TYPES summer school 2005

Program Verification using Coq
Introduction to the WHY tool

Jean-Christophe Filliâtre

CNRS – Université Paris Sud

Contents

Introduction

These lecture notes present some techniques to verify programs correctness using the Coq proof assistant.

In Chapter 1, we show how to use Coq directly to verify purely functional programs, using the Coq extraction mechanism which produces correct ML programs out of constructive proofs. This chapter describes techniques to define and prove correct functions in Coq, focusing on issues such as partial functions, non-structurally recursive functions, or the use of dependent types to specify functions. The interested reader will find much more material related to this subject in Y. Bertot and P. Castéran's book dedicated to Coq [3].

In Chapter 2, we introduce the Why tool for the verification of imperative programs. The Why tool is actually not specifically related to Coq, since it offers a wide range of other back-end provers (PVS, Isabelle/HOL, Simplify, etc.). But Coq plays a particular role since Why can interpret imperative programs as purely functional programs in Coq, providing increased trust in the verification process and making a clear connection between the two chapters of these notes. The interested reader will find more material related to Why on its web site `http://why.lri.fr/`, including a reference manual, many examples and links to related tools for the verification of C and Java programs.

Chapter 1

Verification of purely functional programs

In this chapter, we focus on the specification and verification of purely functional programs. We show how the `Coq` proof assistant can be used to produce correct ML code. This chapter is illustrated using the case study of balanced binary search trees, which constitutes an example of purely functional program simultaneously useful and complex.

In the following, we call *informative* what lies in the sort `Set` and *logic* what lies in the sort `Prop`. This sort distinction is exploited by the `Coq` *extraction* mechanism [19, 20, 17, 18]. This tool extracts the informative contents of a `Coq` term as an ML program. The logical parts disappear (or subsist as a degenerated value with no associated computation). The theoretical foundations of program extraction can be found in the references above.

1.1 Immediate method

The most immediate way to verify a purely functional program consists in defining it as a `Coq` function and then to prove some properties of this function. Indeed, most (purely functional) ML programs can be written in the Calculus of Inductive Constructions.

Generally speaking, we first define in `Coq` a "pure" function, that is with a purely informative type a la ML (a type from system F). Let us assume here a function we only one argument:

$$f \quad : \quad \tau_1 \rightarrow \tau_2$$

We show that this function realizes a given specification $S : \tau_1 \rightarrow \tau_2 \rightarrow$ `Prop` with a theorem of the shape

$$\forall x.\ (S\ x\ (f\ x))$$

The proof is conducted following the definition of f.

Example. We use a finite sets library based on balanced binary trees as a running example. We first introduce a datatype of binary trees containing integers

```
Inductive tree : Set :=
| Empty
| Node : tree -> Z -> tree -> tree.
```

and a membership relation In stating that an element occurs in a tree (independently of any insertion choice):

```
Inductive In (x:Z) : tree -> Prop :=
| In_left : forall l r y, (In x l) -> (In x (Node l y r))
| In_right : forall l r y, (In x r) -> (In x (Node l y r))
| Is_root : forall l r, (In x (Node l x r)).
```

A function testing for the empty set can be defined as

```
Definition is_empty (s:tree) : bool := match s with
| Empty => true
| _ => false end.
```

and its correctness proof is stated as

```
Theorem is_empty_correct :
  forall s, (is_empty s)=true <-> (forall x, ~(In x s)).
```

The proof follows the definition of `is_empty` and is only three lines long:

```
Proof.
  destruct s; simpl; intuition.
  inversion_clear H0.
  elim H with z; auto.
Qed.
```

Let us consider now the membership test within a binary search tree. We first assume an ordering relation over integers:

```
Inductive order : Set := Lt | Eq | Gt.
Hypothesis compare : Z -> Z -> order.
```

Then we define a function `mem` looking for an element in a tree which is *assumed* to be a binary search tree:

```
Fixpoint mem (x:Z) (s:tree) {struct s} : bool := match s with
| Empty =>
   false
| Node l y r => match compare x y with
   | Lt => mem x l
   | Eq => true
   | Gt => mem x r
  end
end.
```

The correctness proof of this function requires the definition of a binary search tree, as the following `bst` predicate:

```
Inductive bst : tree -> Prop :=
| bst_empty :
   (bst Empty)
| bst_node :
   forall x (l r : tree),
   bst l -> bst r ->
   (forall y, In y l -> y < x) ->
   (forall y, In y r -> x < y) -> bst (Node l x r).
```

Now the correctness of the `mem` function can be stated:

```
Theorem mem_correct :
  forall x s, (bst s) -> (mem x s=true <-> In x s).
```

We see on this example that the specification S has the particular shape $P\ x \rightarrow Q\ x\ (f\ x)$. P is called a *precondition* and Q a *postcondition*.

Modularity. When trying to prove `mem_correct` we start with `induction s; simpl` to follow `mem`'s definition. The first case (`Empty`) is trivial. On the second one (`Node s1 z s2`) we bump into the term `match compare x z with ...` and it is not possible to go further. Indeed, we know nothing about the `compare` function used in `mem`. We have to specify it first, using for instance the axiom

```
Hypothesis compare_spec :
  forall x y, match compare x y with
  | Lt => x<y
  | Eq => x=y
  | Gt => x>y
  end.
```

Then we can use this specification in the following way:

```
generalize (compare_spec x z); destruct (compare x z).
```

The proof is completed without difficulty.

Note. For purely informative functions such as `is_empty` or `mem`, the extracted program is identical to the `Coq` term. As an example, the command `Extraction mem` gives the following ocaml code:

```
let rec mem x = function
  | Empty -> false
  | Node (l, y, r) ->
    (match compare x y with
       | Lt -> mem x l
       | Eq -> true
       | Gt -> mem x r)
```

1.1.1 The case of partial functions

A first difficulty occurs when the function is partial, i.e. has a Coq type of the shape

$$f : \forall x : \tau_1.\ (P\ x) \rightarrow \tau_2$$

The typical case is a division function expecting a non-zero divisor.

In our running example, we may want to define a function min_elt returning the smallest element of a set which is *assumed* to be non-empty (that is the leftmost element in the binary search tree). We can give this function the following type:

$$\texttt{min\_elt} : \forall s : \texttt{tree}.\ \neg s = \texttt{Empty} \rightarrow \texttt{Z} \tag{1.1}$$

where the precondition is $\neg s = \texttt{Empty}$. The specification of min_elt can be stated as

$$\forall s.\ \forall h : \neg s = \texttt{Empty}.\ \texttt{bst}\ s \rightarrow \texttt{In}\ (\texttt{min\_elt}\ s\ h)\ s \wedge \forall x.\ \texttt{In}\ x\ s \rightarrow \texttt{min\_elt}\ s\ h \leq x$$

with the same precondition as the function itself (hypothesis h) together with the other precondition bst s. The hypothesis h is mandatory to be able to apply min_elt. We see that using a partial function in Coq is not easy: one has to pass proofs as arguments, and proof terms may be difficult to build.

Even the definition of a partial function may be difficult. Let us write a function min_elt with type (1.1). The ML code we have in mind is:

```
let rec min_elt = function
  | Empty -> assert false
  | Node (Empty, x, _) -> x
  | Node (l, _, _) -> min_elt l
```

Unfortunately, the Coq definition is more difficult. First, the assert false in the first case of the pattern matching corresponds to an absurd case (we assumed a non-empty tree). The Coq definition expresses this absurdity using the False_rec elimination applied to a proof of False. So one has to build such a proof from the precondition. Similarly, the third case of the pattern matching makes a recursive call to min_elt and thus we have to build a proof that l is non-empty. Here it is a consequence of the pattern matching which has already eliminated the case where l is Empty. In both cases, the necessity to build these proof terms complicates the pattern matching, which must be dependent. We get the following definition:

```
Fixpoint min_elt (s:tree) (h:~s=Empty) { struct s } : Z :=
  match s return ~s=Empty -> Z with
  | Empty =>
    (fun h => False_rec _ (h (refl_equal Empty)))
  | Node l x _ =>
    (fun h => match l as a return a=l -> Z with
              | Empty => (fun _ => x)
              | _ => (fun h => min_elt l
                                (Node_not_empty _ _ _ _ h))
              end (refl_equal l))
  end h.
```

The first proof (argument of False_rec) is built directly. The second one uses the following lemma:

```
Lemma Node_not_empty : forall l x r s, Node l x r=s -> ~s=Empty.
```

We can now prove the correctness of min_elt:

```
Theorem min_elt_correct :
  forall s (h:~s=Empty), bst s ->
    In (min_elt s h) s /\
    forall x, In x s -> min_elt s h <= x.
```

Once again, the proof is conducted following the definition of the function and does not contain any difficulty.

One can check that the extracted code is indeed the one we had in mind. Extraction min_elt outputs:

```
let rec min_elt = function
  | Empty -> assert false (* absurd case *)
  | Node (l, x, t) ->
      (match l with
         | Empty -> x
         | Node (t0, z0, t1) -> min_elt l)
```

There are several points of interest here. First, the use of False_rec is extracted to assert false, which is precisely the expected behavior (we proved that this program point was not reachable, so it is legitimate to say that reaching it is absurd i.e. a "proof" of false). Second, we see that the logical arguments that were complicating the definition have disappeared in the extracted code (since they were in sort Prop).

Another solution is to define the min_elt function using a proof rather than a definition. It is then easier to build the proof terms (using the interactive proof editor). Here the definition-proof is rather simple:

```
Definition min_elt : forall s, ~s=Empty -> Z.
Proof.
  induction s; intro h.
  elim h; auto.
  destruct s1.
  exact z.
  apply IHs1; discriminate.
Defined.
```

But it is more difficult to be convinced that we are building the right function (as long as we haven't proved its correctness). In particular, one has to use the automatic tactics such as auto with great care, since it could build a function different from the one we have in mind. One the example above, auto is only used on a logical goal (Empty=Empty).

One way to get convinced that the underlying code is the right one is to have a look at the extracted code. Here we get exactly the same as before.

The refine tactic helps in defining partial functions (but not only). It allows the user to give an incomplete proof term, some parts being omitted (when denoted by _) and turned into subgoals. We can redefine the min_elt function using the refine tactic as follows:

```
Definition min_elt : forall s, ~s=Empty -> Z.
refine
  (fix min (s:tree) (h:~s=Empty) { struct s } : Z :=
  match s return ~s=Empty -> Z with
  | Empty =>
      (fun h => _)
  | Node l x _ =>
      (fun h => match l as a return a=l -> Z with
                | Empty => (fun _ => x)
                | _ => (fun h => min_elt l _)
                end _)
  end h).
```

We get two subgoals that are easy to discharge. However, we notice that a dependent matching is still required.

A last solution consists in making the function total, by completing it in an arbitrary way out of its definition domain. Here we may choose to return the value 0 when the set is empty. This way we avoid the logical argument $\neg s = \texttt{Empty}$ and all its nasty consequences. The type of min_elt is back to tree $\rightarrow$ Z and its definition quite simple:

```
Fixpoint min_elt (s:tree) : Z := match s with
  | Empty => 0
  | Node Empty z _ => z
  | Node l _ _ => min_elt l
end.
```

The correctness theorem is still the same, however:

```
Theorem min_elt_correct :
  forall s, ~s=Empty -> bst s ->
    In (min_elt s) s /\
    forall x, In x s -> min_elt s <= x.
```

The correctness statement still has the precondition $\neg s = \texttt{Empty}$, otherwise it would not be possible to ensure $In\ (\texttt{min\_elt}\ s)\ s$.

Note. The division function Zdiv over integers is defined this way as a total function but its properties are only provided under the assumption that the divisor is non-zero.

Note. Another way to make the function min_elt total would be to make it return a value of type option Z, that is None when the set is empty and Some m when a smallest element m exists. But then the underlying code is slightly different (and so is the correctness statement).

1.1.2 Functions that are not structurally recursive

Issues related to the definition and the use of a partial function are similar to the ones encountered when defining and proving correct a recursive function which is not structurally recursive.

Indeed, one way to define such a function is to use a principle of well-founded induction, such as

```
well_founded_induction
   : forall (A : Set) (R : A -> A -> Prop),
     well_founded R ->
     forall P : A -> Set,
     (forall x : A, (forall y : A, R y x -> P y) -> P x) ->
     forall a : A, P a
```

But then the definition requires to build proofs of `R y x` for each recursive call on `y`; we are faced to the same problems, but also to the same solutions, mentioned in the section above.

Let us assume we want to define a function `subset` checking for set inclusion on our binary search trees. A possible ML code is the following:

```
let rec subset s1 s2 = match (s1, s2) with
  | Empty, _ ->
      true
  | _, Empty ->
      false
  | Node (l1, v1, r1), (Node (l2, v2, r2) as t2) ->
      let c = compare v1 v2 in
      if c = 0 then
        subset l1 l2 && subset r1 r2
      else if c < 0 then
        subset (Node (l1, v1, Empty)) l2 && subset r1 t2
      else
        subset (Node (Empty, v1, r1)) r2 && subset l1 t2
```

We see that recursive calls are performed on trees that are not always strict sub-terms of the initial arguments (not mentioning the additional difficulty of a simultaneous recursion on two arguments). Though there exists a simple termination criterion, that is the total number of elements in the two trees.

Thus we first establish a well-founded induction principle over two trees based on the sum of their cardinalities:

```
Fixpoint cardinal_tree (s:tree) : nat := match s with
  | Empty => O
  | Node l _ r => (S (plus (cardinal_tree l) (cardinal_tree r)))
end.

Lemma cardinal_rec2 :
```

```
forall (P:tree->tree->Set),
(forall (x x':tree),
   (forall (y y':tree),
     (lt (plus (cardinal_tree y) (cardinal_tree y'))
         (plus (cardinal_tree x) (cardinal_tree x'))) -> (P y y'))
   -> (P x x')) ->
forall (x x':tree), (P x x').
```

The proof is simple: we first manage to reuse a well-founded induction principle over type `nat` provided in the Coq library, namely `well_founded_induction_type_2`, and then we prove that the relation is well-founded since it is of the shape `lt` $(f\ y\ y')$ $(f\ x\ x')$ and because `lt` itself is a well-founded relation over `nat` (another result from the Coq library):

```
Proof.
  intros P H x x'.
  apply well_founded_induction_type_2 with
    (R:=fun (yy' xx':tree*tree) =>
        (lt (plus (cardinal_tree (fst yy')) (cardinal_tree (snd yy')))
            (plus (cardinal_tree (fst xx')) (cardinal_tree (snd xx')))));
    auto.
    apply (Wf_nat.well_founded_ltof _
            (fun (xx':tree*tree) =>
              (plus (cardinal_tree (fst xx')) (cardinal_tree (snd xx'))))).
Save.
```

We are now in position to define the `subset` function with a definition-proof using the `refine` tactic:

```
Definition subset : tree -> tree -> bool.
Proof.
```

First we apply the induction principle `cardinal_rec2`:

```
  intros s1 s2; pattern s1, s2; apply cardinal_rec2.
```

Then we match on `x` and `x'`, both `Empty` cases being trivial:

```
  destruct x.
  (* x=Empty *)
  intros; exact true.
  (* x = Node x1 z x2 *)
  destruct x'.
  (* x'=Empty *)
  intros; exact false.
```

Next we proceed by case on the result of `compare z z0`:

```
  (* x' = Node x'1 z0 x'2 *)
  intros; case (compare z z0).
```

In each of the three cases, the recursive calls (hypothesis `H`) are handled using the `refine` tactic. We get a proof obligation expressing the decreasing of the total number of elements, which is automatically discharged by `simpl; omega` (`simpl` is required to unfold the definition of `cardinal_tree`):

```
    (* z < z0 *)
    refine (andb (H (Node x1 z Empty) x'2 _)
                 (H x2 (Node x'1 z0 x'2) _)); simpl; omega.
    (* z = z0 *)
    refine (andb (H x1 x'1 _) (H x2 x'2 _)); simpl ; omega.
    (* z > z0 *)
    refine (andb (H (Node Empty z x2) x'2 _)
                 (H x1 (Node x'1 z0 x'2) _)); simpl ; omega.
Defined.
```

Note. We could have used a single `refine` for the whole definition.

Note. It is interesting to have a look at the extracted code for a function defined using a principle such as `well_founded_induction`. We can first have a look at the extracted code for `well_founded_induction` and we recognize a fixed point operator:

```
  let rec well_founded_induction x a =
    x a (fun y _ -> well_founded_induction x y)
```

When unfolding this operator and two other constants

```
  Extraction NoInline andb.
  Extraction Inline cardinal_rec2 Acc_iter_2 well_founded_induction_type_2.
  Extraction subset.
```

we get exactly the expected ML code:

```
  let rec subset x x' =
    match x with
      | Empty -> True
      | Node (x1, z0, x2) ->
          (match x' with
             | Empty -> False
             | Node (x'1, z1, x'2) ->
                 (match compare z0 z1 with
                    | Lt ->
                        andb (subset (Node (x1, z0, Empty)) x'2)
                          (subset x2 (Node (x'1, z1, x'2)))
                    | Eq -> andb (subset x1 x'1) (subset x2 x'2)
                    | Gt ->
                        andb (subset (Node (Empty, z0, x2)) x'2)
                          (subset x1 (Node (x'1, z1, x'2)))))
```

Many other techniques to define functions that are not structurally recursive are described in the chapter 15 of *Interactive Theorem Proving and Program Development* [3].

1.2 The use of dependent types

Another approach to program verification in Coq consists in using the richness of the type system to express the specification of the function within its type. Actually, a type *is* a specification. In the case of ML, it is only a very poor specification (e.g. a function expects an integer and returns an integer) but in Coq one can express that a function is expecting a non-negative integer and returning a prime integer:

$$f : \{n : Z \mid n \geq 0\} \to \{p : Z \mid \texttt{prime}\ p\}$$

We are going to show how to do this in this section.

1.2.1 The subtype type sig

The Coq notation $\{x : A \mid P\}$ denotes the "subtype of A of values satisfying the property P" or, in a set-theoretical setting, the "subset of A of elements satisfying P". The notation $\{x : A \mid P\}$ actually stands for the application sig A (fun $x \Rightarrow P$) where sig is the following inductive type:

```
Inductive sig (A : Set) (P : A -> Prop) : Set :=
  exist : forall x:A, P x -> sig P
```

This inductive type is similar to the existential ex, apart from its sort which is Set instead of Prop (we aim at defining a function and thus its arguments and result must be informative).

In practice, we need to relate the argument and the result within a postcondition Q and thus we prefer the more general specification:

$$f : \forall\ (x : \tau_1),\ P\ x \to \{y : \tau_2 \mid Q\ x\ y\}$$

If we come back to the min_elt function, its specification can be the following:

```
Definition min_elt :
  forall s, ~s=Empty -> bst s ->
    { m:Z | In m s /\ forall x, In x s -> m <= x }.
```

We still have the definition issues mentioned in the previous section and thus we usually adopt a definition by proof (still with the same caution w.r.t. automatic tactics).

Note. The move of the property bst s from the postcondition to the precondition is not mandatory; it is only more natural.

Note. The extraction of sig A Q forgets the logical annotation Q and thus reduces to the extraction of type A. Said otherwise, the sig type can disappear at extraction time. And indeed we have

```
Coq < Extraction sig.
type 'a sig = 'a
  (* singleton inductive, whose constructor was exist *)
```

1.2.2 Variants of sig

We can introduce other types similar to `sig`. For instance, if we want to define a function returning two integers, such as an Euclidean division function, it seems natural to combine two instances of `sig` as we would do with two existentials `ex`:

$$\mathtt{div} : \forall a\ b,\ b > 0 \rightarrow \{q \mid \{r \mid a = bq + r \wedge 0 \leq r < b\}\}$$

But the second instance of `sig` has sort `Set` and not `Prop`, which makes this statement ill-typed. `Coq` introduces for this purpose a variant of `sig`, `sigS` :

```
Inductive sigS (A : Set) (P : A -> Set) : Set :=
  existS : forall x:A, P x -> sig P
```

where the sole difference is the sort of `P` (`Set` instead of `Prop`). `sigS` A (`fun` $x \Rightarrow P$) is written $\{x : A\ \&\ P\}$, which allows to write

$$\mathtt{div} : \forall a\ b,\ b > 0 \rightarrow \{q\ \&\ \{r \mid a = bq + r \wedge 0 \leq r < b\}\}$$

The extraction of `sigS` is naturally a pair:

```
Coq < Extraction sigS.
type ('a, 'p) sigS =
  | ExistS of 'a * 'p
```

Similarly, if we want a specification looking like

$$\{x : A \mid P\ x \wedge Q\ x\}$$

there exists and inductive "made on purpose", `sig2`, defined as

```
Inductive sig2 (A : Set) (P : A -> Prop) (Q : A -> Prop) : Set :=
  exist2 : forall x : A, P x -> Q x -> sig2 P Q
```

Its extraction is identical to the one of `sig`.

1.2.3 Specification of a boolean function: sumbool

A very common kind of specification is the one of a boolean function. In this case, we want to specify what are the two properties holding when the function is returning `false` and `true` respectively. `Coq` introduces an inductive type for this purpose, `sumbool`, defined as

```
Inductive sumbool (A : Prop) (B : Prop) : Set :=
  | left : A -> sumbool A B
  | right : B -> sumbool A B
```

It is a type similar to bool but each constructor contains a proof, of A and B respectively. sumbool A B is written {A}+{B}. A function checking for the empty set can be specified as follows:

$$\texttt{is\_empty} : \forall s,\ \{s = \texttt{Empty}\} + \{\neg s = \texttt{Empty}\}$$

A more general case, very common in practice, is the one of a decidable equality. Indeed, if a type A is equipped with an equality $\texttt{eq} : \texttt{A} \rightarrow \texttt{A} \rightarrow \texttt{Prop}$, we can specify a function deciding this equality as

$$\texttt{A\_eq\_dec} : \forall x\ y,\ \{\texttt{eq}\ x\ y\} + \{\neg(\texttt{eq}\ x\ y)\}$$

It is exactly as stating

$$\forall x\ y,\ (\texttt{eq}\ x\ y) \vee \neg(\texttt{eq}\ x\ y)$$

apart from the sort which is different. In the latter case, we have a disjunction in sort Prop (an excluded-middle instance for the predicate eq) whereas in the former case we have a "disjunction" in sort Set, that is a program deciding the equality.

The extraction of sumbool is a type isomorphic to bool:

```
Coq < Extraction sumbool.
type sumbool =
  | Left
  | Right
```

In practice, one can tell the Coq extraction to use ML booleans directly instead of Left and Right (which allows the ML if-then-else to be used in the extracted code).

Variant sumor

There exists a variant of sumbool where the sorts are not the same on both sides:

```
Inductive sumor (A : Set) (B : Prop) : Set :=
  | inleft : A -> A + {B}
  | inright : B -> A + {B}
```

This inductive type can be used to specify an ML function returning a value of type α option: the constructor inright stands for the case None and adds a proof of property B, and the constructor inleft stands for the case Some and adds a proof of property A. The extraction of type sumor is isomorphic to the ML type option:

```
Coq < Extraction sumor.
type 'a sumor =
  | Inleft of 'a
  | Inright
```

We can combine sumor and sig to specify the min_elt in the following way:

```
Definition min_elt :
  forall s, bst s ->
  { m:Z | In m s /\ forall x, In x s -> m <= x } + { s=Empty }.
```

It corresponds to the ML function turned total using an `option` type.

We can even combine `sumor` and `sumbool` to specify our ternary `compare` function:

```
Hypothesis compare : forall x y, {x<y} + {x=y} + {x>y}.
```

Note that now this single hypothesis replaces the inductive `order` and the two hypotheses `compare` and `compare_spec`.

Let us go back to the membership function on binary search trees, `mem`. We can now specify it using a dependent type:

```
Definition mem :
  forall x s, bst s -> { In x s }+{ ~(In x s) }.
```

The definition-proof starts with an induction over `s`.

```
Proof.
  induction s; intros.
  (* s = Empty *)
  right; intro h; inversion_clear h.
```

The case `s=Empty` is trivial. In the case `s=Node s1 z s2`, we need to proceed by case on the result of `compare x z`. It is now simpler than with the previous method: no more need to call for the `compare_spec` lemma, since `compare x z` contains its specification in its type.

```
  (* s = Node s1 z s2 *)
  case (compare x z); intro.
```

Similarly, each induction hypothesis (over `s1` and `s2`) is a function containing its specification in its type. We use it, when needed, by applying the `case` tactic. The remaining of the proof is easy.

Note. It is still possible to obtain the pure function as a projection of the function specified using a dependent type:

```
Definition mem_bool x s (h:bst s) := match mem x s h with
  | left _ => true
  | right _ => false
  end.
```

Then it is easy to show the correctness of this pure function (since the proof is "contained" in the type of the initial function):

```
Theorem mem_bool_correct :
  forall x s, forall (h:bst s),
  (mem_bool x s h)=true <-> In x s.
Proof.
  intros.
  unfold mem_bool; simpl; case (mem x s h); intuition.
  discriminate H.
Qed.
```

But this projection has few interests in practice.

Note. It is important to notice that each function is now given its specification when it is defined: it is no more possible to establish several properties of a same function as it was with a pure function.

1.3 Modules and functors

The adequacy of Coq as formalism to specify and prove correct purely functional ML programs extends to the module system. Indeed, Coq is equipped with a module system inspired by the module system of Objective Caml [16, 6, 7] since version 8. As Coq function types can enrich ML types with logical annotations, Coq modules can enrich ML ones.

For instance, if we want to write our finite sets library as a functor taking an arbitrary type as argument (and no more `Z` only as it was the case up to now) equipped with a total order, we start by defining a signature for this functor argument. It packs together a type `t`, an equality `eq` and an order relation `lt` over this type:

```
Module Type OrderedType.
  Parameter t : Set.
  Parameter eq : t -> t -> Prop.
  Parameter lt : t -> t -> Prop.
```

together as a decidability result for `lt` and `eq`:

```
  Parameter compare : forall x y, {lt x y}+{eq x y}+{lt y x}.
```

We also have to include some properties of `eq` (equivalence relation) and `lt` (order relation not compatible with `eq`) without which the functions over binary search trees would not be correct:

```
  Axiom eq_refl : forall x, (eq x x).
  Axiom eq_sym : forall x y, (eq x y) -> (eq y x).
  Axiom eq_trans : forall x y z, (eq x y) -> (eq y z) -> (eq x z).

  Axiom lt_trans : forall x y z, (lt x y) -> (lt y z) -> (lt x z).
  Axiom lt_not_eq : forall x y, (lt x y) -> ~(eq x y).
```

Last, we can add some `Hint` commands for the `auto` tactic to this signature, so that they will be automatically available within the functor body:

```
  Hint Immediate eq_sym.
  Hint Resolve eq_refl eq_trans lt_not_eq lt_trans.
End OrderedType.
```

Then we can write our finite sets library as a functor taking an argument `X` of type `OrderedType` as argument:

```
Module ABR (X: OrderedType).

  Inductive tree : Set :=
  | Empty
```

```
| Node : tree -> X.t -> tree -> tree.

Fixpoint mem (x:X.t) (s:tree) {struct s} : bool := ...

Inductive In (x:X.t) : tree -> Prop := ...
Hint Constructors In.

Inductive bst : tree -> Prop :=
| bst_empty : (bst Empty)
| bst_node :
   forall x (l r : tree),
   bst l -> bst r ->
   (forall y, In y l -> X.lt y x) ->
   (forall y, In y r -> X.lt x y) -> bst (Node l x r).

(* etc. *)
```

Note. The Objective Caml language provides a finite sets library based on balanced binary search trees (AVLs [2]), as a functor taking an ordered type as argument. This library implements all usual operations over sets (union, intersection, difference, cardinal, smallest element, etc.), iterators (`map`, `fold`, `iter`) and even a total order over sets allowing the construction of sets of sets by a second application of the same functor (and so on). This library has been verified using Coq by Pierre Letouzey and Jean-Christophe Filliâtre [11]. This proof exhibited a balancing bug in some functions; the code was fixed in ocaml 3.07 (and the fix verified with Coq).

Chapter 2

Verification of imperative programs: the Why tool

This chapter is 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 2.1 introduces the theory behind the Why tool (syntax, typing, semantics and weakest preconditions for its language). Then Section 2.2 illustrates the practical use of the tool on several examples and quickly describes the application of the Why tool to the verification of C and Java programs.

2.1 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 [12]. 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 polymorphism, 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 2.2.3).

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 [21] 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.

2.1.1 Syntax

Types and specifications

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

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

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 $\mathtt{at}(t, L)$ 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:

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

Program types and specifications are classified as follows:

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

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.

Expressions

The syntax for program expressions is given in Figure 2.1. 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 since there is no polymorphism 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.

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

Figure 2.1: Syntax

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

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

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:

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

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.

2.1.2 Typing

This section introduces typing and semantics for HL.

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

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 $\texttt{pure}$, 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 $\texttt{reads}\ r\ \texttt{writes}\ w\ \texttt{raises}\ e$. 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 2.2. 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 2.1.5 below).

2.1.3 Semantics

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

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

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:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 2.2: Typing

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 2.3.

2.1.4 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 $r = E_1 \Rightarrow q_1; \ldots; E_n \Rightarrow q_n$ 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:

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

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:

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

The case of an infinite loop is more subtle:

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 2.3: Semantics

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

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

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

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

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

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

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.

2.1.5 Interpretation in Type Theory

Expressing program correctness using weakest preconditions is error-prone. Another approach consists in interpreting programs in Type Theory [9, 10] 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.

2.2 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, haRVey, CVC Lite). The Why can be seen from two 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 (as briefly sketched in the next section 2.2.3).

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 first-order predicates 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, but they will be given meaning on the prover side.

2.2.1 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 [1], 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.
```

2.2.2 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(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 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.

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
```

[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.

For the permutation property, we only *declare* a predicate that will be defined on the prover side, whatever the prover is:

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

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:

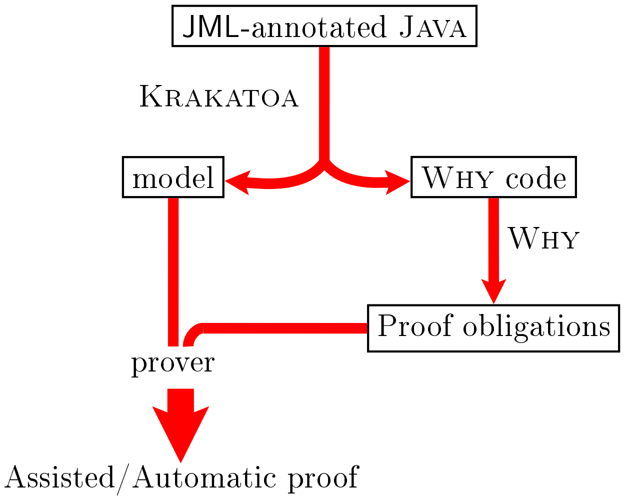

Figure 2.4: Verifying Java programs using Krakatoa and Why

```
...
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).

2.2.3 Application to the verification of C and Java programs

The Why tool is applied to the verification of C and Java programs, as the back-end of two open-source tools CADUCEUS [14] and KRAKATOA [8] respectively. Both tools are based on the same kind of model, following Bornat [4], and handle almost all ANSI C and all sequential Java respectively. As far as KRAKATOA is concerned, Java programs are annotated using the Java Modeling Language (JML) [15] and thus KRAKATOA is very similar to tools like LOOP [22] or JACK [5]. An overview of the KRAKATOA-Why combination is given Figure 2.4. The combination with CADUCEUS is very similar.

Bibliography

[1] The Simplify decision procedure (part of ESC/Java). `http://research.compaq.com/SRC/esc/simplify/`.

[2] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics–Doklady*, 3(5):1259–1263, September 1962.

[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science. An EATCS Series. Springer Verlag, 2004. `http://www.labri.fr/Perso/~casteran/CoqArt/index.html`.

[4] Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[5] Lilian Burdy and Antoine Requet. Jack: Java Applet Correctness Kit. In *Gemplus Developers Conference GDC'2002*, 2002. See also `http://www.gemplus.com/smart/r_d/trends/jack.html`.

[6] Jacek Chrzaszcz. Implementing modules in the system Coq. In *16th International Conference on Theorem Proving in Higher Order Logics*, University of Rome III, September 2003.

[7] Jacek Chrzaszcz. *Modules in Type Theory with generative definitions*. PhD thesis, Warsaw University and Université Paris-Sud, 2003. To be defended.

[8] Claude Marché, Christine Paulin and Xavier Urbain. The Krakatoa Tool for JML/Java Program Certification. Submitted to JLAP. `http://www.lri.fr/~marche/krakatoa/`.

[9] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. Thèse de doctorat, Université Paris-Sud, July 1999.

[10] 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 [9].

[11] Jean-Christophe Filliâtre and Pierre Letouzey. Functors for Proofs and Programs. In *Proceedings of The European Symposium on Programming*, Barcelona, Spain, March 29-April 2 2004. Voir aussi `http://www.lri.fr/~filliatr/fsets/`.

[12] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969. Also in [13] pages 45–58.

[13] C. A. R. Hoare and C. B. Jones. *Essays in Computing Science*. Prentice Hall, 1989.

[14] Jean-Christophe Filliâtre and Claude Marché. The Caduceus tool for the verification of C programs. `http://why.lri.fr/caduceus/`.

[15] 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.

[16] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[17] Pierre Letouzey. A New Extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[18] Pierre Letouzey. *Programmation fonctionnelle certifiée en Coq*. PhD thesis, Université Paris Sud, 2003. To be defended.

[19] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In Association for Computing Machinery, editor, *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989.

[20] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. PhD thesis, Université Paris 7, January 1989.

[21] T. Schreiber. Auxiliary Variables and Recursive Procedures. In *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 697–711. Springer-Verlag, April 1997.

[22] 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.