Why: a multi-language multi-prover verification condition generator

Jean-Christophe Filliâtre

LRI – CNRS UMR 8623
Université Paris-Sud, France
filliatr@lri.fr

Abstract. This article introduces the verification tool Why. This tool produces verification conditions from annotated programs given as input. It differs from other systems in that it accepts several languages as input (currently C and ML, and JAVA with the help of the companion tool Krakatoa) and outputs conditions for several existing provers (currently Coq, PVS, HOL Light, Mizar, Simplify and haRVey). It also provides a great safety through some de Bruijn criterion: once the obligations are established, a proof that the program satisfies its specification is built and type-checked automatically.

Keywords: Verification, C, Java, ML

1 Introduction

This article introduces the verification tool Why [6]. When observed from outside, this tool resembles many others: it takes annotated programs as input and produces verification conditions as output. However, it relies on a technology and on some design choices which are less common.

First, Why is not coming with its own proof tool; instead, it produces conditions for *existing proof tools*. This is particularly important when interactive proof is needed, which always ends up to be the case. Writing a good proof assistant is a tremendous amount of work, which should be left to the field experts. Currently, Why is interfaced with four proof assistants, namely Coq [1], PVS [4], Mizar [2] and HOL Light [14]. The same remark applies as well to automatic decision procedures: instead of involving them inside the verification tool, it is simpler to use existing tools as a back-end. Currently, Why is interfaced with two decision procedures, namely Simplify [5] (part of ESC/Java) and haRVey [22]. Adding a back-end for a new proof tool is really simple—a matter of a few hours—and the requirements over the logic are very small—only a first order minimal logic is needed.

If interfacing to external proof tools may increase the trust in the verification process, it is also important to bring evidence that the verification tool itself is trustworthy. This is particularly important when the tool is discharging some obligations by itself or is involving complex treatments for some particular constructs of the language (C abrupt terminations, JAVA or ML exceptions, etc.)

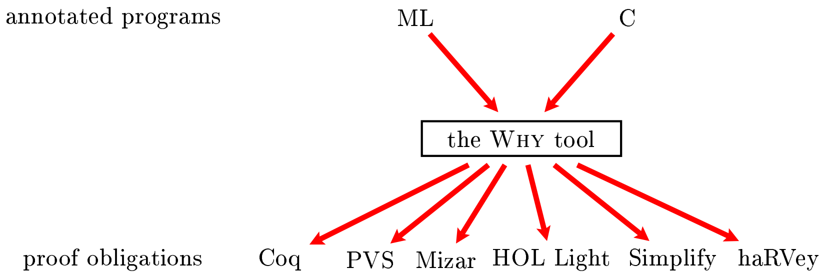

Fig. 1. The Why tool

One solution is to embed the whole verification process inside a proof assistant. Why adopts a slightly different approaches where a proof that the program satisfies its specification is checked *a posteriori*, once all obligations are discharged, either by the user or automatically by Why itself. This check is purely automatic and thus can be considered as a de Bruijn criterion.

Finally, Why is not limited to one input language. Instead, it provides its own internal language, which existing programming languages can be compiled to. This internal language—let us call it WL from now on—is a small ML-like language with imperative features (references and arrays), exceptions and annotations. In the ML tradition, WL merges expressions, statements, local variables and functions into a single syntactic class, which eases symbolic manipulations and limits the number of cases to consider when computing weakest preconditions or verification conditions. Similarly, exceptions are used to deal with abrupt terminations such as `return`, `break` or `continue` when translating C or JAVA programs, and thus there is no need to implement special rules for these constructs; the rules for exceptions are giving the excepted verification conditions. Why currently interprets C programs, and JAVA programs with the help of the companion tool Krakatoa [7]. Why also provides an input syntax for its internal language WL, which is very close to a subset of Objective Caml [3].

This paper is organized as follows. Section 2 exposes the principles and theoretical foundations of the tool. Then Section 3 details its practical use, illustrated on an example. Section 4 is briefly introducing the verification of JAVA programs using the companion tool Krakatoa. Finally, some expected future developments are described.

2 Principles

The theoretical foundations of Why are detailed in the author's PhD thesis [10, 12]. Basically, Why is building a functional interpretation of the imperative program given as input. Expressed in Type Theory, this interpretation mixes the computational and logical parts of the program, using dependent tuples. The computational part is entirely built by the tool, using an effect inference and a notion of monads parameterized with effects [9]. The logical part is usually incomplete, the missing pieces being precisely the proof obligations. These verification conditions are simply collected by traversing the functional interpretation.

For instance, the following Hoare triple

$$\{\ x > 0\ \wedge\ y > 1\ \}\ x\ :=\ x + 1;\ y\ :=\ y \times x\ \{\ y > \overleftarrow{y}\ \} \tag{1}$$

is translated into a proof of

$$\forall x_0, y_0.\ x_0 > 0\ \wedge\ y_0 > 1 \Rightarrow \exists x_1, y_1.\ y_1 > y_0$$

The computational part of this proof consists in interpreting the two assignments $x\ :=\ x + 1$ and $y\ :=\ y \times x$, as the computation of final values x_1 and y_1 from initial values x_0 and y_0. The logical part consists in an hypothesis $x_0 > 0 \wedge y_0 > 1$ and a proof of $y_1 > y_0$. The latter is a verification condition. The entire proof looks like

$$\begin{array}{l}\lambda x_0, y_0.\ \lambda h : x_0 > 0\ \wedge\ y_0 > 1. \\ \quad \mathsf{let}\ x_1 = x_0 + 1\ \mathsf{in} \\ \quad \mathsf{let}\ y_1 = y_0 \times x_1\ \mathsf{in} \\ \quad (x_1, y_1, \Box : y_1 > y_0)\end{array}$$

where the proof obligation is denoted by a box. The exact statement of this obligation is thus

$$x_0 > 0\ \wedge\ y_0 > 1 \Rightarrow x_1 = x_0 + 1 \Rightarrow y_1 = y_0 \times x_1 \Rightarrow y_1 > y_0$$

and simplifies, after substitution, to

$$x_0 > 0\ \wedge\ y_0 > 1 \Rightarrow y_0 \times (x_0 + 1) > y_0$$

As illustrated by this example, the verification conditions for the usual imperative constructs are similar to the ones given by Hoare logic. This was clearly demonstrated by verifying with Why the program *Find*, proved correct by Hoare himself thirty years ago [16]. The verifications conditions appeared to be *exactly* the same, and the proofs were conducted in Coq following Hoare's paper in a straightforward way [11]. But it is important to notice that Why is *not* implementing some Hoare logic, even if it seems to do so observationally.

The remaining of this section describes the Why technology in more details. The WL concrete syntax is used throughout this section.

2.1 Types, programs and annotations

Annotations are written in first-order logic. Terms (t) are made of constants, variables and application of function symbols to terms:

$$t\ ::=\ \mathit{constant} \mid x \mid x\texttt{@}L \mid f(t, \ldots, t)$$

When relevant, a variable x can be annotated with a label L, written x@L, to denote the value of a mutable x at a given program point L. It is important to notice that f is a function symbol from the logical world, which cannot be defined or used in programs. Predicates (p) are built with the usual constructs of first-order logic:

$$\begin{array}{rcl} p & ::= & x \mid x(t,\dots,t) \mid \texttt{true} \mid \texttt{false} \mid \texttt{not}\ p \mid p\ \texttt{and}\ p \mid p\ \texttt{or}\ p \mid \\ & & \texttt{if}\ t\ \texttt{then}\ p\ \texttt{else}\ p \mid \texttt{forall}\ x:\beta\texttt{.}\ p \mid \texttt{exists}\ x:\beta\texttt{.}\ p \end{array}$$

In the conditional construct `if` t `then` p_1 `else` p_2, t is a boolean term, not a proposition; indeed, propositions are not necessary booleans (we do not assume classical logic *a priori*). For the same reason, `true` and `false` are not the two boolean constants (also written `true` and `false` with no possible ambiguity), but the two propositions always and never valid. Quantification is limited to primitive types.

Primitive types (β) contain a type `unit` with a single value `void`, the boolean type `bool`, a type `int` for integers, a type `float` for floating point numbers:

$$\beta \ ::= \ \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \mid \texttt{float} \mid x$$

A type variable x stands for an abstract type introduced by the user; it is supposed to be defined on the logical side and thus is pure (*i.e.* does not contain mutable parts).

Types for programs distinguish types for values (τ) and types for computations (κ). The former include primitive types, references, arrays and function types. The latter add a precondition, an effect and a postcondition to a value type. An effect (ϵ) is made of three lists of variables: the mutables possibly accessed and those possibly updated.

$$\begin{array}{rcl} \tau & ::= & \beta \mid \beta\ \texttt{ref} \mid \beta\ \texttt{array} \mid x\texttt{:}\tau\ \texttt{->}\ \kappa \\ \kappa & ::= & \{p\}\ \tau\ \epsilon\ \{p\} \\ \epsilon & ::= & \texttt{reads}\ x,\dots,x\ \texttt{writes}\ x,\dots,x \end{array}$$

Programs are built from usual constructs of ML with references and arrays, with no distinction between expressions and statements:

$$\begin{array}{rcl} e & ::= & \{p\}\ e\ \{p\} \mid \\ & & t \mid \texttt{!}x \mid x\ \texttt{:=}\ e \mid \texttt{ref}\ e \mid x\texttt{[}e\texttt{]} \mid x\texttt{[}e\texttt{]}\ \texttt{:=}\ e \mid e\texttt{;}\ e \mid L\texttt{:}e \mid \\ & & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e \mid \texttt{fun}\ \texttt{(}x\texttt{:}\tau\texttt{)}\ \texttt{->}\ e \mid \texttt{(}e\ e\texttt{)} \mid \\ & & \texttt{rec}\ x:\tau\ \{\texttt{variant}\ t\} = e \mid \\ & & \texttt{while}\ e\ \texttt{do}\ \{\texttt{invariant}\ p\ \texttt{variant}\ t\}\ e\ \texttt{done} \end{array}$$

Any expression can be given pre- and postcondition using the Hoare triple notation $\{p_1\}\ e\ \{p_2\}$. In the postcondition, the variable *result* is bound to the result of the computation and an empty label refers to the precondition point (*i.e.* x@ stands in p_2 for the value of x before the evaluation of e). The construct L:e explicitly places a label L right before the evaluation of e, to be used in annotations inside e. Recursive functions and loops are given a variant: it is a term

t which must decrease for a given well-founded order relation. This relation can be specified explicitly; to simplify this presentation we assume here the usual order relation on natural numbers. Loops can be given invariants (this is for convenience, since the Hoare triple already allows to annotate the loops, both inside and outside).

Typing rules closely follow those of traditional ML typing, with additional inference of effects and check for well-formedness of pre- and postconditions. All these rules are given in [12].

2.2 Aliasing exclusion

The reader may have already noticed severe restrictions with respect to ML. First, references and arrays are limited to primitive types. Second, left values are limited to variables (in accessing or updating references or arrays). The goal behind these restrictions is to be able to get a precise effect analysis (*i.e.* to know *for each variable separately* if it is possibly accessed or modified). This leads to a very precise interpretation in Type Theory and thus to very natural proof obligations, as illustrated at the beginning of this section. Consequently, aliasing between different mutable variables must be excluded and this is guaranteed by typing rules. Mainly, rules for `let in` and function application prevent the user from creating an alias; see [12] for details.

However, we show in Section 4 that it is still possible to cover *all features* of ML, C or JAVA programs, using a low level memory model.

2.3 Weakest preconditions

To get the expected obligations, the input program must be adequately annotated: intermediate program points must be given the right annotations. Why discharges the user from this painful task using a weakest precondition (WP) calculus. The weakest precondition of a program e for a postcondition q is written $wp(e, q)$.

The originality of this WP calculus is its treatment of annotated subexpressions *i.e.* of Hoare triples:

$$wp(\{p'\}\, e\, \{q'\}, q) \;=\; p' \;\wedge\; \forall result.\, \forall\omega.\; q' \Rightarrow q$$

where ω stands for the set of variables possibly modified by e. This formulation expresses a crucial notion of modularity: as seen from outside, the Hoare triple $\{p'\}\, e\, \{q'\}$ is a black box with a specification given by p', q' and its effect, and its code e is not examined to compute the weakest precondition for q. Instead, a new WP calculus starts inside e with the postcondition q'.

In particular, a loop and a function call are naturally annotated with the loop invariant and the function specification, respectively. Thus they are seen as Hoare triples by the WP calculus and do not need any special treatment.

The weakest precondition is defined for the other constructs in a usual way:

$$\begin{aligned}
wp(t,q) &= q[\mathit{result} \leftarrow t] \\
wp(\texttt{!}x,q) &= q[\mathit{result} \leftarrow x] \\
wp(x\ \texttt{:=}\ e,q) &= wp(e, q[\mathit{result} \leftarrow \texttt{void}; x \leftarrow \mathit{result}]) \\
wp(t\texttt{[}e\texttt{]},q) &= wp(e, q[\mathit{result} \leftarrow (\mathit{access}\ t\ \mathit{result})]) \\
wp(t\texttt{[}e_1\texttt{]}\ \texttt{:=}\ e_2,q) &= wp(e_1, wp(e_2,q_1)[v_1 \leftarrow \mathit{result}]) \\
&\quad \text{with } q_1 = q[\mathit{result} \leftarrow \texttt{void}; t \leftarrow (\mathit{update}\ t\ v_1\ \mathit{result})] \\
wp(e_1\texttt{;}\ e_2,q) &= wp(e_1, wp(e_2,q)) \\
wp(L\texttt{:}e,q) &= wp(e,q)[x\texttt{@}L \leftarrow x] \\
wp(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,q) &= wp(e_1, \texttt{if}\ \mathit{result}\ \texttt{then}\ wp(e_2,q)\ \texttt{else}\ wp(e_3,q)) \\
wp(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2,q) &= wp(e_1, wp(e_2,q)[x \leftarrow \mathit{result}])
\end{aligned}$$

where *access* and *update* are the purely functional operations over the arrays as modeled on the logical side.

2.4 Exceptions

Why also supports exceptions. For the clarity of this presentation we assume a single exception E with one argument of type `int`; but Why supports an unlimited number of exceptions declared by the user.

First, the notion of effect is extended to indicate the possible raise of exception E:

$$\epsilon\ ::=\ \texttt{reads}\ x,\dots,x\ \texttt{writes}\ x,\dots,x\ [\texttt{raises}\ E]$$

Postconditions are also extended to include a second predicate, namely the property to be valid when exception E is raised:

$$\kappa\ ::=\ \{p\}\,\tau\,\epsilon\,\{p \mid E \Rightarrow p\}$$

Finally, the Hoare triple is extended similarly and constructs to raise and catch exception E are added:

$$e\ ::=\ \dots \mid \{p\}\,e\,\{p \mid E \Rightarrow p\} \mid \texttt{raise (}E\ e\texttt{)} \mid \texttt{try}\ e\ \texttt{with}\ E\ x\ \texttt{->}\ e\ \texttt{end}$$

The notion of weakest precondition is extended accordingly. It becomes a ternary operator $wp(e,q,r)$ where q is the normal postcondition and r the exceptional one. All rules of the WP calculus but one are unchanged *i.e.* r is just added as a third argument everywhere. Only the rule for the Hoare triple must be adapted, as follows:

$$wp(\{p'\}\,e\,\{q' \mid E \Rightarrow r'\},q,r)\ =\ p'\ \wedge\ \forall \mathit{result}.\,\forall\omega.\,(q' \Rightarrow q\ \wedge\ r' \Rightarrow r)$$

Two new rules are added to handle the `raise` and `try` constructs:

$$\begin{aligned}
wp(\texttt{raise (}E\ e\texttt{)},q,r) &= wp(e,r,r) \\
wp(\texttt{try}\ e_1\ \texttt{with}\ E\ x\ \texttt{->}\ e_2\ \texttt{end},q,r) &= wp(e_1,q,wp(e_2,q,r)[x \leftarrow \mathit{result}])
\end{aligned}$$

2.5 Discharging conditions automatically

Why is discharging a lot of proof obligations by itself. As already mentioned, safety is ensured since Why builds corresponding proof objects (proof terms for the Coq logic). In particular, it is very important to discharge tautologies coming from the WP calculus. By construction, such tautologies lie in linear first-order minimal logic, which correspond to the following rules

$$\frac{}{\Gamma, P \vdash P} \qquad \frac{\Gamma, P, Q \vdash R}{\Gamma, P, P \Rightarrow Q \vdash R} \qquad \frac{\Gamma, P, Q \vdash R}{\Gamma, P \wedge Q \vdash R} \qquad \frac{}{\Gamma, \forall x. P(x) \vdash P(t)}$$

$$\frac{\Gamma, P(t), \forall y. Q(t, y) \vdash R}{\Gamma, P(t), \forall x, y. P(x) \Rightarrow Q(x, y) \vdash R} \qquad \frac{\Gamma, \forall x. P(x), \forall x. Q(x) \vdash R}{\Gamma, \forall x. (P(x) \wedge Q(x)) \vdash R}$$

where x and y (resp. t) may stand for several variables (resp. several terms). Why is implementing a goal directed proof search using these rules, which returns a proof object.

2.6 Verifying C programs

Why deals with C programs, written in standard ANSI C and annotated using a particular kind of C comments. Internally, Why translates C programs into its internal language WL. Currently, only a simple translation is performed, covering a subset of C where pointers usage is restricted to reference passing and arrays, with no pointer aliasing. A lower-level approach using a memory model, in a way similar to the Krakatoa approach for JAVA programs, is currently undertaken (see Section 4 and 5). We only describe here the simple C to WL translation.

Types. Currently, only C base types (`void`, `char`, `short`, `int`, `long`, `float` and `double`), arrays and pointers are considered. Following the same restrictions as WL (see Section 2.2), arrays and pointers are limited to base types. Type `void` is mapped to WL type `unit`; all integer types are mapped to WL type `int`; both floating point types are mapped to WL type `float`. In the following, β stands for a C base type or its WL counterpart, without distinction.

C variable declarations allowed and their WL translations are given in the following schema:

- global variable
  - $[\![\beta\ \mathtt{x}]\!] = \mathtt{x} : \beta\ \mathtt{ref}$
  - $[\![\beta{*}\ \mathtt{x}]\!] = \mathtt{x} : \beta\ \mathtt{ref}$
  - $[\![\beta\ \mathtt{x[]}]\!] = \mathtt{x} : \beta\ \mathtt{array}$
- local variable
  - $[\![\beta\ \mathtt{x}]\!] = \mathtt{x} : \beta\ \mathtt{ref}$
- function parameter
  - $[\![\beta\ \mathtt{x}]\!] = \mathtt{x} : \beta$
  - $[\![\beta{*}\ \mathtt{x}]\!] = \mathtt{x} : \beta\ \mathtt{ref}$
  - $[\![\beta\ \mathtt{x[]}]\!] = \mathtt{x} : \beta\ \mathtt{array}$

In the following we write $\mathtt{x} : \tau_1/\tau_2$ if a variable `x` has C type τ_1 and WL type τ_2.

Left values. The translation of a C left value e is written $[\![e]\!]_l$ and defined by:

$$[\![\mathtt{x}]\!] = \mathtt{x} \qquad \text{if } \mathtt{x} : \beta/\beta\ \mathtt{ref}$$

$$[\![\mathtt{*x}]\!] = \mathtt{x} \qquad \text{if } \mathtt{x} : \beta\mathtt{*}/\beta\ \mathtt{ref}$$

$$[\![\mathtt{x}\,\mathtt{[}e\mathtt{]}]\!] = \mathtt{x}\,\mathtt{[}e\mathtt{]} \qquad \text{if } \mathtt{x} : \beta\mathtt{[]}/\beta\ \mathtt{array}$$

Expressions. The translation of a C expression e is written $[\![e]\!]$ and defined by:

- $[\![\mathtt{x}]\!] = \begin{cases} \mathtt{!x} & \text{if } \mathtt{x} : \beta/\beta\ \mathtt{ref} \\ \mathtt{x} & \text{otherwise} \end{cases}$
- $[\![e_1\ \mathtt{=}\ e_2]\!] = \begin{cases} \mathtt{x\ :=}\ [\![e_2]\!]\mathtt{;\ !x} & \text{if } [\![e_1]\!]_l = \mathtt{x} \\ \mathtt{let}\ i\ \mathtt{=}\ [\![e'_1]\!]\ \mathtt{in\ x[}i\mathtt{]\ :=}\ [\![e_2]\!]\mathtt{;\ x[}i\mathtt{]} & \text{if } [\![e_1]\!]_l = \mathtt{x[}e'_1\mathtt{]} \end{cases}$
- $[\![e_1\ op\mathtt{=}\ e_2]\!] = \begin{cases} \mathtt{x\ :=\ !x}\ op\ [\![e_2]\!]\mathtt{;\ !x} & \text{if } [\![e_1]\!]_l = \mathtt{x} \\ \mathtt{let}\ i\ \mathtt{=}\ [\![e'_1]\!]\ \mathtt{in\ x[}i\mathtt{]\ :=\ x[}i\mathtt{]}\ op\ [\![e_2]\!]\mathtt{;\ x[}i\mathtt{]} & \text{if } [\![e_1]\!]_l = \mathtt{x[}e'_1\mathtt{]} \end{cases}$

  with $op \in \{\mathtt{+}, \mathtt{-}, \mathtt{*}, \mathtt{/}, \mathtt{\%}, \mathtt{\&}, \mathtt{\hat{}}, \mathtt{|}\}$. Notice that $\mathtt{++}e$ is $e\ \mathtt{+=\ 1}$ and $\mathtt{--}e$ is $e\ \mathtt{-=\ 1}$.
- $[\![e\mathtt{++}]\!] = \begin{cases} \mathtt{let}\ v\ \mathtt{=\ !x\ in\ x\ :=\ !x\ +\ 1;\ !}v & \text{if } [\![e]\!]_l = \mathtt{x} \\ \mathtt{let}\ i\ \mathtt{=}\ [\![e']\!]\ \mathtt{in\ let}\ v\ \mathtt{=\ x[}i\mathtt{]\ in\ x[}i\mathtt{]\ :=\ x[}i\mathtt{]\ +\ 1;}\ v & \text{if } [\![e]\!]_l = \mathtt{x[}e'\mathtt{]} \end{cases}$
- $[\![e_1\ \mathtt{,}\ e_2]\!] = [\![e_1]\!]\mathtt{;}\ [\![e_2]\!]$
- $[\![e_1\ \mathtt{?}\ e_2\ \mathtt{:}\ e_3]\!] = \mathtt{if}\ [\![e_1]\!]\ \mathtt{then}\ [\![e_2]\!]\ \mathtt{else}\ [\![e_3]\!]$
- $[\![e_1\ op\ e_2]\!] = \mathtt{let}\ v_1\ \mathtt{=}\ [\![e_1]\!]\ \mathtt{in\ let}\ v_2\ \mathtt{=}\ [\![e_2]\!]\ \mathtt{in}\ v_1\ op\ v_2$

  with $op \in \{\mathtt{+}, \mathtt{-}, \mathtt{*}, \mathtt{/}, \mathtt{\%}, \mathtt{\&}, \mathtt{\hat{}}, \mathtt{|}\}$
- $[\![e_1\ op\ e_2]\!] = \mathtt{if}\ [\![e_1\ op\ e_2]\!]_b\ \mathtt{then\ 1\ else\ 0}$

  with $op \in \{\mathtt{==}, \mathtt{!=}, \mathtt{>}, \mathtt{>=}, \mathtt{<}, \mathtt{<=}, \mathtt{\&\&}, \mathtt{||}\}$
- $[\![\mathtt{!}e]\!] = \mathtt{if}\ [\![e]\!]_b\ \mathtt{then\ 0\ else\ 1}$
- $[\![\mathtt{-}e]\!] = \mathtt{-}[\![e]\!]$
- $[\![\mathtt{*x}]\!] = \mathtt{!x}$, if $\mathtt{x} : \beta\mathtt{*}/\beta\ \mathtt{ref}$
- $[\![\mathtt{\&x}]\!] = \mathtt{x}$, if $\mathtt{x} : \beta/\beta\ \mathtt{ref}$
- $[\![\mathtt{x[}e\mathtt{]}]\!] = \mathtt{x[}[\![e]\!]\mathtt{]}$
- $[\![c]\!] = c$, for a constant c
- $[\![\mathtt{f(}e_1, \ldots, e_n\mathtt{)}]\!] = \mathtt{let}\ v_1\ \mathtt{=}\ [\![e_1]\!]\ \mathtt{in}\ \ldots \mathtt{let}\ v_n\ \mathtt{=}\ [\![e_n]\!]\ \mathtt{in\ (f}\ v_1\ \ldots\ v_n\mathtt{)}$

Boolean expressions. C does not have booleans. However, the translation is more natural when a C expression e used as a boolean is directly translated into an WL boolean. Such an interpretation is written $[\![e]\!]_b$ and defined by:

- $[\![e_1\ op\ e_2]\!] = \mathtt{let}\ v_1\ \mathtt{=}\ [\![e_1]\!]\ \mathtt{in\ let}\ v_2\ \mathtt{=}\ [\![e_2]\!]\ \mathtt{in}\ v_1\ op\ v_2$,

  with $op \in \{\mathtt{==}, \mathtt{!=}, \mathtt{>}, \mathtt{>=}, \mathtt{<}, \mathtt{<=}\}$
- $[\![e_1\ \mathtt{\&\&}\ e_2]\!] = \mathtt{if}\ [\![e_1]\!]_b\ \mathtt{then}\ [\![e_2]\!]_b\ \mathtt{else\ false}$
- $[\![e_1\ \mathtt{||}\ e_2]\!] = \mathtt{if}\ [\![e_1]\!]_b\ \mathtt{then\ true\ else}\ [\![e_2]\!]_b$
- $[\![\mathtt{!}e]\!] = \mathtt{not}\ [\![e]\!]_b$
- $[\![e]\!]_b = [\![e]\!]\ \mathtt{<>\ 0}$, otherwise

Note: When translating expressions and boolean expressions, coercions from integers to floating points are inserted when needed.

Statements. The translation of a C statement s is written $[\![s]\!]_s$ and defined by:

– $[\![e]\!]_s = \texttt{let \_ = } [\![e]\!] \texttt{ in void}$
– $[\![e_1\,;\ e_2]\!]_s = [\![e_1]\!]_s\texttt{;}\ [\![e_2]\!]_s$
– $[\![\{\ \beta_1\ \texttt{x}_1\texttt{=}\ e_1\,;\ \ldots\ \beta_n\ \texttt{x}_n\texttt{=}\ e_n\,;\ s\ \}]\!]_s$
  $= \texttt{let x}_1 \texttt{ = ref } [\![e_1]\!] \texttt{ in } \ldots \texttt{ let x}_n \texttt{ = ref } [\![e_n]\!] \texttt{ in } [\![s]\!]_s$

The constructs `break` and `continue` are translated using two WL exceptions `Break` and `Continue` (without argument):

– $[\![\texttt{break}]\!]_s = \texttt{raise Break}$
– $[\![\texttt{continue}]\!]_s = \texttt{raise Continue}$

Helper functions *break*(m) and *continue*(m) are defined as

$$\begin{aligned} \mathit{break}(m) &= \texttt{try } m \texttt{ with Break -> void end} \\ \mathit{continue}(m) &= \texttt{try } m \texttt{ with Continue -> void end} \end{aligned}$$

whenever m may raise `Break` or `Continue`, respectively, and as m otherwise. Then loops are translated as follows:

– $[\![\texttt{for(}e_1\,;\ e_2\,;\ s_3\texttt{)}\ s]\!]_s =$
  $[\![e_1]\!]\texttt{;}\ \mathit{break}(\texttt{while } [\![e_2]\!] \texttt{ do } \mathit{continue}([\![s\texttt{;}\ s_3]\!]_s) \texttt{ done})$
– $[\![\texttt{while (}e\texttt{)}\ s]\!]_s = \mathit{break}(\texttt{while } [\![e]\!] \texttt{ do } \mathit{continue}([\![s]\!]_s) \texttt{ done})$
– $[\![\texttt{do } e \texttt{ while(}s\texttt{)}]\!]_s = [\![s\texttt{; while (}e\texttt{)}\ s]\!]_s$

Finally, abrupt `return`s are also translated using an WL exception `Return` with the returned value as argument (thus, such an exception is introduced for each possible returned type):

– $[\![\texttt{return } e]\!]_s = \texttt{raise (Return } [\![e]\!]\texttt{)}$
– $[\![\texttt{f(}\tau_1\ \texttt{x}_1, \ldots, \tau_n\ \texttt{x}_n\texttt{)}\ \{\ s\ \}]\!]_s =$
  $\texttt{let f (x}_1\texttt{:}[\![\tau_1]\!]\texttt{)} \ldots \texttt{(x}_n\texttt{:}[\![\tau_n]\!]\texttt{) = try } [\![s]\!]_s \texttt{ with Return v -> v end}$

3 Practical use

Why is run as a batch compiler, taking input source files on its command line and producing files containing the verification conditions. The prover is selected using a command line option. Input source files can be WL or C programs, with a common syntax for annotations. The latter is a homemade syntax for first-order predicates.

WL syntax is very close to Objective Caml's one [3], with a few differences: annotations are part of the syntax, enclosed with brackets; there is no type inference and thus types must be explicitly declared; finally, array syntax is C-like. For instance, here is how the Hoare triple (1) given in Section 2 is written in Why's WL input syntax:

```
{ x > 0 and y > 1 } begin x := !x + 1; y := !y * !x end { y > y@ }
```

where `y@` is the notation for the value of `y` in the prestate.

C programs are written in the standard ANSI C syntax and annotated using the distinguished kind of C comments `/*@ ... */`. Here is for instance how the Hoare triple (1) is passed to Why in C syntax:

```
/*@ x > 0 and y > 1 */ { x++; y *= x; } /*@ y > y@ */
```

The `goto` set apart, all C constructs are covered. C programs are only limited by: (1) the aliasing restriction already discussed in Section 2.2; and (2) the absence of pointer arithmetic, which means that the notions of arrays and pointers are clearly separated.

3.1 Example

We illustrate the use of Why on the verification of the small piece of C code given in Figure 2. This is a function `index` which looks for a value `v` in an array `t` of integers. The size of `t` is given as a parameter `n`. When `v` is found, an index is returned giving one position for `v` in `t`; otherwise `n` is returned, meaning that `v` was not found in `t`. The code uses a `while` loop, from which we exit with a `break` as soon as `v` is found. (The code could be written with a `for` loop or could use a `return` to terminate as soon as `v` is found; the annotations and proofs would be exactly the same.)

```
int index(int t[], int n, int v) {
  int i = 0;
  while (i < n) {
    if (t[i] == v) break;
    i++;
  }
  return i;
}
```

Fig. 2. The C function `index`

First, we give the function a specification, as a precondition and a post-condition. Both are inserted respectively before and after the function body. In particular, the precondition clearly appears *after* the function parameters, thus expressing that they are part of the prestate. The precondition expresses that `n` is the size of `t`:

```
/*@ array_length(t) = n */
```

Note that `array_length` is not a C function; it is used inside an annotation and thus belongs to the logical world (the *model*), where C arrays are modeled by some datatype for which the size is computable (by `array_length`). Though C arrays do not have a computable size, this is a mandatory notion on the logical side to generate conditions expressing that array accesses and updates are legal, *i.e.* done within the array bounds. The postcondition simply expresses that the returned value is an index where `v` occurs in `t`, as soon as it is within the bounds:

```
/*@ 0 <= result < n -> t[result] = v */
```

The next step consists in annotating the loop, with an invariant and a variant. The invariant expresses that `v` was not yet found, *i.e.* does not appear in `t` for indices less that `i`. It also maintains the property `0 <= i`, which is needed to prove that the access `t[i]` is legal. The variant is `n - i` (when no order relation is specified, it defaults to the well-founded relation R on $\mathbb{Z}^2$ defined by $x\ R\ y \equiv x < y\ \wedge\ 0 \leq y$). The C code fully annotated in given in Figure 3; note that this is still ANSI C code.

```
int index(int t[], int n, int v) /*@ array_length(t) = n */ {
  int i = 0;
  while (i < n)
    /*@ invariant 0 <= i  variant n - i */ {
    if (t[i] == v) break;
    i++;
  }
  return i;
}
/*@ 0 <= result < n -> t[result] = v */
```

Fig. 3. The C function `index` annotated

Assuming this code to be in `search.c` and a user willing to use Coq as prover, Why is simply invoked by

```
why --coq search.c
```

A Coq file `search_why.v` is produced, which contains five lemmas with empty proofs to be filled in. These lemmas express: (1) the legality of the array access `t[i]`; (2) the validity of the postcondition when exiting the loop using `break`; (3) the preservation of the invariant by the loop body, together with the decreasing of the variant; (4) the validity of the invariant when entering the loop; and (5) the validity of the postcondition when exiting the loop normally. All lemmas are actually automatically discharged by Coq, except the third one which requires two lines of proof script.

These obligations are exactly what the user is expecting and the internal translation to WL code, including the use of exceptions implied by the `break` construct, is totally transparent. This internal WL code is given Figure 4 (it can be obtained from Why automatically). As expected, the C code annotations become annotations for the WL code without any change.

```
let index = fun (t: int array) (n:int) (v:int) ->
  { array_length(t) = n }
  let i = ref 0 in
  begin
    try
      while !i < n do
        { invariant 0 <= i  variant n - i }
        if t[!i] = v then raise Break;
        i := !i + 1;
      done
    with Break => void end;
    !i
  end
  { 0 <= result < n -> t[result] = v }
```

Fig. 4. The WL translation for function `index`

3.2 Availability

Why is open source and freely available from `http://why.lri.fr/`. It is written in Objective Caml [3]. Documentation includes a tutorial, a reference manual and many examples, all freely available from the web site.

4 Verifying Java programs

Why does not handle JAVA programs by itself. However, this can be achieved by using Why in combination with the Krakatoa tool [7]. Developed by C. Marché, C. Paulin and X. Urbain, this other tool tackles JAVA programs annotated using the Java Modeling Language (JML for short) [19]. Such programs are translated into Why input code, expressing the semantics of the original JAVA code. A model is produced beside. The combined use of Why and Krakatoa is illustrated on Figure 5.

This model is a low level memory model, where objects are addresses in the JAVA heap. This heap is modeled as a mapping from addresses to typed objects, which contain primitive values or other addresses. From the point of view of Why, there is only one mutable data, which is the reference on the current state of the heap. Consequently, the anti-aliasing restrictions exposed in Section 2.2 do not apply and the *whole* sequential JAVA is covered. Currently, only the Coq and Simplify outputs of Why are meaningful when verifying JAVA-JML programs using Krakatoa. Adapting the model to other provers is feasible, though nontrivial.

Other tools such as LOOP [24] aim at the formal verification of JML-annotated JAVA programs. One obvious advantage of the Why-Krakatoa approach is to offer the use of multiple back-end provers, while LOOP is linked to PVS for instance. In particular, the use of the Simplify decision procedure gives Krakatoa a high

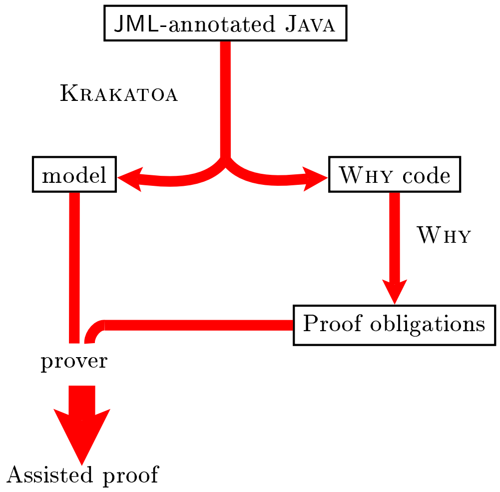

Fig. 5. Verifying JAVA programs using Krakatoa and Why

degree of automation. It even compares to ESC/Java when applied to static analysis typical specifications (non-null pointers, valid access in arrays, etc.), though Krakatoa is not restricted to this use.

5 Future work

Why can be improved in many directions. Here are some of the planned developments.

Symbolic evaluation. Why is the ideal place to perform symbolic evaluation of annotated programs. As suggested in [8], this is an efficient way to debug specifications. For instance, a loop can be unrolled a given number of times and verification conditions will be generated for many different control flow paths, corresponding to the first iterations of the loop. If there is a bug in the specification or the program, it is likely to be discovered while trying to establish these conditions, without entering the heavy process of finding and verifying a loop invariant. Similarly, partial evaluation can be performed on a program where some parameters are instantiated on suitable test values and then conditions can be generated for the resulting program, possibly leading to the early discovery of a bug.

Memory model. To tackle *all* aspects of the ML and C languages—including data structures with pointers, possible aliasing, pointer arithmetic, etc.—a low level memory model has to be designed (two models, actually, since ML and C obviously differ on this point). Following the Krakatoa approach, the memory model is designed entirely on the logical side and handled in programs through a global reference on the current state of the heap. (Stack variables, however, are still be represented by local references.) Then the language constructs can be interpreted in this new framework, by mere syntactic sugar. The whole technology—typing with effects, weakest preconditions, functional interpretation—is unchanged. A model covering (almost) all ANSI C is work in progress; the translation of C programs reuses ideas from the *C Intermediate Language* [13].

Realistic integer and floating point arithmetic. Currently, integers are modeled using arbitrary precision arithmetic provided by the provers (type `Z` in Coq, `int` in PVS and `num` in HOL Light). Though satisfactory in many cases, it does not reflect the program semantics: integer overflow has to be considered when seriously verifying some critical code. Clearly, there are two main solutions. The first one is to precisely model the machine behavior and its overflows. This is for instance the approach of B. Jacobs [18] in the context of the Loop project [24]. The second solution is to add the necessary conditions to exclude overflows. Then arbitrary precision arithmetic could still be used on the prover side. This second solution seems reasonable since few programs may actually rely on integer overflow to behave correctly. However, we plan to implement both approaches in Why.

Similarly, floating point numbers are currently modeled using the axiomatization of $\mathbb{R}$ provided by the provers (type `R` in Coq and `real` in PVS and HOL Light). Existing axiomatizations of the IEEE 754 floating point arithmetic (by J. Harrison in HOL Light [15] and by L. Théry in Coq [21]) could be used as models. Though really more difficult, it is a challenging goal to verify a floating point algorithm without neglecting the roundings.

Module system. Why is already implementing some parts of the ML language and of its type system. It seems natural to pursue in this way and to incorporate other ML features. The ML module system is of particular interest. X. Leroy *modular module system* [20] could be nicely adapted to our types with effect and specification. This would lift the notion of modularity, currently at the level of functions, to the level of modules. Moreover, this would permit a greater abstraction at the level module, where the interface could show a model actually quite far from the implementation. This is similar to what *Extended ML* is providing for the *applicative* part of SML [23].

Acknowledgments. I am grateful to the Krakatoa authors, C. Marché, C. Paulin and X. Urbain, for many stimulating discussions. I also thank the very first Why users, S. Boulmé, M. Lévy, S. Ranise and L. Théry, for their precious feedback.

References

1. The Coq Proof Assistant. `http://coq.inria.fr/`.
2. The Mizar project. `http://mizar.uwb.edu.pl/`.

3. The Objective Caml language. `http://caml.inria.fr/`.
4. The PVS Specification and Verification System. `http://pvs.csl.sri.com/`.
5. The Simplify decision procedure (part of ESC/Java). `http://research.compaq.com/SRC/esc/simplify/`.
6. The Why verification tool. `http://why.lri.fr/`.
7. Claude Marché, Christine Paulin and Xavier Urbain. The Krakatoa Tool for JML/Java Program Certification. Submitted to JLAP. `http://www.lri.fr/~marche/krakatoa/`.
8. D. Déharbe and S. Ranise. BDD-Driven First-Order Satisfiability Procedures. Technical Report 4630, INRIA, November 2002.
9. J.-C. Filliâtre. A theory of monads parameterized by effects. Research Report 1367, LRI, Université Paris Sud, November 1999.
10. J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. Thèse de doctorat, Université Paris-Sud, July 1999.
11. J.-C. Filliâtre. Formal Proof of a Program: Find. *Science of Computer Programming*, 2001. To appear.
12. J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, July 2003. English translation of [10].
13. S. P. Rahul George C. Necula, Scott McPeak and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Conference on Compiler Construction (CC'02)*, Grenoble, March 2002.
14. John Harrison. HOL Light. `http://www.cl.cam.ac.uk/users/jrh/hol-light/`.
15. John Harrison. A Machine-Checked Theory of Floating Point Arithmetic. In *International Conference on Theorem Proving in Higher Order Logics*, LNCS, pages 113–130, Nice, France, 1999. Springer-Verlag.
16. C. A. R. Hoare. Proof of a program: *Find. Communications of the ACM*, 14(1):39–45, January 1971. Also in [17] pages 59–74.
17. C. A. R. Hoare and C. B. Jones. *Essays in Computing Science*. Prentice Hall, 1989.
18. B. Jacobs. Java's Integral Types in PVS. Manuscript. `http://www.cs.kun.nl/~bart/PAPERS/integral.ps.Z`.
19. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Iowa State University, 2000.
20. Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3), 2000.
21. Laurence Rideau Marc Daumas and Laurent Théry. A Generic Library for Floating-Point Numbers and its Application to Exact Computing. In *International Conference on Theorem Proving in Higher Order Logics*, volume 2152 of *LNCS*, pages 169–184, 2001.
22. Silvio Ranise and David Déharbe. The haRVey decision procedure. `http://www.loria.fr/~ranise/haRVey/`.
23. D. Sannella S. Kahrs and A. Tarlecki. The definition of Extended ML: a gentle introduction. *Theoretical Computer Science*, 173:445–484, 1997.
24. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi (eds.), editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS*, volume 2031 of *LNCS*, pages 299–312. Springer-Verlag, 2001.