TYPES summer school 2007

Why: an Intermediate Language for Program Verification

Jean-Christophe Filliâtre

CNRS – Université Paris Sud

Contents

These notes are an introduction to the Why tool. This tool implements a programming language designed for the verification of sequential programs. This is an intermediate language to which existing programming languages can be compiled and from which verification conditions can be computed.

Section 1 introduces the theory behind the Why tool (syntax, typing, semantics and weakest preconditions for its language). Then Section 2 illustrates the practical use of the tool on several examples.

Chapter 1

Underlying Theory

Implementing a verification condition generator (VCG) for a realistic programming language such as C is a lot of work. Each construct requires a specific treatment and there are many of them. Though, almost all rules will end up to be instances of the five historical Hoare Logic rules [?]. Reducing the VCG to a core language thus seems a good approach. Similarly, if one has written a VCG for C and has to write another one for Java, there are clearly enough similarities to hope for this core language to be reused. Last, if one has to experiment with several logics, models and/or proof tools, this core language should ideally remain the same.

The Why tool implements such an intermediate language for VCGs, that we call HL in the following (for *Hoare Language*). Syntax, typing, semantics and weakest preconditions calculus are given below, but we first start with a tour of HL features.

Genericity. HL annotations are written in a first-order predicate syntax but are not interpreted at all. This means that HL is independent of the underlying logic in which the annotations are interpreted. The WP calculus only requires the logic to be minimal i.e. to include universal quantification, conjunction and implication.

ML syntax. HL has an ML-like syntax where there is no distinction between expressions and statements. This greatly simplifies the language—not only the syntax but also the typing and semantics. However HL has few in common with the ML family languages (functions are not first-class values, there is no type inference, etc.)

Aliases. HL is an alias-free language. This is ensured by the type checking rules. Being alias free is crucial for reasoning about programs, since the rule for assignment

$$\{P[x \leftarrow E]\}\, x \texttt{ := } E\, \{P\}$$

implicitly assumes that any variable other than x is left unmodified. Note however that the absence of alias in HL does not prevent the interpretation of programs with possible aliases: such programs can be interpreted using a more or less complex memory model made of several unaliased variables (see Section **??**).

Exceptions. Beside conditional and loop, HL only has a third kind of control statement, namely exceptions. Exceptions can be thrown from any program point and caught anywhere upper in the control-flow. Arbitrary many exceptions can be declared and they may carry values. Exceptions can be used to model exceptions from the source language (e.g. Java's exceptions) but also to model all kinds of abrupt statements (e.g. C and Java's `return`, `break` or `continue`).

Typing with effects. HL has a typing with effects: each expression is given a type together with the sets of possibly accessed and possibly modified variables and the set of possibly raised exceptions. Beside its use for the alias check, this is the key to modularity: one can declare and use a function without implementing it, since its type mentions its side-effects. In particular, the WP rule for function call is absolutely trivial.

Auxiliary variables. The usual way to relate the values of variables at several program points is to used the so-called *auxiliary variables.* These are variables only appearing in annotations and implicitly universally quantified over the whole Hoare triple. Though auxiliary variables can be given a formal meaning [?] their use is cumbersome in practice: they pollute the annotations and introduce unnecessary equality reasoning on the prover side. Instead we propose the use of program *labels*—similar to those used for `goto`s—to refer to the values of variables at specific program points. This appears to be a great improvement over auxiliary variables, without loss of expressivity.

1.1 Syntax

1.1.1 Types and Specifications

Program annotations are written using the following minimal first-order logic:

$$\begin{array}{rcl} t & ::= & c \mid x \mid \texttt{!}x \mid \phi(t,\dots,t) \mid \texttt{old}(t) \mid \texttt{at}(t,L) \\ p & ::= & P(t,\dots,t) \mid \forall x:\beta.p \mid p \Rightarrow p \mid p \wedge p \mid \dots \end{array}$$

A term t can be a constant c, a variable x, the contents of a reference x (written $\texttt{!}x$) or the application of a function symbol ϕ. It is important to notice that ϕ is a function symbol belonging to the logic: it is not defined in the program. The construct $\texttt{old}(t)$ denotes the value of term t in the precondition state (only meaningful within the corresponding postcondition) and the construct $\texttt{at}(t, L)$ denotes the value of the term t at the program point L (only meaningful within the scope of a label L).

We assume the existence of a set of *pure types* (β) in the logical world, containing at least a type `unit` with a single value `void` and a type `bool` for booleans with two values `true` and `false`.

Predicates necessarily include conjunction, implication and universal quantification as they are involved in the weakest precondition calculus. In practice, one is likely to add at least disjunction, existential quantification, negation and true and false predicates. An atomic predicate is the application of a predicate symbol P and is not interpreted. For the forthcoming WP calculus, it is also convenient to introduce an `if-then-else` predicate:

$$\begin{array}{l} \texttt{if}\ t\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \equiv \\ \quad (t = \texttt{true} \Rightarrow p_1) \wedge (t = \texttt{false} \Rightarrow p_2) \end{array}$$

Program types and specifications are classified as follows:

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

A value of type τ is either an immutable variable of a pure type (β), a reference containing a value of a pure type (β ref) or a function of type $(x : \tau) \rightarrow \{p\}\,\beta\ \epsilon\,\{q\}$ mapping the formal parameter x to the specification of its body, that is a precondition p, the type τ 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 $\{p\}\,\beta\ \epsilon\,\{q\}$ has no precondition and no postcondition (both being `true`) and no effect (ϵ is made of three empty lists) it can be shortened to τ. In particular, $(x_1 : \tau_1) \rightarrow \cdots \rightarrow (x_n : \tau_n) \rightarrow \kappa$ 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.

1.1.2 Expressions

The syntax for program expressions is given in Figure 1.1. 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. The syntax mostly follows ML's one. `ref` e introduces a new reference initialized with e. `loop` e $\{$`invariant` p `variant` $t\}$ is an infinite loop of body e, invariant p and which termination is ensured by the variant t. The `raise` construct is annotated with a type τ (there is no type inference in HL). There are two ways to insert proof obligations in programs: `assert` $\{p\}$; e places an assertion p to be checked right before e and e $\{q\}$ places a postcondition q to be checked right after e.

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

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

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

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

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:

```
while $e_1$ do $e_2$ {invariant $p$ variant $t$} ≡
  try
    loop if $e_1$ then $e_2$ else raise $Exit$
    {invariant $p$ variant $t$}
  with $Exit$ _ -> void end
```

$$
\begin{array}{lcl}
t & ::= & c \mid x \mid \texttt{!}x \mid \phi(t,\dots,t) \\
e & ::= & t \\
 & \mid & x \texttt{ := } e \\
 & \mid & \texttt{let } x \texttt{ = } e \texttt{ in } e \\
 & \mid & \texttt{let } x \texttt{ = ref } e \texttt{ in } e \\
 & \mid & \texttt{if } e \texttt{ then } e \texttt{ else } e \\
 & \mid & \texttt{loop } e \ \{\texttt{invariant } p \texttt{ variant } t\} \\
 & \mid & L\!:\!e \\
 & \mid & \texttt{raise } (E\ e) : \tau \\
 & \mid & \texttt{try } e \texttt{ with } E\ x \rightarrow e \texttt{ end} \\
 & \mid & \texttt{assert } \{p\};\ e \\
 & \mid & e\ \{q\} \\
 & \mid & \texttt{fun } (x:\tau) \rightarrow \{p\}\ e \\
 & \mid & \texttt{rec } x\ (x:\tau)\dots(x:\tau) : \beta\ \{\texttt{variant } t\} = \{p\}\ e \\
 & \mid & e\ e
\end{array}
$$

Figure 1.1: Syntax

1.1.3 Functions and Programs

A program (p) is a list of declarations. A declaration (d) is either a definition introduced with `let` or a declaration introduced with `val`, or an exception declaration.

$$
\begin{array}{lcl}
p & ::= & \emptyset \mid d\ p \\
d & ::= & \texttt{let } x \texttt{ = } e \\
 & \mid & \texttt{val } x : \tau \\
 & \mid & \texttt{exception } E \texttt{ of } \beta
\end{array}
$$

1.2 Typing

This section introduces typing and semantics for HL.

Typing environments contain bindings from variables to types of values, exceptions declarations and labels:

$$\Gamma \ ::= \ \emptyset \mid x:\tau,\Gamma \mid \texttt{exception } E \texttt{ of } \beta,\Gamma \mid \texttt{label } L,\Gamma$$

The type of a constant or a function symbol is given by the operator *Typeof*. A type τ is said to be *pure*, and we write τ `pure`, if it is not a reference type. We write $x \in \tau$ whenever the reference x appears in type τ i.e. in any annotation or effect within τ.

An effect is composed of three sets of identifiers. When there is no ambiguity we write (r, w, e) for the effect `reads` r `writes` w `raises` e. Effects compose a natural semi-lattice of bottom element $\bot = (\emptyset, \emptyset, \emptyset)$ and supremum $(r_1, w_1, e_1) \sqcup (r_2, w_2, e_2) =$

$(r_1 \cup r_2, w_1 \cup w_2, e_1 \cup e_2)$. We also define the erasing of the identifier x in effect $\epsilon = (r, w, e)$ as $\epsilon \backslash x = (r \backslash \{x\}, w \backslash \{x\}, e \backslash \{x\})$.

We introduce the typing judgment $\Gamma \vdash e : (\tau, \epsilon)$ with the following meaning: in environment Γ the expression e has type τ and effect ϵ. Typing rules are given in Figure 1.2. They assume the definitions of the following extra judgments:

- $\Gamma \vdash \kappa \; \texttt{wf}$: the specification κ is well formed in environment Γ,
- $\Gamma \vdash p \; \texttt{wf}$: the precondition p is well formed in environment Γ,
- $\Gamma \vdash q \; \texttt{wf}$: the postcondition q is well formed in environment Γ,
- $\Gamma \vdash t : \beta$: the logical term t has type β in environment Γ.

The purpose of this typing with effects is two-fold. First, it rejects aliases: it is not possible to bind one reference variable to another reference, neither using a `let in` construct, nor a function application. Second, it will be used when interpreting programs in Type Theory (in Section 1.5 below).

1.3 Semantics

We give a big-step operational semantics to HL. The notions of values and states are the following:

$$\begin{array}{rcl} v & ::= & c \mid E\ c \mid \texttt{rec}\ f\ x = e \\ s & ::= & \{(x,c), \dots, (x,c)\} \end{array}$$

A value v is either a constant value (integer, boolean, etc.), an exception E carrying a value c or a closure $\texttt{rec}\ f\ x = e$ representing a possibly recursive function f binding x to e. For the purpose of the semantic rules, it is convenient to add the notion of closure to the set of expressions:

$$e \ ::= \ \dots \mid \texttt{rec}\ f\ x = e$$

In order to factor out all semantic rules dealing with uncaught exceptions, we introduce the following set of contexts R:

$$\begin{array}{rcl} R & ::= & [] \mid x \texttt{ := } R \mid \texttt{let}\ x \texttt{ = } R \texttt{ in } e \mid \texttt{let}\ x \texttt{ = ref } R \texttt{ in } e \\ & \mid & \texttt{if } R \texttt{ then } e \texttt{ else } e \mid \texttt{loop } R \ \{\texttt{invariant } p \texttt{ variant } t\} \\ & \mid & \texttt{raise } (E\ R) : \tau \mid R\ e \end{array}$$

The semantics rules are given Figure 1.3.

1.4 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 $r = E_1 \Rightarrow q_1; \dots; E_n \Rightarrow q_n$ 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 $wp(e, \mathsf{True})$.

$$\frac{\mathit{Typeof}(c) = \beta}{\Gamma \vdash c : (\beta, \bot)} \qquad \frac{x : \tau \in \Gamma \qquad \tau\ \texttt{pure}}{\Gamma \vdash x : (\tau, \bot)} \qquad \frac{x : \beta\ \mathsf{ref} \in \Gamma}{\Gamma \vdash !x : (\beta, \texttt{reads}\ x)}$$

$$\frac{\Gamma \vdash t_i : (\beta_i, \epsilon_i) \quad \mathit{Typeof}(\phi) = \beta_1, \ldots, \beta_n \rightarrow \beta}{\Gamma \vdash \phi(t_1, \ldots, t_n) : (\beta, \bigsqcup_i \epsilon_i)}$$

$$\frac{x : \beta\ \mathsf{ref} \in \Gamma \qquad \Gamma \vdash e : (\beta, \epsilon)}{\Gamma \vdash x \texttt{ := } e : (\texttt{unit}, (\texttt{writes}\ x) \sqcup \epsilon)}$$

$$\frac{\Gamma \vdash e_1 : (\tau_1, \epsilon_1) \qquad \tau_1\ \texttt{pure} \qquad \Gamma, x : \tau_1 \vdash e_2 : (\tau_2, \epsilon_2)}{\Gamma \vdash \texttt{let}\ x \texttt{ = } e_1\ \texttt{in}\ e_2 : (\tau_2, \epsilon_1 \sqcup \epsilon_2)}$$

$$\frac{\Gamma \vdash e_1 : (\beta_1, \epsilon_1) \qquad \Gamma, x : \beta_1\ \mathsf{ref} \vdash e_2 : (\tau_2, \epsilon_2) \qquad x \notin \tau_2}{\Gamma \vdash \texttt{let}\ x \texttt{ = ref } e_1\ \texttt{in}\ e_2 : (\tau_2, \epsilon_1 \sqcup \epsilon_2 \backslash x)}$$

$$\frac{\Gamma \vdash e_1 : (\texttt{bool}, \epsilon_1) \qquad \Gamma \vdash e_2 : (\tau, \epsilon_2) \qquad \Gamma \vdash e_3 : (\tau, \epsilon_3)}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : (\tau, \epsilon_1 \sqcup \epsilon_2 \sqcup \epsilon_3)}$$

$$\frac{\Gamma \vdash e : (\texttt{unit}, \epsilon) \qquad \Gamma \vdash p\ \texttt{wf} \qquad \Gamma \vdash t : \texttt{int}}{\Gamma \vdash \texttt{loop}\ e\ \{\texttt{invariant}\ p\ \texttt{variant}\ t\} : (\texttt{unit}, \epsilon)}$$

$$\frac{\Gamma, \texttt{label}\ L \vdash e : (\tau, \epsilon)}{\Gamma \vdash L{:}e : (\tau, \epsilon)}$$

$$\frac{\texttt{exception}\ E\ \texttt{of}\ \beta \in \Gamma \qquad \Gamma \vdash e : (\beta, \epsilon)}{\Gamma \vdash \texttt{raise}\ (E\ e) : \tau : (\tau, (\texttt{raises}\ E) \sqcup \epsilon))}$$

$$\frac{\texttt{exception}\ E\ \texttt{of}\ \beta \in \Gamma \qquad \Gamma \vdash e_1 : (\tau, \epsilon_1) \qquad \Gamma, x : \beta \vdash e_2 : (\tau, \epsilon_2)}{\Gamma \vdash \texttt{try}\ e_1\ \texttt{with}\ E\ x \rightarrow e_2\ \texttt{end} : (\tau, \epsilon_1 \backslash \{\texttt{raises}\ E\} \sqcup \epsilon_2)}$$

$$\frac{\Gamma \vdash p\ \texttt{wf} \qquad \Gamma \vdash e : (\tau, \epsilon)}{\Gamma \vdash \texttt{assert}\ \{p\};\ e : (\tau, \epsilon)} \qquad \frac{\Gamma \vdash e : (\tau, \epsilon) \qquad \Gamma, \mathit{result} : \tau \vdash q\ \texttt{wf}}{\Gamma \vdash e\ \{q\} : (\tau, \epsilon)}$$

$$\frac{\Gamma, x : \tau \vdash p\ \texttt{wf} \qquad \Gamma, x : \tau \vdash e\ \{q\} : (\tau', \epsilon)}{\Gamma \vdash \texttt{fun}\ (x : \tau) \rightarrow \{p\}\ e\ \{q\} : ((x : \tau) \rightarrow \{p\}\, \tau'\ \epsilon\, \{q\}, \bot)}$$

$$\frac{\begin{array}{c} \Gamma' \equiv \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \qquad \Gamma' \vdash p\ \texttt{wf} \qquad \Gamma' \vdash t : \texttt{int} \\ \Gamma', f : (x_1 : \tau_1) \rightarrow \cdots (x_n : \tau_n) \rightarrow \{p\}\, \tau\ \epsilon\, \{q\} \vdash e\ \{q\} : (\tau, \epsilon) \end{array}}{\begin{array}{l} \Gamma \vdash \ \texttt{rec}\ f\ (x_1 : \tau_1) \ldots (x_n : \tau_n) : \tau\ \{\texttt{variant}\ t\} = \{p\}\ e\ \{q\} \\ \qquad : ((x_1 : \tau_1) \rightarrow \cdots (x_n : \tau_n) \rightarrow \{p\}\, \tau\ \epsilon\, \{q\}, \bot) \end{array}}$$

$$\frac{\Gamma \vdash e_1 : ((x : \tau_2) \rightarrow \{p\}\, \tau_2\ \epsilon\, \{q\}, \epsilon_1) \qquad \Gamma \vdash e_2 : (\tau_2, \epsilon_2)) \qquad \tau_2\ \texttt{pure}}{\Gamma \vdash e_1\ e_2 : (\tau, \epsilon_1 \sqcup \epsilon_2 \sqcup \epsilon)}$$

$$\frac{\Gamma \vdash e_1 : ((x : \beta\ \mathsf{ref}) \rightarrow \{p\}\, \tau_2\ \epsilon\, \{q\}, \epsilon_1) \qquad x_2 : \beta\ \mathsf{ref} \in \Gamma \qquad x_2 \notin \tau_2}{\Gamma \vdash e_1\ x_2 : (\tau[x \leftarrow x_2], \epsilon_1 \sqcup \epsilon[x \leftarrow x_2])}$$

Figure 1.2: Typing

$$\frac{}{s, c \longrightarrow s, c} \qquad \frac{}{s, \texttt{!}x \longrightarrow s, s(x)} \qquad \frac{s, t_i \longrightarrow s, c_i}{s, \phi(t_1, \ldots, t_n) \longrightarrow s, \phi(c_1, \ldots, c_n)}$$

$$\frac{s, e \longrightarrow s', E\ c}{s, R[e] \longrightarrow s', E\ c} \qquad \frac{s, e \longrightarrow s', c}{s, x \texttt{ := } e \longrightarrow s' \oplus \{x \mapsto c\}, \texttt{void}}$$

$$\frac{s, e_1 \longrightarrow s_1, v_1 \quad v_1 \text{ not exc.} \quad s_1, e_2[x \leftarrow v_1] \longrightarrow s_2, v_2}{s, \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \longrightarrow s_2, v_2}$$

$$\frac{s, e_1 \longrightarrow s_1, c_1 \quad s_1 \oplus \{x \mapsto c_1\}, e_2 \longrightarrow s_2, v_2}{s, \texttt{let } x \texttt{ = ref } e_1 \texttt{ in } e_2 \longrightarrow s_2, v_2}$$

$$\frac{s, e_1 \longrightarrow s_1, \texttt{true} \quad s_1, e_2 \longrightarrow s_2, v_2}{s, \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \longrightarrow s_2, v_2} \qquad \frac{s, e_1 \longrightarrow s_1, \texttt{false} \quad s_1, e_3 \longrightarrow s_3, v_3}{s, \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \longrightarrow s_3, v_3}$$

$$\frac{s, e \longrightarrow s', \texttt{void} \quad s', \texttt{loop } e\ \{\texttt{invariant } p \texttt{ variant } t\} \longrightarrow s'', v}{s, \texttt{loop } e\ \{\texttt{invariant } p \texttt{ variant } t\} \longrightarrow s'', v}$$

$$\frac{s, e \longrightarrow s', v}{s, L\texttt{:}e \longrightarrow s', v} \qquad \frac{s, e \longrightarrow s', c}{s, \texttt{raise } (E\ e) : \tau \longrightarrow s', E\ c}$$

$$\frac{s, e_1 \longrightarrow s_1, E'\ c \quad E' \neq E}{s, \texttt{try } e_1 \texttt{ with } E\ x \rightarrow e_2 \texttt{ end} \longrightarrow s_1, E'\ c}$$

$$\frac{s, e_1 \longrightarrow s_1, E\ c \quad s_1, e_2[x \leftarrow c] \longrightarrow s_2, v_2}{s, \texttt{try } e_1 \texttt{ with } E\ x \rightarrow e_2 \texttt{ end} \longrightarrow s_2, v_2} \qquad \frac{s, e_1 \longrightarrow s_1, v_1 \quad v_1 \text{ not exc.}}{s, \texttt{try } e_1 \texttt{ with } E\ x \rightarrow e_2 \texttt{ end} \longrightarrow s_1, v_1}$$

$$\frac{s, e \longrightarrow s', v}{s, \{p\}\ e \longrightarrow s', v} \qquad \frac{s, e \longrightarrow s', v}{s, e\ \{q\} \longrightarrow s', v}$$

$$\frac{}{s, \texttt{fun } (x : \tau) \rightarrow \{p\}\ e \longrightarrow s, \texttt{rec } \_\ x = e}$$

$$\frac{}{\begin{array}{c} s, \texttt{rec } f\ (x_1 : \tau_1) \ldots (x_n : \tau_n) : \tau\ \{\texttt{variant } t\} = \{p\}\ e \longrightarrow \\ s, \texttt{rec } f\ x_1 = \texttt{rec } \_\ x2 = \ldots \texttt{rec } \_\ x_n = e \end{array}}$$

$$\frac{s, e_1 \longrightarrow s_1, \texttt{rec } f\ x = e \quad s_1, e_2 \longrightarrow s_2, v_2 \quad s_2, e[f \leftarrow \texttt{rec } f\ x = e, x \leftarrow v_2] \longrightarrow s_3, v}{e_1\ e_2 \longrightarrow s_3, v}$$

Figure 1.3: Semantics

The rules for the basic constructs are the following:

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

On the traditional constructs of Hoare logic, these rules simplify to the well known identities. For instance, the case of the assignment of a side-effect free expression gives

$$wp(x\ \texttt{:=}\ t, q) = q[x \leftarrow t]$$

and the case of a (exception free) sequence gives

$$wp(e_1\texttt{;}\ e_2, q) = wp(e_1, wp(e_2, q))$$

The cases of exceptions and annotations are also straightforward:

$$\begin{array}{rl}
wp(\texttt{raise}\ (E\ e) : \tau, q; r) & = wp(e, r(E); r) \\
wp(\texttt{try}\ e_1\ \texttt{with}\ E\ x \rightarrow e_2\ \texttt{end}, q; r) & = wp(e_1, q; E \Rightarrow wp(e_2, q; r)[x \leftarrow \mathit{result}]; r) \\
wp(\texttt{assert}\ \{p\};\ e, q; r) & = p \wedge wp(e, q; r) \\
wp(e\ \{q'; r'\}, q; r) & = wp(e, q' \wedge q; r' \wedge r)
\end{array}$$

The case of an infinite loop is more subtle:

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

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:

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

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:

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

$$\begin{array}{l}
wp(\texttt{rec}\ f\ (x_1 : \tau_1) \ldots (x_n : \tau_n) : \tau\ \{\texttt{variant}\ t\} = \{p\}\ e, q; r) \\
= q\ \wedge\ \forall x_1 \ldots \forall x_n.\forall\rho.p \Rightarrow wp(L\!:\!e, \mathsf{True})
\end{array}$$

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 $(x_1 : \tau_1) \rightarrow \cdots \rightarrow (x_n : \tau_n) \rightarrow \{p \wedge t < \mathtt{at}(t, L)\}\, \tau\ \epsilon\, \{q\}$ 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:

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

Then assuming that x_1 has type $(x : \tau) \rightarrow \{p'\}\, \tau'\ \epsilon\, \{q'\}$, 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 an abuse for the conjunction of all implications for each postcondition part.

1.5 Interpretation in Type Theory

Expressing program correctness using weakest preconditions is error-prone. Another approach consists in interpreting programs in Type Theory [**?**, **?**] in such a way that if the interpretation can be typed then the initial imperative program is correct. It can be shown that the resulting set of proof obligations is equivalent to the weakest precondition.

The purpose of these notes is not to detail this methodology, only to introduce the language implemented in the Why tool.

Chapter 2

The Why Tool in Practice

The Why tool implements the programming language presented in the previous section. It takes annotated programs as input and generates proof obligations for a wide set of proof assistants (Coq, PVS, Isabelle/HOL, HOL 4, HOL-Light, Mizar) and decision procedures (Simplify, Ergo, Yices, CVC Lite, CVC3, haRVey, Zenon). The Why can be seen from several angles:

1. as a tool to verify *algorithms* rather than *programs*, since it implements a rather abstract and idealistic programming language. Several non-trivial algorithms have already been verified using the Why tool, such as the Knuth-Morris-Pratt string searching algorithm for instance.

2. as a tool to compute weakest preconditions, to be used as an intermediate step in the verification of existing programming languages. It has already been successfully applied to the verification of C and Java programs (see the tools Caduceus and Krakatoa, at `http://caduceus.lri.fr/` and `http://krakatoa.lri.fr/` respectively).

3. as a tool to write axiomatizations and goals and to dispatch them to several existing provers.

To remain independent of the back-end prover that will be used (it may even be several of them), the Why tool makes no assumption regarding the logic used. It uses a *syntax* of polymorphic first-order predicate logic for annotations with no particular interpretation (apart from the usual connectives). Function symbols and predicates can be declared in order to be used in annotations, together with axioms, and they may be given definitions on the prover side later, if needed.

2.1 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 `old`(x).

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 (result: Z),
  forall (Post2: result = (min r n)),
  result <= 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.
```

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

void flag(color 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 arrays:

```
type color_array

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

Again, this is an immutable type, so it comes with a purely applicative signature (`upd` is returning a *new* 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[1].

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 point within the array `t` 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.

[1]The Why tool actually provides a datatype of arrays, exactly in the way we are doing it here, and even a nice syntax for array operations.

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 *declare* a predicate `permutation` as follows:

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

The intended meaning of `permutation`(t_1, t_2, i, j) is "the multi-sets of elements in $t_1[i..j]$ and $t_2[i..j]$ are the same". Since the program is only performing transpositions in the array, the most convenient way to axiomatize `permutation` is to say that it is an equivalence relation containing transpositions:

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

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)
```

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:

```
  ...
  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
                 permutation(t,t@init,0,n-1)
       variant r - i }
  ...
```

We can now proceed to the verification of the program, which causes no difficulty (most proof obligations are even discharged automatically by Simplify).