$$\begin{array}{lcl}
\langle \mathit{identifier} \rangle & ::= & \langle \mathit{letter} \rangle\ (\langle \mathit{letter} \rangle \mid \texttt{0..9} \mid \texttt{_} \mid \texttt{'})^{\star} \\
\langle \mathit{letter} \rangle & ::= & \texttt{A..Z} \mid \texttt{a..z} \\
\langle \mathit{lab_identifier} \rangle & ::= & \langle \mathit{identifier} \rangle\ [\ \texttt{@}\ [\ \langle \mathit{identifier} \rangle\ ]\ ]
\end{array}$$

Keywords are the following:

absurd	and	array	as	assert
axiom	begin	bool	do	done
else	end	exception	exists	external
false	for	forall	fun	function
goal	if	in	int	invariant
let	logic	not	of	
or	parameter	predicate	prop	raise
raises	reads	real	rec	ref
returns	then	true	try	type
unit	variant	void	while	with
writes				

2.2.2 Grammar

Logic

Syntax for terms is given Figure 2.1. Arithmetical operations have usual precedences and are left associative. The conditional construct `if` t_1 `then` t_2 `else` t_3 can also be written in prefix notation as `if_then_else(`t_1`,` t_2`,` t_3`)`.

Syntax for predicates is given Figure 2.2. Precedences are the following (from strongest to weakest): `not`, `and`, `or`, implication, and `forall`. Implication, conjunction and disjunction are right associative. Syntactic sugar: $t_1\ R_1\ t_2\ R_2\ t_3$ is equivalent to $t_1\ R_1\ t_2$ `and` $t_2\ R_2\ t_3$ for any relations R_1 and R_2. Example: `0 <= x < y` is `0 <= x and x < y`.

Syntax for logic declarations is given Figure 2.3.

$$
\begin{array}{lcl}
\langle \textit{term} \rangle & ::= & \langle \textit{constant} \rangle \\
 & | & \langle \textit{term} \rangle\ \langle \textit{arith\_op} \rangle\ \langle \textit{term} \rangle \\
 & | & \texttt{-}\ \langle \textit{term} \rangle \\
 & | & \langle \textit{lab\_identifier} \rangle \\
 & | & \langle \textit{identifier} \rangle\ \texttt{(}\ \langle \textit{term} \rangle^{+}_{,}\ \texttt{)} \\
 & | & \langle \textit{lab\_identifier} \rangle\ \texttt{[}\ \langle \textit{term} \rangle\ \texttt{]} \\
 & | & \texttt{if}\ \langle \textit{term} \rangle\ \texttt{then}\ \langle \textit{term} \rangle\ \texttt{else}\ \langle \textit{term} \rangle \\
 & | & \texttt{(}\ \langle \textit{term} \rangle\ \texttt{)} \\
 & & \\
\langle \textit{constant} \rangle & ::= & \langle \textit{integer-constant} \rangle \\
 & | & \langle \textit{real-constant} \rangle \\
 & | & \texttt{true} \\
 & | & \texttt{false} \\
 & | & \texttt{void} \\
 & & \\
\langle \textit{arith\_op} \rangle & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%}
\end{array}
$$

Figure 2.1: Syntax of terms

$$
\begin{array}{lcl}
\langle \textit{predicate} \rangle & ::= & \texttt{true} \\
& \mid & \texttt{false} \\
& \mid & \langle \textit{identifier} \rangle \\
& \mid & \langle \textit{identifier} \rangle \ \texttt{(} \ \langle \textit{term} \rangle^{+}_{\texttt{,}} \ \texttt{)} \\
& \mid & \langle \textit{term} \rangle \ \langle \textit{relation} \rangle \ \langle \textit{term} \rangle \ [\ \langle \textit{relation} \rangle \ \langle \textit{term} \rangle \ ] \\
& \mid & \langle \textit{predicate} \rangle \ \texttt{->} \ \langle \textit{predicate} \rangle \\
& \mid & \langle \textit{predicate} \rangle \ \texttt{<->} \ \langle \textit{predicate} \rangle \\
& \mid & \langle \textit{predicate} \rangle \ \texttt{or} \ \langle \textit{predicate} \rangle \\
& \mid & \langle \textit{predicate} \rangle \ \texttt{and} \ \langle \textit{predicate} \rangle \\
& \mid & \texttt{not} \ \langle \textit{predicate} \rangle \\
& \mid & \texttt{if} \ \langle \textit{term} \rangle \ \texttt{then} \ \langle \textit{predicate} \rangle \ \texttt{else} \ \langle \textit{predicate} \rangle \\
& \mid & \texttt{forall} \ \langle \textit{identifier} \rangle^{+}_{\texttt{,}} \ \texttt{:} \ \langle \textit{primitive\_type} \rangle \ [\ \langle \textit{triggers} \rangle \ ] \ \texttt{.} \ \langle \textit{predicate} \rangle \\
& \mid & \texttt{exists} \ \langle \textit{identifier} \rangle^{+}_{\texttt{,}} \ \texttt{:} \ \langle \textit{primitive\_type} \rangle \ \texttt{.} \ \langle \textit{predicate} \rangle \\
& \mid & \texttt{(} \ \langle \textit{predicate} \rangle \ \texttt{)} \\
& \mid & (\langle \textit{identifier} \rangle \mid \langle \textit{string} \rangle) \ \texttt{:} \ \langle \textit{predicate} \rangle \\
\\
\langle \textit{triggers} \rangle & ::= & \texttt{[} \ \langle \textit{trigger} \rangle^{+}_{\texttt{|}} \ \texttt{]} \\
\langle \textit{trigger} \rangle & ::= & \langle \textit{term} \rangle^{+}_{\texttt{,}} \\
\\
\langle \textit{primitive\_type} \rangle & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{real} \mid \texttt{unit} \mid \langle \textit{identifier} \rangle \mid \texttt{'} \ \langle \textit{identifier} \rangle \\
& \mid & \langle \textit{primitive\_type} \rangle \ \langle \textit{identifier} \rangle \mid \texttt{(} \ \langle \textit{primitive\_type} \rangle^{\star}_{\texttt{,}} \ \texttt{)} \ \langle \textit{identifier} \rangle \\
\\
\langle \textit{relation} \rangle & ::= & \texttt{=} \mid \texttt{<>} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=}
\end{array}
$$

Figure 2.2: Syntax of predicates

⟨*l_declaration*⟩	::=	[`external`] `type` [⟨*type_parameters*⟩] ⟨*identifier*⟩
	|	[`external`] `logic` ⟨*identifier*⟩$^+_,$ `:` ⟨*logic_type*⟩
	|	`function` ⟨*identifier*⟩ `(` ⟨*logic_binder*⟩$^\star_,$ `)` `:` ⟨*primitive_type*⟩ `=` ⟨*term*⟩
	|	`predicate` ⟨*identifier*⟩ `(` ⟨*logic_binder*⟩$^\star_,$ `)` `=` ⟨*predicate*⟩
	|	`axiom` ⟨*identifier*⟩ `:` ⟨*predicate*⟩
	|	`goal` ⟨*identifier*⟩ `:` ⟨*predicate*⟩
⟨*logic_type*⟩	::=	⟨*logic_arg_type*⟩$^\star_,$ `->` `prop`
	|	⟨*logic_arg_type*⟩$^\star_,$ `->` ⟨*primitive_type*⟩
⟨*logic_arg_type*⟩	::=	⟨*primitive_type*⟩ | ⟨*primitive_type*⟩ `array`
⟨*logic_binder*⟩	::=	⟨*identifier*⟩ `:` ⟨*primitive_type*⟩
⟨*type_parameters*⟩	::=	`'`⟨*identifier*⟩ | `(` (`'`⟨*identifier*⟩)$^+_,$ `)`

Figure 2.3: Syntax of logic declarations

Programs

Syntax of types is given Figure 2.4. Syntax of annotated programs is given Figure 2.5. Syntax of input files is given Figure 2.6.

2.3 Semantics

2.3.1 Logic

The abstract syntax for types (τ), terms (t) and predicates (P) is given by the following grammars:

$$\begin{array}{rcl}
\tau & ::= & \alpha \mid (\tau, \ldots, \tau)\ s \\
t & ::= & x \mid f(t, \ldots, t) \\
P & ::= & p(t, \ldots, t) \\
& \mid & \top \mid \bot \mid P \wedge P \mid P \vee P \mid \neg P \mid P \Rightarrow P \\
& \mid & \forall x : \tau.\, P \mid \exists x : \tau.\, P
\end{array}$$

A theory Σ is a finite list of declarations δ where

$$\begin{array}{rcl}
\delta & ::= & \texttt{type}\ \vec{\alpha}\ s \mid x : \tau \mid \texttt{logic}\ f : \forall \vec{\alpha}.\, \tau, \ldots, \tau \rightarrow \tau \\
& \mid & \texttt{logic}\ p : \forall \vec{\alpha}.\, \tau, \ldots, \tau \rightarrow \texttt{prop} \mid \texttt{axiom}\ \forall \vec{\alpha}.\, P \mid \texttt{goal}\ \forall \vec{\alpha}.\, P
\end{array}$$

Typing

Well-formed types ($\Sigma \vdash \tau$ wf):

$$\frac{}{\Sigma \vdash \alpha\ \mathsf{wf}}(\mathsf{Ty}_1) \qquad \frac{\texttt{type}\ (\alpha_1, \ldots, \alpha_n)\ s \in \Sigma \quad \forall i,\ \Sigma \vdash \tau_i\ \mathsf{wf}}{\Sigma \vdash (\tau_1, \ldots, \tau_n)\ s\ \mathsf{wf}}(\mathsf{Ty}_2)$$

⟨*simple_value_type*⟩	::=	⟨*primitive_type*⟩
	|	⟨*primitive_type*⟩ `ref`
	|	⟨*primitive_type*⟩ `array`
	|	`(` ⟨*value_type*⟩ `)`
⟨*value_type*⟩	::=	⟨*simple_value_type*⟩
	|	⟨*simple_value_type*⟩ `->` ⟨*computation_type*⟩
	|	⟨*identifier*⟩ `:` ⟨*simple_value_type*⟩ `->` ⟨*computation_type*⟩
⟨*computation_type*⟩	::=	{ [⟨*precondition*⟩] }
		[`returns` ⟨*identifier*⟩ `:`] ⟨*value_type*⟩ ⟨*effects*⟩
		{ [⟨*postcondition*⟩] }
	|	⟨*value_type*⟩
⟨*effects*⟩	::=	[`reads` ⟨*identifier*⟩$^{\star}_{,}$] [`writes` ⟨*identifier*⟩$^{\star}_{,}$] [`raises` ⟨*identifier*⟩$^{\star}_{,}$]
⟨*precondition*⟩	::=	⟨*assertion*⟩
⟨*postcondition*⟩	::=	⟨*assertion*⟩ ⟨*exn_condition*⟩*
⟨*exn_condition*⟩	::=	`|` ⟨*identifier*⟩ `=>` ⟨*assertion*⟩
⟨*assertion*⟩	::=	⟨*predicate*⟩ [`as` ⟨*identifier*⟩]

Figure 2.4: Syntax of types

```
⟨prog⟩        ::=  ⟨constant⟩
                |  ⟨identifier⟩
                |  ! ⟨identifier⟩
                |  ⟨identifier⟩ := ⟨prog⟩
                |  ⟨identifier⟩ [ ⟨prog⟩ ]
                |  ⟨identifier⟩ [ ⟨prog⟩ ] := ⟨prog⟩
                |  ⟨prog⟩ ⟨infix⟩ ⟨prog⟩
                |  ⟨prefix⟩ ⟨prog⟩
                |  let ⟨identifier⟩ = ⟨prog⟩ in ⟨prog⟩
                |  let ⟨identifier⟩ = ref ⟨prog⟩ in ⟨prog⟩
                |  if ⟨prog⟩ then ⟨prog⟩ [ else ⟨prog⟩ ]
                |  while ⟨prog⟩ do [ ⟨loop_annot⟩ ] ⟨prog⟩ done
                |  ⟨prog⟩ ; ⟨prog⟩
                |  ⟨identifier⟩ : ⟨prog⟩
                |  assert ({ ⟨assertion⟩ })+ ; ⟨prog⟩
                |  ⟨prog⟩ { ⟨postcondition⟩ }
                |  ⟨prog⟩ {{ ⟨postcondition⟩ }}
                |  fun ⟨binders⟩ -> ⟨prog⟩
                |  let ⟨identifier⟩ ⟨binders⟩ = ⟨prog⟩ in⟨prog⟩
                |  let rec ⟨recfun⟩ [ in ⟨prog⟩ ]
                |  ⟨prog⟩ ⟨prog⟩
                |  raise ⟨identifier⟩ [ : ⟨value_type⟩ ]
                |  raise ( ⟨identifier⟩ ⟨prog⟩ ) [ : ⟨value_type⟩ ]
                |  try ⟨prog⟩ with ⟨handler⟩+| end
                |  absurd [ : ⟨value_type⟩ ]
                |  ( ⟨prog⟩ )
                |  begin ⟨prog⟩ end

⟨infix⟩       ::=  + | - | * | / | % | = | <> | < | <= | > | >= | || | &&
⟨prefix⟩      ::=  - | not

⟨binders⟩     ::=  ( ⟨identifier⟩+, : ⟨value_type⟩ )+

⟨recfun⟩      ::=  ⟨identifier⟩ ⟨binders⟩ : value_type
                   { variant ⟨wf_arg⟩ } = ⟨prog⟩

⟨loop_annot⟩  ::=  { [ invariant ⟨assertion⟩ ] [ variant ⟨wf_arg⟩ ] }

⟨wf_arg⟩      ::=  ⟨term⟩ [ for ⟨identifier⟩ ]

⟨handler⟩     ::=  ⟨identifier⟩ -> ⟨prog⟩
                |  ⟨identifier⟩ ⟨identifier⟩ -> ⟨prog⟩
```

Figure 2.5: Syntax of annotated programs

⟨*file*⟩	::=	⟨*declaration*⟩*
⟨*declaration*⟩	::=	`let` ⟨*identifier*⟩ [⟨*binders*⟩] `=` ⟨*prog*⟩
	|	`let rec` ⟨*recfun*⟩
	|	[`external`] `parameter` ⟨*identifier*⟩$^+_,$ `:` ⟨*value_type*⟩
	|	`exception` ⟨*identifier*⟩ [`of` ⟨*primitive_type*⟩]
	|	⟨*l_declaration*⟩

Figure 2.6: Syntax of input files

Well-typed terms ($\Sigma \vdash t : \tau$):

$$\frac{x : \tau \in \Sigma}{\Sigma \vdash x : \tau}(\mathsf{T}_1) \qquad \frac{\begin{array}{c}\texttt{logic}\ f : \forall\vec{\alpha}.\,\tau_1,\ldots,\tau_n \to \tau \in \Sigma \\ \mathsf{Subst}(\sigma,\Sigma) \quad \forall i,\, \Sigma \vdash t_i : \sigma(\tau_i)\end{array}}{\Sigma \vdash f(t_1,\ldots,t_n) : \sigma(\tau)}(\mathsf{T}_2)$$

where σ is a mapping from type variables to types, naturally extended to types with $\sigma((\tau_1,\ldots,\tau_n)\ s) = (\sigma(\tau_1),\ldots,\sigma(\tau_n))\ s$. We write $\mathsf{Subst}(\sigma,\Sigma)$ whenever $\Sigma \vdash \sigma(\alpha)\ \mathsf{wf}$ holds for any type variable α.

Well-typed predicates ($\Sigma \vdash P\ \mathsf{wf}$):

$$\frac{\texttt{logic}\ p : \forall\vec{\alpha}.\,\tau_1,\ldots,\tau_n \to \texttt{prop} \in \Sigma \quad \mathsf{Subst}(\sigma,\Sigma) \quad \forall i,\, \Sigma \vdash t_i : \sigma(\tau_i)}{\Sigma \vdash p(t_1,\ldots,t_n)\ \mathsf{wf}}(\mathsf{P}_1)$$

$$\frac{}{\Sigma \vdash \top\ \mathsf{wf}}(\mathsf{P}_2) \qquad \frac{}{\Sigma \vdash \bot\ \mathsf{wf}}(\mathsf{P}_3) \qquad \frac{\Sigma \vdash P_1\ \mathsf{wf} \quad \Sigma \vdash P_2\ \mathsf{wf}}{\Sigma \vdash P_1 \wedge P_2\ \mathsf{wf}}(\mathsf{P}_4)$$

$$\frac{\Sigma \vdash P_1\ \mathsf{wf} \quad \Sigma \vdash P_2\ \mathsf{wf}}{\Sigma \vdash P_1 \vee P_2\ \mathsf{wf}}(\mathsf{P}_5) \qquad \frac{\Sigma \vdash P\ \mathsf{wf}}{\Sigma \vdash \neg P\ \mathsf{wf}}(\mathsf{P}_6)$$

$$\frac{\Sigma \vdash P_1\ \mathsf{wf} \quad \Sigma \vdash P_2\ \mathsf{wf}}{\Sigma \vdash P_1 \Rightarrow P_2\ \mathsf{wf}}(\mathsf{P}_7)$$

$$\frac{\Sigma, x : \tau \vdash P\ \mathsf{wf}}{\Sigma \vdash \forall x : \tau.\, P\ \mathsf{wf}}(\mathsf{P}_8) \qquad \frac{\Sigma, x : \tau \vdash P\ \mathsf{wf}}{\Sigma \vdash \exists x : \tau.\, P\ \mathsf{wf}}(\mathsf{P}_8)$$

Well-formed theories ($\vdash \Sigma\ \mathsf{wf}$):

$$\frac{}{\vdash \emptyset\ \mathsf{wf}}(\mathsf{Th}_1) \qquad \frac{\texttt{type}\ s \notin \Sigma}{\vdash \Sigma, \texttt{type}\ (\alpha_1,\ldots,\alpha_n)\ s\ \mathsf{wf}}(\mathsf{Th}_2) \qquad \frac{x \notin \Sigma \quad \Sigma \vdash \tau\ \mathsf{wf}}{\vdash \Sigma, x : \tau\ \mathsf{wf}}(\mathsf{Th}_3)$$

$$\frac{\texttt{logic}\ f \notin \Sigma \quad \forall i,\, \Sigma \vdash \tau_i\ \mathsf{wf}}{\vdash \Sigma, \texttt{logic}\ f : \forall\vec{\alpha}.\,\tau_1,\ldots,\tau_n \to \tau_{n+1}\ \mathsf{wf}}(\mathsf{Th}_4)$$

$$\frac{\texttt{logic}\ p \notin \Sigma \quad \forall i,\, \Sigma \vdash \tau_i\ \mathsf{wf}}{\vdash \Sigma, \texttt{logic}\ p : \forall\vec{\alpha}.\,\tau_1,\ldots,\tau_n \to \texttt{prop}\ \mathsf{wf}}(\mathsf{Th}_5)$$

$$\frac{\Sigma \vdash P\ \mathsf{wf}}{\vdash \Sigma, \texttt{axiom}\ \forall\vec{\alpha}.\,P\ \mathsf{wf}}(\mathsf{Th}_6) \qquad \frac{\Sigma \vdash P\ \mathsf{wf}}{\vdash \Sigma, \texttt{goal}\ \forall\vec{\alpha}.\,P\ \mathsf{wf}}(\mathsf{Th}_6)$$

Well-formed definitions of functions and predicates:

$$\frac{\Sigma \vdash \tau_i \ \mathsf{wf} \quad \Sigma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t : \tau}{\Sigma \vdash \texttt{function}\ f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = t \ \mathsf{wf}}(\mathsf{Th}_7)$$

$$\frac{\Sigma \vdash \tau_i \ \mathsf{wf} \quad \Sigma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash P \ \mathsf{wf}}{\Sigma \vdash \texttt{predicate}\ p(x_1 : \tau_1, \ldots, x_n : \tau_n) = P \ \mathsf{wf}}(\mathsf{Th}_8)$$

Then, as far as typing and validity in concerned, the function f (resp. the predicate p) is added to Σ as $\texttt{logic}\ f : \tau_1, \ldots, \tau_n \rightarrow \tau$ (resp. $\texttt{logic}\ p : \tau_1, \ldots, \tau_n \rightarrow \texttt{prop}$).

Validity

Validity is defined as a set of natural deduction rules ($\Sigma \models P$). For the sake of clarity, we write Σ, P for $\Sigma, \texttt{axiom}\ P$ in the following. A substitution σ over from type variables to types in extended to terms and predicates in the obvious way. We write $P[t/x]$ for the substitution of all the occurrences of a free variable x in P by a term t.

$$\frac{\texttt{axiom}\ \forall \vec{\alpha}.\, P \in \Sigma \quad \mathsf{Subst}(\sigma, \Sigma)}{\Sigma \models \sigma(P)}(\mathsf{Ax}) \qquad \frac{\Sigma \models Q \quad \Sigma, Q \models P}{\Sigma \models P}(\mathsf{Cut})$$

$$\frac{}{\Sigma \models \top}(\mathsf{True}) \qquad \frac{\Sigma \models \bot \quad \Sigma \vdash P \ \mathsf{wf}}{\Sigma \models P}(\mathsf{False}) \qquad \frac{\Sigma \vdash P \ \mathsf{wf}}{\Sigma \models P \vee \neg P}(\mathsf{EM})$$

$$\frac{\Sigma \models P \quad \Sigma \models Q}{\Sigma \models P \wedge Q}(\mathsf{And}_1) \qquad \frac{\Sigma \models P \wedge Q}{\Sigma \models P}(\mathsf{And}_2) \qquad \frac{\Sigma \models P \wedge Q}{\Sigma \models Q}(\mathsf{And}_3)$$

$$\frac{\Sigma \models P \quad \Sigma \vdash Q \ \mathsf{wf}}{\Sigma \models P \vee Q}(\mathsf{Or}_1) \qquad \frac{\Sigma \models Q \quad \Sigma \vdash P \ \mathsf{wf}}{\Sigma \models P \vee Q}(\mathsf{Or}_2)$$

$$\frac{\Sigma \models P \vee Q \quad \Sigma, P \models R \quad \Sigma, Q \models R}{\Sigma \models R}(\mathsf{Or}_3)$$

$$\frac{\Sigma \vdash P \ \mathsf{wf} \quad \Sigma, P \models \bot}{\Sigma \models \neg P}(\mathsf{Not}_1) \qquad \frac{\Sigma \models P \quad \Sigma \models \neg P}{\Sigma \models \bot}(\mathsf{Not}_2)$$

$$\frac{\Sigma \vdash P \ \mathsf{wf} \quad \Sigma, P \models Q}{\Sigma \models P \Rightarrow Q}(\mathsf{Imp}_1) \qquad \frac{\Sigma \models P \Rightarrow Q \quad \Sigma \models P}{\Sigma \models Q}(\mathsf{Imp}_2)$$

$$\frac{x \notin \Sigma \quad \Sigma \vdash \tau \ \mathsf{wf} \quad \Sigma, x : \tau \models P}{\Sigma \models \forall x : \tau.\, P}(\mathsf{Forall}_1)$$

$$\frac{\Sigma \models \forall x : \tau.\, P \quad \Sigma \vdash t : \tau}{\Sigma \models P[t/x]}(\mathsf{Forall}_2)$$

$$\frac{\Sigma \vdash t : \tau \quad x \notin \Sigma \quad \Sigma \models P[t/x]}{\Sigma \models \exists x : \tau.\, P}(\mathsf{Exists}_1)$$

$$\frac{\Sigma \models \exists x : \tau.\, P \quad x \notin \Sigma \quad \Sigma, x : \tau, P \models Q}{\Sigma \models Q}(\mathsf{Exists}_2)$$

Deduction rules for equality:

$$\frac{\Sigma \vdash t : \tau}{\Sigma \models t = t}(\mathsf{Eq}_1) \qquad \frac{\Sigma \models x = y \quad \Sigma, z : \tau \vdash P \ \mathsf{wf} \quad \Sigma \models P[x/z]}{\Sigma \models P[y/z]}(\mathsf{Eq}_2)$$

Arithmetic: An arithmetic proposition P is a proposition built from variables of type *int*, integers constants, the functions symbols *add_int* and *sub_int*, the predicates *lt_int, le_int, gt_int, ge_int* and the equality. If $x_1, \dots, x_n$ are the free variables of P, we write $x_1, \dots, x_n \models_A P$ whenever P is valid in Presburger arithmetic (which is decidable). Then we have the following deduction rule for arithmetic:

$$\frac{x_1, \dots, x_n \models_A P}{\Sigma \models \forall x_1 : int. \dots \forall x_n : int. P}(\mathsf{Arith})$$

2.3.2 Programs

Program types (θ) and specifications (κ) are classified as follows:

$$\begin{array}{lcl} \theta & ::= & \tau \mid \tau\ \mathsf{ref} \mid (x : \theta) \rightarrow \kappa \\ \kappa & ::= & \{p\}\ \theta\ \epsilon\ \{q\} \\ q & ::= & p; E \Rightarrow p; \dots ; E \Rightarrow p \\ \epsilon & ::= & \mathsf{reads}\ x, \dots, x\ \mathsf{writes}\ x, \dots, x\ \mathsf{raises}\ E, \dots, E \end{array}$$

A value of type θ is either an immutable variable of a pure type (τ), a reference containing a value of a pure type (τ ref) or a function of type $(x : \theta) \rightarrow \{p\}\ \theta'\ \epsilon\ \{q\}$ mapping the formal parameter x to the specification of its body, that is a precondition p, the type θ' for the returned value, an effect ϵ and a postcondition q. An effect is made of tree lists of variables: the references possibly accessed (`reads`), the references possibly modified (`writes`) and the exceptions possibly raised (`raises`). A postcondition q is made of several parts: one for the normal termination and one for each possibly raised exception (E stands for an exception name).

When a function specification $\{p\}\ \theta\ \epsilon\ \{q\}$ has no precondition and no postcondition (both being $\top$) and no effect (ϵ is made of three empty lists) it can be shortened to θ. In particular, $(x_1 : \theta_1) \rightarrow \cdots \rightarrow (x_n : \theta_n) \rightarrow \kappa$ denotes the type of a function with n arguments that has no effect as long as it not applied to n arguments. Note that functions can be partially applied.

The syntax for program expressions is given in Figure 2.7. In particular, programs contain *pure terms* (t) made of constants, variables, dereferences (written `!`x) and application of function symbols from the logic to pure terms. $\mathsf{loop}\ e\ \{\mathsf{invariant}\ p\ \mathsf{variant}\ t\}$ is an infinite loop of body e, invariant p and which termination is ensured by the variant t.

Syntactic sugar. The traditional sequence construct is only syntactic sugar for a `let-in` binder where the variable does not occur in e_2:

$$e_1\,\texttt{;}\ e_2 \equiv \mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2$$

The assignment construct is syntactic sugar for the application of the function `ref_set` from Why's prelude (see 2.4):

$$x \texttt{ := } e \equiv \texttt{ref\_set}\ x\ e$$

We also simplify the `raise` construct whenever both the exception contents and the whole `raise` expression have type `unit`:

$$\mathsf{raise}\ E \equiv \mathsf{raise}\ (E\ \mathsf{void}) : \mathsf{unit}$$

$$
\begin{array}{rcl}
t & ::= & x \mid {!x} \mid f(t, \ldots, t) \\
e & ::= & t \\
 & \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e \\
 & \mid & \mathsf{let}\ x = \mathsf{ref}\ e\ \mathsf{in}\ e \\
 & \mid & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
 & \mid & \mathsf{loop}\ e\ \{\mathsf{invariant}\ p\ \mathsf{variant}\ t\} \\
 & \mid & L{:}e \\
 & \mid & \mathsf{raise}\ (E\ e) : \theta \\
 & \mid & \mathsf{try}\ e\ \mathsf{with}\ E\ x \rightarrow e\ \mathsf{end} \\
 & \mid & \mathsf{assert}\ \{p\};\ e \\
 & \mid & e\ \{q\} \\
 & \mid & e\ \{\{q\}\} \\
 & \mid & \mathsf{fun}\ (x : \theta) \rightarrow \{p\}\ e \\
 & \mid & \mathsf{rec}\ x\ (x : \theta) \ldots (x : \theta) : \theta\ \{\mathsf{variant}\ t\} = \{p\}\ e \\
 & \mid & e\ e
\end{array}
$$

Figure 2.7: Abstract syntax of program expressions

The traditional `while` loop is also syntactic sugar for a combination of an infinite loop and the use of an exception *Exit* to exit the loop:

$$
\begin{array}{l}
\mathsf{while}\ e_1\ \mathsf{do}\ e_2\ \{\mathsf{invariant}\ p\ \mathsf{variant}\ t\}\ \equiv \\
\quad \mathsf{try} \\
\quad\quad \mathsf{loop}\ \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ \mathsf{raise}\ \mathit{Exit} \\
\quad\quad \{\mathsf{invariant}\ p\ \mathsf{variant}\ t\} \\
\quad \mathsf{with}\ \mathit{Exit}\ \_\ \texttt{->}\ \mathsf{void}\ \mathsf{end}
\end{array}
$$

Typing

The typing of program expressions is straightforward, with polymorphism but no type inference.

Operational semantics

The language is strict (all arguments always evaluated before the function call), with a left-to-right evaluation order.

Weakest preconditions

Programs correctness is defined using a calculus of weakest preconditions. We note $wp(e, q; r)$ the weakest precondition for a program expression e and a postcondition $q; r$ where q is the property to hold when terminating normally and $r = E_1 \Rightarrow q_1; \ldots; E_n \Rightarrow q_n$ is the set of properties to hold for each possibly uncaught exception. Expressing the correctness of a program e is simply a matter of computing $wp(e, \mathsf{True})$.

The rules for the basic constructs are the following:

$$
\begin{aligned}
wp(t,q;r) &= q[result \leftarrow t]\\
wp(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2,q;r) &= wp(e_1, wp(e_2,q;r)[x \leftarrow result];r)\\
wp(\mathsf{let}\ x = \mathsf{ref}\ e_1\ \mathsf{in}\ e_2,q;r) &= wp(e_1, wp(e_2,q;r)[x \leftarrow result];r)\\
wp(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3,q;r) &= wp(e_1, \mathsf{if}\ result\ \mathsf{then}\ wp(e_2,q;r)\ \mathsf{else}\ wp(e_3,q;r);r)\\
wp(L\!:\!e,q;r) &= wp(e,q;r)[\mathtt{at}(x,L) \leftarrow x]
\end{aligned}
$$

On the case of an exception free sequence, we retrieve the usual rule:

$$wp(e_1\,;\ e_2, q) = wp(e_1, wp(e_2, q))$$

The cases of exceptions and annotations are also straightforward:

$$
\begin{aligned}
wp(\mathsf{raise}\ (E\ e) : \theta, q; r) &= wp(e, r(E); r)\\
wp(\mathsf{try}\ e_1\ \mathsf{with}\ E\ x \rightarrow e_2\ \mathsf{end}, q; r) &= wp(e_1, q; E \Rightarrow wp(e_2,q;r)[x \leftarrow result];r)\\
wp(\mathsf{assert}\ \{p\};\ e,q;r) &= p \wedge wp(e,q;r)\\
wp(e\ \{q';r'\},q;r) &= wp(e, q' \wedge q; r' \wedge r)\\
wp(e\ \{\{q';r'\}\},q;r) &= wp(e,q';r') \wedge \forall\omega.\forall result.q' \Rightarrow q \wedge r' \Rightarrow r
\end{aligned}
$$

The last implication $r' \Rightarrow r$ is actually an abuse for the conjunction of all implications for each postcondition part. The case of an infinite loop is more subtle:

$$wp(\mathsf{loop}\ e\ \{\mathsf{invariant}\ p\ \mathsf{variant}\ t\}, q; r) \ = p\ \wedge\ \forall\omega.\ p \Rightarrow wp(L\!:\!e, p \wedge t < \mathtt{at}(t,L); r)$$

where ω stands for the set of references possibly modified by the loop body (the `writes` part of e's effect). Here the weakest precondition expresses that the invariant must hold initially and that for each turn in the loop (represented by ω), either p is preserved by e and e decreases the value of t (to ensure termination), or e raises an exception and thus must establish r directly.

By combining this rule and the rule for the conditional, we can retrieve the rule for the usual while loop:

$$
\begin{aligned}
& wp(\mathsf{while}\ e_1\ \mathsf{do}\ e_2\ \{\mathsf{invariant}\ p\ \mathsf{variant}\ t\}, q; r)\\
=\ & p\ \wedge\ \forall\omega.\ p \Rightarrow\\
& wp(L\!:\!\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else\ raise}\ E, p \wedge t < \mathtt{at}(t,L), E \Rightarrow q; r)\\
=\ & p\ \wedge\ \forall\omega.\ p \Rightarrow\\
& wp(e_1, \mathsf{if}\ result\ \mathsf{then}\ wp(e_2, p \wedge t < \mathtt{at}(t,L))\ \mathsf{else}\ q, r)[\mathtt{at}(x,L) \leftarrow x]
\end{aligned}
$$

Finally, we give the rules for functions and function calls. Since a function cannot be mentioned within the postcondition, the weakest preconditions for function constructs `fun` and `rec` are only expressing the correctness of the function body:

$$wp(\mathsf{fun}\ (x:\theta) \rightarrow \{p\}\ e, q; r) \ = q\ \wedge\ \forall x.\forall\rho.p \Rightarrow wp(e, \mathsf{True})$$

$$
\begin{aligned}
& wp(\mathsf{rec}\ f\ (x_1:\theta_1)\ldots(x_n:\theta_n) : \theta\ \{\mathsf{variant}\ t\} = \{p\}\ e, q; r)\\
& = q\ \wedge\ \forall x_1.\ldots\forall x_n.\forall\rho.p \Rightarrow wp(L\!:\!e, \mathsf{True})
\end{aligned}
$$

where ρ stands for the set of references possibly accessed by the loop body (the `reads` part of e's effect). In the case of a recursive function, $wp(L\!:\!e, \mathsf{True})$ must be computed within an environment where f is assumed to have type $(x_1 : \theta_1) \rightarrow \cdots \rightarrow (x_n : \theta_n) \rightarrow \{p \wedge t < \mathtt{at}(t, L)\}\, \theta\, \epsilon\, \{q\}$ i.e. where the decreasing of the variant t has been added to the precondition of f.

The case of a function call $e1\ e2$ can be simplified to the case of an application $x_1\ x_2$ of one variable to another, using the following transformation if needed:

$$e_1\ e_2 \equiv \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ x_1\ x_2$$

Then assuming that x_1 has type $(x : \theta) \rightarrow \{p'\}\, \theta'\, \epsilon\, \{q'\}$, we define

$$wp(x_1\ x_2, q) = p'[x \leftarrow x_2]\ \wedge\ \forall \omega. \forall result.(q'[x \leftarrow x_2] \Rightarrow q)[\mathtt{old}(t) \leftarrow t]$$

that is (1) the precondition of the function must hold and (2) its postcondition must imply the expected property q whatever the values of the modified references and of the result are. Note that q and q' may contain exceptional parts and thus the implication is again an abuse for the conjunction of all implications for each postcondition part.

In the case of an assignment x := e the rules above combined with the specification of `ref_set` give (assuming that *result* does not occur in q):

$$wp(x \mathrel{\mathtt{:=}} e, q; r) = wp(e, x = result \Rightarrow q; r)$$

And in the case of the assigment of a side-effect free expression t, it simplifies to

$$wp(x \mathrel{\mathtt{:=}} t, q) = (x = t \Rightarrow q)$$

which is equivalent to the usual rule $wp(x \mathrel{\mathtt{:=}} t, q) = q[x \leftarrow t]$.

2.4 Prelude

Apart from the type names `unit`, `bool`, `int`, `real`, and the type constructor `ref`, nothing is built-in in the Why tool. But two prelude files are read before considering the files given on the command line. The prelude files are

- `/usr/local/lib/why/prelude.why`, and
- `/usr/local/lib/why/arrays.why`

(unless your library path is set differently). The interpretation of this prelude is prover-dependent. It is possible not to load the prelude using the command-line option `--no-prelude`, or to load only the first prelude file using the command-line option `--no-arrays`. The remaining of this section describes the contents of the prelude files.

2.4.1 prelude.why

```
(* Built-in types: unit, bool, int, real *)
(* Built-in type constructor: 'a ref *)


(* Integers *)
```

```
logic add_int : int,int -> int (* + *)
logic sub_int : int,int -> int (* - *)
logic mul_int : int,int -> int (* * *)
logic div_int : int,int -> int (* / in logic only *)
logic mod_int : int,int -> int (* % in logic only *)
logic neg_int : int -> int     (* prefix - *)

(* / and % in programs: *)
parameter div_int_ : x:int -> y:int -> { y<>0 } int { result = x / y }
parameter mod_int_ : x:int -> y:int -> { y<>0 } int { result = x % y }

logic lt_int : int,int -> prop  (* <  *)
logic le_int : int,int -> prop  (* <= *)
logic gt_int : int,int -> prop  (* >  *)
logic ge_int : int,int -> prop  (* >= *)
logic eq_int : int,int -> prop  (* =  *)
logic neq_int : int,int -> prop (* <> *)

(* < <= > >= = and <> in programs: *)
parameter lt_int_ : x:int -> y:int -> {} bool { if result then x<y else x>=y }
parameter le_int_ : x:int -> y:int -> {} bool { if result then x<=y else x>y }
parameter gt_int_ : x:int -> y:int -> {} bool { if result then x>y else x<=y }
parameter ge_int_ : x:int -> y:int -> {} bool { if result then x>=y else x<y }
parameter eq_int_ : x:int -> y:int -> {} bool { if result then x=y else x<>y }
parameter neq_int_ : x:int -> y:int -> {} bool { if result then x<>y else x=y }

(* boolean versions of the comparisons and the corresponding axioms *)
logic lt_int_bool : int,int -> bool
logic le_int_bool : int,int -> bool
logic gt_int_bool : int,int -> bool
logic ge_int_bool : int,int -> bool
logic eq_int_bool : int,int -> bool
logic neq_int_bool : int,int -> bool

axiom lt_int_bool_axiom :
  forall x:int. forall y:int. lt_int_bool(x,y)=true <-> x<y
axiom le_int_bool_axiom :
  forall x:int. forall y:int. le_int_bool(x,y)=true <-> x<=y
axiom gt_int_bool_axiom :
  forall x:int. forall y:int. gt_int_bool(x,y)=true <-> x>y
axiom ge_int_bool_axiom :
  forall x:int. forall y:int. ge_int_bool(x,y)=true <-> x>=y
axiom eq_int_bool_axiom :
  forall x:int. forall y:int. eq_int_bool(x,y)=true <-> x=y
axiom neq_int_bool_axiom :
  forall x:int. forall y:int. neq_int_bool(x,y)=true <-> x<>y

(* Reals *)
```

```
logic add_real : real,real -> real (* + *)
logic sub_real : real,real -> real (* - *)
logic mul_real : real,real -> real (* * *)
logic div_real : real,real -> real (* / *)
logic pow_real : real,real -> real
logic neg_real : real -> real      (* prefix - *)
logic abs_real : real -> real

(* / in programs: *)
parameter div_real_ : x:real -> y:real -> { y<>0.0 } real { result = x / y }

logic sqrt_real : real -> real
parameter sqrt_real_ : x:real -> { x >= 0.0 } real { result = sqrt_real(x) }

logic real_of_int : int -> real
logic int_of_real : real -> int

(* real comparisons in logic... *)
logic lt_real : real,real -> prop
logic le_real : real,real -> prop
logic gt_real : real,real -> prop
logic ge_real : real,real -> prop
logic eq_real : real,real -> prop
logic neq_real : real,real -> prop

(* ... and in programs: *)
parameter lt_real_ : x:real -> y:real -> {} bool {if result then x<y else x>=y}
parameter le_real_ : x:real -> y:real -> {} bool {if result then x<=y else x>y}
parameter gt_real_ : x:real -> y:real -> {} bool {if result then x>y else x<=y}
parameter ge_real_ : x:real -> y:real -> {} bool {if result then x>=y else x<y}
parameter eq_real_ : x:real -> y:real -> {} bool {if result then x=y else x<>y}
parameter neq_real_: x:real -> y:real -> {} bool {if result then x<>y else x=y}

(* Equality over types bool and unit *)

logic eq_bool : bool,bool -> prop
logic neq_bool : bool,bool -> prop

parameter eq_bool_ : x:bool -> y:bool -> {} bool {if result then x=y else x<>y}
parameter neq_bool_: x:bool -> y:bool -> {} bool {if result then x<>y else x=y}

logic eq_unit : unit,unit -> prop
logic neq_unit : unit,unit -> prop

parameter eq_unit_ : x:unit -> y:unit -> {} bool {if result then x=y else x<>y}
parameter neq_unit_: x:unit -> y:unit -> {} bool {if result then x<>y else x=y}

(* if-then-else function symbol *)
```

```
logic ite : bool,'a,'a -> 'a

axiom ite_true : forall x, y : 'a. ite(true, x, y) = x
axiom ite_false : forall x, y : 'a. ite(false, x, y) = y

(* References. := is syntactic sugar for ref_set: *)

parameter ref_set : x:'a ref -> v:'a -> {} unit writes x { x = v }

(* Non-deterministic values *)

parameter any_bool : unit -> {} bool { true }
parameter any_int : unit -> {} int { true }
parameter any_real : unit -> {} real { true }
```

2.4.2 arrays.why

```
(* Arrays.
   Type 'a array is a built-in shortcut for 'a farray ref, where 'a farray
   is the following abstract type for purely applicative arrays *)

type 'a farray

logic array_length : 'a farray -> int

parameter array_length_ :
  t:'a array -> {} int reads t { result=array_length(t) }

logic access : 'a farray, int -> 'a
logic update : 'a farray, int, 'a -> 'a farray

axiom access_update :
  forall t : 'a farray. forall i : int. forall v : 'a.
  access(update(t,i,v), i) = v

axiom access_update_neq :
  forall t : 'a farray. forall i : int. forall j : int. forall v : 'a.
  i <> j -> access(update(t,i,v), j) = access(t,j)

(* t[e] is syntactic sugar for (array_get t e) *)
parameter array_get :
  t:'a array -> i:int ->
    { 0 <= i < array_length(t) } 'a reads t { result = access(t,i) }

(* t [e1] := e2 is syntactic sugar for (array_set t e1 e2) *)
parameter array_set :
  t:'a array -> i:int -> v:'a ->
    { 0 <= i < array_length(t) } unit writes t { t = update(t@,i,v) }
```

```
(* Sorted arrays *)

predicate sorted_array(t:int farray, i:int, j:int) =
  forall k:int. i <= k < j -> t[k] <= t[k+1]

logic array_le : int farray, int, int, int -> prop
logic array_ge : int farray, int, int, int -> prop

(* Arrays permutations *)

predicate exchange(t1:int farray, t2:int farray, i:int, j:int) =
  array_length(t1) = array_length(t2) and
  t1[i] = t2[j] and t2[i] = t1[j] and
  forall k:int. (k <> i and k <> j) -> t1[k] = t2[k]

logic permut : int farray, int farray -> prop

axiom permut_refl :
  forall t:int farray. permut(t,t)

axiom permut_sym :
  forall t1:int farray. forall t2:int farray.
    permut(t1,t2) -> permut(t2,t1)

axiom permut_trans :
  forall t1:int farray. forall t2:int farray. forall t3:int farray.
    (permut(t1,t2) and permut(t2,t3)) -> permut(t1,t3)

axiom permut_exchange :
  forall t:int farray. forall i:int. forall j:int.
    permut(t, update(update(t,i,t[j]),j,t[i]))

logic sub_permut : int, int, int farray, int farray -> prop

axiom sub_permut_refl :
  forall t:int farray. forall g:int. forall d:int.
    sub_permut(g,d,t,t)

axiom sub_permut_sym :
  forall t1:int farray. forall t2:int farray. forall g:int. forall d:int.
    sub_permut(g,d,t1,t2) -> sub_permut(g,d,t2,t1)

axiom sub_permut_trans :
  forall t1:int farray. forall t2:int farray. forall t3:int farray.
  forall g:int. forall d:int.
    sub_permut(g,d,t1,t2) -> sub_permut(g,d,t2,t3) ->
    sub_permut(g,d,t1,t3)
```

```
axiom sub_permut_exchange_1 :
  forall t:int farray.
  forall g:int. forall d:int. forall i:int. forall j:int.
    (g <= i and i <= d and g <= j and j <= d) ->
    sub_permut(g,d,t,update(update(t,i,t[j]),j,t[i]))

axiom sub_permut_exchange_2 :
  forall t1:int farray. forall t2:int farray.
  forall g:int. forall d:int. forall i:int. forall j:int.
    (g <= i and i <= d and g <= j and j <= d and
    exchange(t1,t2,i,j)) -> sub_permut(g,d,t1,t2)

axiom sub_permut_weakening :
  forall a1:int farray. forall a2:int farray.
  forall l1:int. forall r1:int. forall l2:int. forall r2:int.
    l1 <= l2 and l2 <= r2 and r2 <= r1 ->
    sub_permut(l2,r2,a1,a2) ->
    sub_permut(l1,r1,a1,a2)

axiom sub_permut_eq :
  forall a1:int farray. forall a2:int farray.
  forall l:int. forall r:int.
    l <= r ->
    sub_permut(l,r,a1,a2) ->
    forall i:int. (i < l or r < i) -> access(a2,i) = access(a1,i)

axiom sub_permut_permut :
  forall t1:int farray. forall t2:int farray. forall g:int. forall d:int.
    sub_permut(g,d,t1,t2) -> permut(t1,t2)

(* Axioms about array_length *)

axiom array_length_update :
  forall t:int farray. forall i:int. forall v:int.
    array_length(update(t,i,v)) = array_length(t)

axiom sub_permut_array_length :
  forall t1:int farray. forall t2:int farray. forall g:int. forall d:int.
    sub_permut(g,d,t1,t2) -> array_length(t1) = array_length(t2)

axiom permut_array_length :
  forall t1:int farray. forall t2:int farray.
    permut(t1,t2) -> array_length(t1) = array_length(t2)
```

2.5 Provers specificities

2.5.1 Coq

Several Coq modules are delivered with Why. If installation is properly done, they should be found in subdirectory `user-contrib` of the Coq standard library (usually `/usr/lib/coq` unless some other path was specified when installing Coq).

Types. Types are mapped as follows:

Why	Coq
`unit`	`unit`
`bool`	`bool`
`int`	`Z`
`real`	`R`
τ `array`	`(array` τ`)`, from module `WhyArrays`

Arrays. Arrays are introduced in module `WhyArrays`. Some tactics are provided to help the user simplifying array expressions in proof obligations:

`WhyArrays` : repeatedly simplifies `access`/`update`using combinations and `array_length` expressions

`AccessSame` : rewrites `access` (`update` t i v) i into v, simplifies with `WhyArrays` and attempts `omega` on every subgoal

`AccessOther` : rewrites `access` (`update` $t\ i\ v$) j into `access` $t\ j$, simplifies with `WhyArrays` and attempts `omega` on every subgoal

`ArraySubst` t : similar to `subst` t, with additional simplifications

2.5.2 PVS

Types. Types are mapped as follows:

Why	Coq
`unit`	`unit`, from theory `why`
`bool`	`bool`
`int`	`int`
`real`	`real`
τ `array`	`warray[`τ`]`, from theory `why`

Arrays. An array whose elements are of type `T` is a PVS value of type `warray[T]`. This type is defined in the theory `why_arrays`, as a pair of an integer, the array length, and a function mapping indices to elements :

```
warray: TYPE = [ n:int, [ below(n) -> T ] ]
```

Two other theories, `why_int_array_pred` and `why_array_pred`, introduce the various predicates over arrays presented in Section 2.4.

Index