Who: A Verifier for Effectful Higher-Order Programs

Johannes Kanig
INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893
LRI, Univ Paris-Sud, CNRS, Orsay, F-91405
kanig@lri.fr

Jean-Christophe Filliâtre
CNRS, LRI, Univ Paris-Sud, Orsay F-91405
INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893
filliatr@lri.fr

Abstract

We present Who, a tool for verifying effectful higher-order functions. It features *Effect polymorphism*, higher-order logic and the possibility to reason about state in the logic, which enable highly modular specifications of generic code. Several small examples and a larger case study demonstrate its usefulness. The Who tool is intended to be used as an intermediate language for verification tools targeting ML-like programming languages.

Categories and Subject Descriptors F3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Verification

Keywords Hoare Logic, Higher-Order Programs

1. Introduction and Motivation

Higher-order functions are a key feature, not only of purely functional languages like Haskell and the pure fragment of ML, but also in combination with imperative features, like references. The presence and frequent use of higher-order functions such as `Array.iter` in Ocaml prove this importance. However, reasoning about higher-order functions is difficult, and especially so in the presence of effectful computations. Any function's specification depends on its arguments. In a higher-order setting, arguments may be functions that have their own specification. In consequence, to be modular, the specification of a higher-order function can only be formulated *depending on the specifications of its functional arguments*. In a setting where side effects are possible, one also needs to formulate the specification *depending on the side effects of the functional arguments*. We claim that to be truly modular, the specification of a higher-order function must be generic with respect to the specification *and* the effect of its arguments.

We present the tool *Who*[1], an intermediate language aimed at verification of effectful higher-order programs. It features effect polymorphism, a higher-order logic to conveniently formulate specifications as well as special objects in the logic which represent different states of parts of the store. This permits for natural specifications and proof obligations.

The Who tool has one fundamental restriction: aliasing of references is excluded. This allows for vastly simpler specifications. We do not consider this a limitation, because Who is intended to be used as an intermediate language, target of a more sophisticated analysis tool. Such a tool could, using some memory model, translate for example Ocaml programs to the Who language. A similar approach has been applied successfully to first-order programs and object oriented programs (Filliâtre and Marché 2007). In the absence of such a tool for Ocaml programs, all the examples in this paper have been proved correct directly using Who.

Section 2 recalls the basic principles of tools based on Hoare logic. This section can be skipped by readers who are already familiar with Hoare logic, perhaps with the exception of section 2.1. Section 3 explains the new concepts introduced by Who with many examples. In section 4, as a more complex case study, the Koda-Ruskey algorithm is considered.

2. A Short Introduction to Hoare Logic

Hoare logic is concerned about correctness of programs[2]. Consider the following simple program which fills an array `ar` with the value `v`, starting from `ofs` for `len` cells[3].

```
let fill ar ofs len v =
  for i = ofs to ofs + len - 1 do
    set ar i v
  done
```

where `set ar i v` sets cell at index `i` of array `ar` to the value `v`. Now Hoare logic permits to write *pre- and postconditions* for such a function, specifying using logical formulas what `fill` does or should do. The postcondition specifies what should be true *after* executing the function. In this case, we want the function to set all the elements between `ofs` and $\texttt{ofs} + \texttt{len} - 1$ to `v`:

```
  ∀(k : int).
    ofs ≤ k ≤ ofs + len - 1 → get ar k = v.
```

For the postcondition to be complete, one should also specify that the array remains unchanged outside of $[\texttt{ofs}..\texttt{ofs} + \texttt{len} - 1]$.

We also need to add a precondition. Indeed, the function `fill` won't work for unsuitable values of `ofs` and `len`. We have to require

```
  0 ≤ ofs ≤ length ar - len ∧ 0 ≤ len.
```

The function is now specified from the outside — we have written down its pre- and postcondition. Using this specification and the rules of Hoare logic, we can prove that every use of `fill` adheres

[1] Who is freely distributed at `http://www.lri.fr/~kanig/files/who-0.1.tgz`.

[2] We only consider partial correctness in this paper. Total correctness requires termination proofs as well.

[3] Most of the code in this paper is taken from the Ocaml standard library and has been only slightly changed.

ML'09, August 30, 2009, Edinburgh, Scotland, UK.
Copyright © 2009 ACM 978-1-60558-509-3/09/08…$5.00

```
let fill (ar : α array) (ofs : int)
         (len : int) (v: α) =
  { 0 ≤ ofs ≤ length ar - len ∧ 0 ≤ len }
  for i = ofs to ofs + len - 1 do
    { ∀(k : int). ofs ≤ k < i → get ar k = v }
    set ar i v
  done
  { ∀(k : int).
      ofs ≤ k ≤ ofs + len - 1 → get ar k = v }
```

Figure 1. A simple first-order program.

to the requirements and does not use more information than what we have specified. However, to be able to prove the correctness of `fill` itself, with respect to the specification we have given, some more annotations are needed. In this case, the function body is a simple `for` loop. If we can state a formula which is initially true, preserved by the loop body, and which, using the upper loop bound, implies the postcondition, we are done. Such a formula is called a *loop invariant*. In this case, the invariant is simply a reformulation of the postcondition:

```
∀(k : int). ofs ≤ k → k < i → get ar k = v
```

For $\texttt{i} = \texttt{ofs} + \texttt{len}$, this clearly implies the postcondition. Also, assuming the precondition, the formula is initially true for $\texttt{i} = \texttt{ofs}$ (because the premise of the implication is false). Fig. 1 summarizes the specification of the function `fill`.

The essence of traditional Hoare logic is what we have shown in this section: every function comes with a pre- and postcondition, loops have loop invariants. Before calling a function (entering a loop), one has to show that one can derive the precondition (the loop invariant) from the current context. Returning from a function call (exiting a loop), one can add the postcondition (the loop invariant) to the current context and proceed. The mechanical work of traversing the program code and generating *proof obligations* can be automated. Typically, one starts from the postcondition, traverses the code in reverse order and establish a necessary precondition; this is called a *weakest precondition calculus*.

There are many tools which implement Hoare logic in the described or a similar way. Examples are the Why platform (Filliâtre 2003) and the Boogie system (Barnett et al. 2005).

2.1 Functional Programs

The mechanism we have shown in the previous section leaves some open questions when it comes to higher-order functions. It is unclear how to deal with the following code, for example:

```
let apply f x = f x
```

We can't give a precondition to `apply` because we don't know yet what function it is applied to. Similarly, we don't know what postcondition we can guarantee after calling `f`.

Intuitively, however, the situation is clear: `apply` should have the same precondition and postcondition as its argument `f`. In the Pangolin system (Régis-Gianas and Pottier 2008) this can be expressed like this:

```
let apply f x =
  { pre f x }
  f x
  { r : post f x r }
```

Here, we state that the precondition of `apply` is the precondition of `f` applied to `x` and that *the result* `r` of `apply` does verify the postcondition of `f` applied to `x`. One can see that in Pangolin, `f` is treated quite differently in the logic and in the program. Indeed, types of functional objects are translated as follows. The function type `t1 → t2` becomes a tuple type in the logic:

```
(t1 → prop) × (t1 → t2 → prop)
```

A (program) function becomes a pair of predicates, precisely its pre- and postcondition. This serves two purposes. First, this makes it impossible to *call* the function in the logic, avoiding issues with effectful and nonterminating functions. Second, it gives access to the specification of the function. The terms `pre` and `post` above are simply other names for `fst` and `snd`. To make things work out, however, this trick requires a *higher-order logic*, a logic where one can quantify over predicates, store predicates in tuples, etc.

Using this technique, the Pangolin system is capable of reasoning about purely functional higher-order functions as well as common constructs of function programs such as algebraic datatypes and pattern matching. The Pangolin system cannot deal with effectful functions.

2.2 Logic Functions and Axioms

In tools for program verification, one is often confronted with the problem of specifying the behaviour of a program where the most natural description of the behaviour is the program itself! Let us take an example: Suppose we want to prove correct the following program, computing the maximum of two integer values:

```
let max (a b : int) =
  if a < b then b else a
```

How can one express the specification of the program? One would like to say that it computes the maximum of its arguments, but then one would need to define *maximum*, and this definition would probably look like the body of `max`. The way out is to state the properties one expects from the maximum of two values. At least, one would like to have the two properties that if $a \leq b$, then $max\ a\ b = b$ and similarly if $b \leq a$. From here, there are now two possibilities: one can either accept `max` as the definition of the maximum function and prove that it verifies the two desired properties. Or one can declare a *logical function max* with two integer arguments, postulate the two desired properties as axioms and define a *program function* `pmax` whose specification states it computes precisely the logical function *max*. In other words, one has to write:

```
logic max : int → int → int
axiom maxright :
  ∀(a b : int) a ≤ b → max a b = b
axiom maxleft :
  ∀(a b : int) a >= b → max a b = a
let pmax (a b : int) =
  if a < b then b else a
  { r : r = max a b }
```

Notice that we have not given a *definition* for `max`, we have only postulated its existence. The other difference between a logical and a program function is that a logical function has to terminate and cannot have any side effects. In the Who system, these are also the only differences. Indeed, one can use logical functions in programs, if it is convenient. Continuing the above example, if one is not interested in the (trivial) correction of `pmax`, one can omit the definition of `pmax` and directly use *max* in the rest of the program. This also helps to avoid the recurring problem of defining everything twice — in the logic and in the program. As a final remark, let us state that this sharing between logic and programs is reminiscent to traditional Hoare logic, where arithmetic and boolean expressions are shared in the logic and in the program.

Modeling Data Structures. We can also apply this mecanism of logical declarations to *model* more complex data structures that are not *a priori* part of the programming language. For example one can define immutable arrays simply by postulating their existence as follows:

```
type α farray

logic get : ∀α. α farray → int → α
logic set :
  ∀α. α farray → int → α → α farray
logic length : ∀α. α farray → int
```

Notice that the type `farray` describes *functional* (i.e., immutable) arrays. The `set` function returns a new array instead of `unit`. One also needs to specify axioms to describe the behaviour of these functions. This so-called *theory of arrays* is standard. From this starting point, one can model imperative arrays by representing them as references to functional arrays. An array update in Ocaml syntax (setting the array `a` at index `i` to `x`)

```
a.(i) ← x
```

becomes

```
a := set !a i x
```

In words: `set` returns a *new* array which has been modified at index i and we set the contents of `a` to this new array.

The examples in this paper using the Who system all use the above model for arrays. In the following, α `array` is thus an abbreviation for a reference to a α `farray`. The reader should keep in mind that Who is an intermediate language and that this kind of transformation could be done automatically. It shall be noted that this particular approach has the drawback that the array length is part of the state of an array. So, according to the model, an array update can change the length of an array. We therefore need an additional axiom which states that update does not change the length of the array. In the following, when dealing with arrays, we will omit annotations that deal with this issue.

3. An Overview of Who

The Who system is an effort to combine the complementary aspects of traditional first-order tools such as Why and the higher-order tool Pangolin. It contains the features we have described before: traditional Hoare style reasoning, logic declarations and axioms as well as reasoning about higher-order functions. In this section, we discuss the modifications present in Who, which are necessary to specify effectful higher order functions.

3.1 Features of Who

Regions. To be able to state properties about references, one needs to *track* them in a special way. It is clear that program identifiers are not suitable to track references, because several identifiers may denote the same physical reference. A standard solution to this problem is the use of *regions* (Tofte and Talpin 1997) to denote references. A region is simply an identifier for a reference cell, but on *type system level*. Reference types are annotated with a region: $\texttt{t ref}_\rho$ denotes the type of references which point to the cell ρ, which contains an object of type `t`. Annotated in this way, reference types become *singleton types*; each such type has only a single inhabitant, precisely the reference cell with that region name. One recovers typing flexibility by adding *region polymorphism*, which permits to use a function at different reference types.

From a specification point of view, regions replace program identifiers when dealing with objects stored in reference cells. However, in most situations (and in all situations in this paper), it is convenient to introduce syntactic sugar which permits use a program variable x instead of a region ρ, if x is of type $\texttt{t ref}_\rho$. This will be done systematically throughout the document.

Effects. A distinguished feature of the Who system is the use of an *effect system*. In Ocaml, the type of `fill` is

$$\alpha\ \texttt{array} \to \texttt{int} \to \texttt{int} \to \alpha \to \texttt{unit},$$

but in Who, its type becomes

$$\forall \texttt{ar}.\ \alpha\ \texttt{farray ref}_{ar} \to^{\emptyset} \texttt{int} \to^{\emptyset} \texttt{int} \to^{\emptyset} \alpha \to^{\{\texttt{ar}\}} \texttt{unit}$$

Every arrow is annotated with an *effect*, *i.e.* the list of regions which might change on function application. Here, only when all arguments are applied, the function has an effect on its first argument `ar`. The quantifier at the beginning binds the region `ar`, making it possible to use `fill` with different arrays.

To take effects into account, we also need to modify the way functions are handled in the logic. First, we introduce a new type for *states*, also called *effect records* in the logic. To every effect expression `e` we attribute an effect type $\langle \texttt{e} \rangle$ which represents the part of the store which contain exactly these reference cells. Similarly to the Pangolin system, a function of type $\texttt{t1} \to^{\{\texttt{e}\}} \texttt{t2}$ is translated to its pre and postcondition

$$(\texttt{t1} \to \langle \texttt{e} \rangle \to \texttt{prop}) \times (\texttt{t1} \to \langle \texttt{e} \rangle \to \langle \texttt{e} \rangle \to \texttt{t2} \to \texttt{prop}).$$

The precondition has an additional argument which corresponds to the state when entering the function. The postcondition has *two* additional arguments representing the states *before* and *after* executing the function. The corresponding arguments corresponding to states contain only the mutable variables which are relevant to the execution of the function. We will see many uses of this mechanism in the following.

Effect polymorphism and the aliasing restriction. So far we have seen that the *specification* of a functional argument, such as `f` in the example of `apply`, can be dealt with generically. But we also need to deal with the *effect* of `f`. Consider the following code:

```
let apply_reset f x =
  r := 0;
  f x
```

where, before calling `f`, some global integer reference `r` is set to 0. Can we claim that `r` contains 0 after execution of the function body? And what can we say about the effect of `f`? The Who system answers the second question as follows. As we don't know the effect of `f`, we introduce an *effect variable* `e` for its effects and go on as before. This variable is generalized at the `let`-binding, just as type variables. The following code in Who shows some new elements of the syntax:

```
let apply_reset [e] (f : α →{e} β) (x : α) =
  { pre f x e}
  r := 0; f x
  { res : r = 0 ∧ post f x e|old e res}
```

The brackets after the function name introduce an effect variable. We can use the effect variable `e` in the effect of `f`. We have to require the precondition of `f` applied to `x` and the effect represented by `e`. The postcondition states that `r` contains 0 and that we have the postcondition of `f` on its own part of the store, relating the old ($\texttt{e}_{|\texttt{old}}$) and the new state (`e`).

The reader is probably puzzled by the syntax of the annotations of the previous paragraph. Let us explain a bit more. The precondition of `apply_reset` as stated above is an abbreviation for the following predicate:

```
(fun (cur : ⟨r e⟩) → pre f x e|cur)
```

The precondition of every function introduces its own name for the current piece of state `cur` which corresponds to its effects (`r` and `e` here). One can refer to the components of this state (access the field of the effect record) by writing the name of the field and a bar |, followed by the name of the effect record. The explicit introduction of the current state `cur` can be omitted, it will be done automatically. Also, there is a notion of *default state*. In the pre- and postconditions, the default state is always `cur`, so fields of this state can simply be referred to by their name. We have made use

```
let for [e]
    (inv : ⟨e⟩ → int → prop)
    (start : int) (end : int) (f : int →{e} unit) =
    { inv cur start ∧
      ∀(i : int). start ≤ i ∧ i ≤ end →
      ∀(m : ⟨e⟩). inv m i → pre f i m ∧
      ∀(n : ⟨e⟩). post f i m n () → inv n (i+1)
      }
    let rec aux (i : int) : int →{e} unit =
      {start ≤ i ∧ i ≤ end + 1 ∧  inv cur i}
      if i ≤ end then
        let () = f i in
        aux (i + 1)
      else ()
      {inv cur (end + 1)}
    in
    if start ≤ end + 1 then aux start else ()
    { inv cur (max start (end + 1))}
```

Figure 2. The `for` loop as a higher-order function.

```
let iter [e] (ar : α farray ref_r)
  (inv : ⟨ar e⟩ → int →  prop)
  (f : α →{e} unit) =
  { inv cur 0 ∧
    ∀(i:int). 0 ≤ i ∧ i < length ar →
    ∀(m : ⟨ar e⟩).
      inv m i → pre f (get i ar|m) e|m ∧
    ∀(n : ⟨ar e⟩).
      post f (get i ar|m) e|m e|n () → inv n (i+1)
  }
  let l = length !ar in
  for index = 0 to l - 1 do
    { inv cur index a }
    f (get index !ar)
  done
  {inv cur (length ar|old) }
```

Figure 3. Annotated code for `Array.iter`.

of similar abbreviations for the postcondition, in which there is an additional state `old`, representing the state before exectuting the current function.

The reader may still wonder how we can claim `r = 0` in the postcondition of `apply_reset`. The reason is the *aliasing restriction* of the Who system, herited from the Why system. Effect variables, introduced at `let`-bindings, are assumed to be disjoint from the other effects of the function. In our example, the effect `e` cannot contain `r`. This is verified when the effect variables are instantiated; invalid instantiations are rejected.

The region mechanism prevents other forms of aliasing as well. First of all, the instantiation of regions is subject to a similar restriction, to avoid aliasing between reference variables. Furthermore, since reference types are annotated with regions, polymorphic containers (lists, arrays) may not contain references to locations. An array of type

$$\texttt{int ref}_r\ \texttt{array}$$

can only contain pointers to the *same* reference cell. Finally, while references to functions are allowed and supported, a stratification of the store similar to (Boudol 2007) prevents circular use of references, such as Landin's knot (Landin 1964). The aliasing restriction is the key to simpler annotations.

To summarize, let us define the syntax of types:

$$\begin{aligned} \texttt{t} ::= &\ \texttt{int} \mid \texttt{bool} \mid \texttt{..} \mid \texttt{t} \rightarrow \texttt{t} \mid \texttt{t} \xrightarrow{\{e\}} \texttt{t} \\ &\mid \texttt{t} \times \texttt{t} \mid \texttt{t ref}_r \end{aligned}$$

where `r` stands for a region name and `e` stands for an effect, i.e. a set of regions and effect variables. Types without the effectful arrow and without reference types can be used in the logic.

From a theoretical point of view, Who is based on a weakest precondition calculus which can deal with effect polymorphism and higher-order functions. The definition and correctness of this calculus are developped in (Kanig and Filliâtre 2009).

3.2 Higher-order Specifications Involving State

We are now in the position to give examples for the specification of effectful higher-order functions. We consider two examples: a `for` loop implemented as a recursive function and the higher-order iterator for arrays.

The* `for` *loop as higher-order function. In a higher-order language, one can implement the first-order `for` loop by a function

$$\texttt{for : int} \rightarrow \texttt{int} \rightarrow \texttt{(int} \rightarrow \texttt{unit)} \rightarrow \texttt{unit,}$$

where the two integer arguments are the start and end value of the for loop, and the function argument corresponds to the body of the for loop and depends on the current value of the counter. While it is better to add the for loop directly to the language for convenience, let us demonstrate how to implement and specify such a function in the Who system.

As `for` is a higher-order function, it must be polymorphic with respect to the effects of its function argument. This is easily achieved by giving it the type

$$\texttt{for : } \forall \texttt{e. int} \rightarrow \texttt{int} \rightarrow \texttt{(int} \xrightarrow{\{e\}} \texttt{unit)} \xrightarrow{\{e\}} \texttt{unit}$$

If we want to be able to use this function in a way similar to the `for` loop in first-order Hoare logic, we also need the notion of invariant. As we can have logical arguments in the language, the simplest solution is just to add the invariant as an argument of the function. In this case, the invariant depends on the counter and the part of the state that is modified by the body of the loop. It thus can be given the type ⟨e⟩ → `int` → `prop`. In summary, the `for` function we want to implement has the following type:

$$\begin{aligned} \texttt{for : } \forall \texttt{e. } (\langle \texttt{e} \rangle \rightarrow \texttt{int} \rightarrow \texttt{prop}) \rightarrow \ \texttt{int} \rightarrow \texttt{int} \rightarrow \\ \texttt{(int} \xrightarrow{\{e\}} \texttt{unit)} \xrightarrow{\{e\}} \texttt{unit} \end{aligned}$$

Figure 2 shows a possible implementation along with its specification. The looping construct is replaced by a recursive function `aux` which tests if the counter is still below the end value, executes the body and continues recursively, if this is the case. Otherwise, it does nothing. Finally, we enter this auxiliary function only when the start value is actually at least as small as the end value.

From a specification point of view, the `aux` function corresponds *exactly* to the body of the for loop. Here we know that i is between the start and the end value (however, we may have exceeded the end value by one, because we test at the entry of the loop, not at the exit), and we know that the invariant is true for the current value of i and the current state, designated by the special variable `cur`. At the exit of the auxiliary function, we know that we have iterated until the end value.

To be able to prove the correctness of `aux`, we need the following premises: When the precondition of `aux` is true and when $i \leq end$ (the *then* branch of the conditional), the precondition of `f` is true as well (we have the right to call `f`). Also we need to prove that once we have called `f`, i.e., we have its postcondition, we can deduce the precondition of `aux` for the next step, where i is incremented by one. All these proof obligations cannot be proved here, because we do not know the invariant, nor do we know the pre- and postconditions of `f`. Therefore, they are stated in the precondition

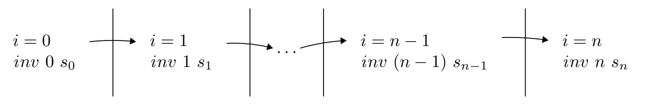


Figure 4. The succession of states during the execution of `Array.iter`.

of `for`. To use `for` with some particular invariant and loop body, this precondition must be valid.

Note that the proposed implementation and specification of `for` is not the only possibility, nor is it the most general one. For convenience, we have required the invariant to be true at least for the initial state. One could imagine a specification where this is only required when the loop body is executed at least once. In this case, the postcondition of `for` would need to express that nothing has changed otherwise.

Iteration over Arrays. Once we have expressed the specification of the `for` higher-order function, we can move on the more complex iteration schemes. For example, the iteration function for arrays[4] is a very often used higher-order function, and it is exclusively used for the side effects it produces. It can be implemented as follows in Ocaml:

```
  let iter f a =
    for i = 0 to length a - 1 do
      f (get a i)
    done
```

where `get a i` denotes the ith element of array `a`.

Figure 3 shows the implementation of this function in Who. In some sense, it is simply a special case of the specification of the `for` loop seen in the previous section. It is a good example of a more interesting manipulation of state in the logic.

First of all, the specification of `iter` is similar to the specification of `for`. We require some kind of "stepping" condition from the function `f` to be iterated on (lines 6 to 9). We also expect an invariant as argument, which has to hold for the starting point of the iteration and which is guaranteed to hold for its final state. However, the pre- and postcondition of `f` now depend on a state which is *smaller* than the overall effect of the loop body. Indeed, `f` only affects `e`, the unknown part of the store, while the overall effect also contains read effects on `ar`, the array we iterate over. Remember that `ar` and `e` are disjoint, by the typing constraints of the language. Therefore, we can reason about the evolution of these two components independently. For example, we state in the precondition of `iter` that the invariant, for any step `i` and some corresponding state `m`, implies the precondition of `f` for the `i`-th cell of the array and the *part of* `m` *designated by* `e` (line 7). Similarly, the postcondition of `f` must ensure that the invariant is preserved (line 9).

Additionally, the invariant may refer to the current contents of `ar`. Therefore `inv` is introduced with type

```
  inv : ⟨ar e⟩ → int →  prop
```

where the first argument is the current state and the second argument is the current step `i` (line 2). Fig. 4 summarizes the succession of states of `Array.iter`.

Let us try to use `iter` to sum all the elements of an integer array. To specify this instance, we need a (logical) function which gives us the sum of an immutable array:

```
logic sum_array : int → int array → int
axiom sum_array_none :
  ∀(t : int array). sum_array 0 t = 0
axiom sum_array_step :
  ∀(t : int array) (i : int).
    0 ≤ i → i < length t →
    sum_array (i+1) t = sum_array i t + get i t
```

[4] `Array.iter` in the Ocaml standard library.

The logical term `sum_array k a` describes the sum of the elements of the array `a` up to (but not including) `k`. This logical description of the problem is the key to the specification; here is the annotated code:

```
parameter sum : int ref_r

let main () =
  {sum = 0}
  iter [{sum}]
    ar
    {{ fun (s : ⟨ar sum⟩) (k:int) →
         sum|s = sum_array k ar|s }}
    (fun (z : int) →
        {} sum := !sum + z { sum = sum|old + z} )
  {sum = sum_array (length ar) ar }
```

Note that the effect and type polymorphic function `iter` has to be instantiated before use; we do not currently support type and effect inference. Concretely, the effect variable is instantiated with the effect of the function argument, `{sum}` here, and the type variable is instantiated with `int`, the content type of the array. This instantiated version is then applied to the array, an invariant, and a function which accumulates the sum of its integer arguments in a reference `sum`. The specification of this function simply states that it increments `sum` by its argument; it doesn't relate `sum` to the contents of the array. The invariant of the iteration is a logical function, introduced by `{{...}}`. It simply states that `sum` contains the sum of the elements of the array up to `k`, using the logical function `sum_array`. As the postcondition of `iter` guarantees the invariant for the last iteration state and the length of the array, we can deduce that `sum` finally contains indeed the sum of the array.

Notice that it is possible to write a variant of `iter` where the function argument is specified to modify the array argument. The specification and proof of this variant are essentially the same as the one we have shown.

3.3 Proof obligations

Specifying programs is but one part of the task of proving program correctness. The two other main tasks are

- generating proof obligations from the specification and
- proving these obligations.

We have seen that Who deals with the part which concerns the generation of proof obligations, thanks to its weakest precondition calculus. It remains the last point, the task of actually proving them. Currently, Who only outputs the proof obligations in Coq syntax. The proofs can then be done inside the proof assistant Coq (The Coq Development Team 2006). If one changes the program or its specification, an already existing proof script is updated to match the new proof obligations.

4. A Complete Case Study

This section presents a realistic and non trivial case study involving an effectful continuation-based functional program. This example is contained in Who's sources.

4.1 Koda-Ruskey's Algorithm

The algorithm considered in this section is due to Y. Koda and F. Ruskey (Koda and Ruskey 1993; Knuth 2001). It enumerates the ideals of certain finite partially ordered sets—namely, those whose Hasse diagram is a forest—as a Gray code. The algorithm can be described in a simple way. The task is to enumerate all *colorings* of

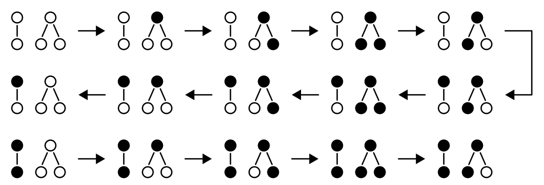
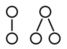


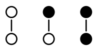
Figure 5. Koda and Ruskey's algorithm applied to the forest (1).

a given, arbitrary forest. A coloring consists in marking every node as either black or white, with the sole constraint that all descendants of a white node be white as well. For instance, the following forest:

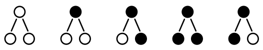

(1)

admits exactly 15 distinct colorings, all of which are given in Figure 5. By definition, a sequence of colorings forms a Gray code if and only if every coloring of the forest appears exactly once in it and two consecutive colorings differ by the color of exactly one node.

Let us illustrate the algorithm's functioning on the forest (1). The main idea is to interleave the sequences of colorings which correspond to each of the trees that form the forest. Here, one must interlace the sequence of the three colorings of the left-hand tree, namely:


(2)

with the sequence of the five colorings of the right-hand tree, given below:


(3)

Thus, the first line of Figure 5 exhibits the first coloring of the left-hand tree, combined successively with all colorings of the right-hand tree. The second line shows the second coloring of the left-hand tree, again combined with all colorings of the right-hand tree, but this time in reverse order—indeed, it is clear that the mirror image of a Gray code remains a Gray code. Lastly, the third line exhibits the third coloring of the left-hand tree and all colorings of the right-hand tree, this time again in their initial order.

There remains to explain how to enumerate all colorings of a tree. Let the first coloring be uniformly white. Then, to obtain the remainder of the sequence, color the root node black and enumerate all colorings of the forest formed by its children. The sequence thus obtained is indeed a Gray code, because (i) the first and second colorings differ only by the color of the root node and (ii) from then on, the root node remains unaffected, and the sequence of the colorings of the children forms a Gray code by construction. This process is illustrated by (2) and (3) above. Note that the coloring where every node is black does not necessarily appear last in a sequence.

4.2 Functional Implementation

We consider an Ocaml implementation of Koda-Ruskey's algorithm which makes use of higher-order functions (Filliâtre and Pottier 2003). First, we introduce the types for trees, forests and colors as follows:

```
type tree  = Node of int × forest
and forest = tree list
type color = White | Black
```

A tree is thus a term Node(i, f) where i is the index of its root and f the forest of its subtrees. A forest is simply a list of trees. The current coloring of the considered forest will be materialized in a

```
let rec enum_forest k = function
  | [] →
      k ()
  | Node (i, f') :: f →
      let k () = enum_forest k f in
      if bits.(i) = White then begin
        k (); bits.(i) ← Black; enum_forest k f'
      end else begin
        enum_forest k f'; bits.(i) ← White; k ()
      end
```

Figure 6. An Ocaml implementation of Koda-Ruskey's algorithm.

global array bits, which is assumed to be large enough to contain all indices of the forest.

A nice way to implement Koda-Ruskey's algorithm is to use a continuation-based approach, using a recursive function enum_forest with the following type:

```
  enum_forest : (unit → unit) → forest → unit
```

It takes a continuation k and a forest f as arguments. Then it enumerates all colorings of f, applying continuation k once for each different coloring of f.

The code for enum_forest is given in Figure 6 and proceeds as follows. If the forest is empty, we simply call the continuation k. Otherwise, the forest contains at least one tree, say Node (i,f'), next to a sub-forest f. We first build a new continuation k which enumerates the colorings of f, using the old continuation k. Then we consider the tree itself. The function must be able to enumerate the colorings in both directions (as explained in the next section). To determine which, we look up the color of the tree's root, that is bits.(i). If it is currently white, then the whole tree must be white. We have a complete coloring, so we signal the continuation k; then, we color the root black and enumerate its children's colorings using enum_forest. If, on the other hand, the root is currently black, we do the converse. That is, we first use enum_forest to enumerate the children's colorings in reverse order, which leaves all of the children entirely white; then, we color the root white, and signal the continuation k.

4.3 Formal Specification

We are now going to give a formal specification to this functional implementation. In particular, we should characterize what is the effect of continuation k. Obviously, it modifies the contents of array bits, since it is precisely used to do so in recursive calls. But k may have other effects, if for instance the initial continuation is used to print the current coloring or to record it in some array[5]. Therefore, we use effect polymorphism to indicate that k may have some effect e, disjoint from bits:

```
enum_forest :
  ∀e. (unit →{bits e} unit) → forest →{bits e} unit
```

To specify the behavior of enum_forest, we must also exhibit the forest whose colorings are enumerated by the continuation, as an additional argument, say f0. Thus the Who implementation of Koda-Ruskey's algorithm has three parameters, and looks like

```
let rec enum_forest
  [e|] (f0 : forest) (k : unit →{bits e} unit)
  (f : forest) = ...
```

The additional argument f0 is *logical*, since it only participates to the specification and not to the computation. As we already claimed, Who is intended to be an intermediate language for pro-

[5] For the purpose of drawing pictures such as the one in Figure 5, enum_forest was even used with a continuation k producing pictures.

```
 1 let rec enum_forest [e] (f0 : forest) (k : unit →{bits e} unit) (f : forest) =
 2   { validf (append f f0) ∧ anyf bits (append f f0) ∧
 3     (∀(s : ⟨bits e⟩).  validf f0 → anyf bits|s f0 → pre k () s) ∧
 4     (∀(s s': ⟨bits e⟩). post k () s s' () → mirrorf bits|s bits|s' f0 ∧ eq_out bits|s bits|s' f0) }
 5   if is_nil f then k ()
 6   else
 7     let k () =
 8       { validf (append (tail f) f0) ∧ anyf bits (append (tail f) f0) }
 9       enum_forest f0 k (tail f)
10       { mirrorf bits|old bits (append (tail f) f0) ∧ eq_out bits|old bits (append (tail f) f0) })
11     in
12     let i = nodi (head f) in
13     let f' = nodf (head f) in
14     if is_white (get !bits i) then
15       (k (); bits := set !bits i Black ; enum_forest (append (tail f) f0) k f')
16     else
17       (enum_forest (append (tail f) f0) k f'; bits := set !bits i White; k ())
18   { mirrorf bits|old bits (append f f0) ∧ eq_out bits|old bits (append f f0) }
```

Figure 7. Who implementation of Koda-Ruskey's algorithm.

gram verification, where programs do not necessarily need to coincide with the original Ocaml programs.

We now turn to the specification itself. Here, we focus on the behavior of `enum_forest` with respect to the current coloring, *i.e.* we characterize the conditions under which the function can be called and its effect on the contents of array `bits`. We do not prove that the set of all colorings form a Gray code, but this could be deduced without too much effort. The Who code for function `enum_forest` is given in Figure 7 and its annotations are detailed in the remaining of this section.

The first requirement is a sanity condition over forests `f` and `f0`, which says that they do not contain duplicate indices. We write $i \in t$ (resp. $i \in f$) when i is an index occurring in tree t (resp. in forest f). We also write $\texttt{valid}\ t$ (resp. $\texttt{valid}\ f$) to characterize a tree t (resp. a forest f) where all indices are different. These two notions of occurrence and validity are easily defined inductively over trees and forests[6]. If `append` denotes the concatenation of forests, we thus require `valid (append f f0)` as a precondition (line 2).

The next requirement is a condition over the current coloring, *i.e.* the state of `bits`, for `enum_forest` to execute correctly. Requiring the nodes to be all colored in white is a too strong condition, since recursive calls are going to be used to "decolor" some trees, as in the second row of Figure 5. We must thus characterize the final coloring of a tree or a forest. Obviously, the parity of the number of colorings plays a role. Indeed, in a forest containing two trees, say t_1 and t_2 in that order, the final coloring of t_2 will be all white if t_1 admits an even number of colorings, and will be itself a final coloring of t_2 otherwise. We introduce the predicates `even` and `odd`, over trees and forests, to indicate an even (resp. odd) number of colorings. They are inductively defined as follows:

$$\frac{\texttt{even}\ t}{\texttt{even}\ (t::f)} \qquad \frac{\texttt{even}\ f}{\texttt{even}\ (t::f)} \qquad \frac{\texttt{odd}\ f}{\texttt{even}\ (\texttt{Node}\ (i,f))}$$

$$\frac{}{\texttt{odd}\ []} \qquad \frac{\texttt{odd}\ t \quad \texttt{odd}\ f}{\texttt{odd}\ (t::f)} \qquad \frac{\texttt{even}\ f}{\texttt{odd}\ (\texttt{Node}\ (i,f))}$$

We can now define the notions of initial and final colorings. In the following, s stands for a possible state of array `bits`, that is an array of colors. The predicate $\texttt{I}\ s\ f$ characterizes an initial state s for a given forest f, as being all-white:

$$\texttt{I}\ s\ f \overset{\text{def}}{=} \forall i,\ i \in f \Rightarrow s(i) = \texttt{White}$$

Similarly, the predicate $\texttt{F}\ s\ t$ (resp. $\texttt{F}\ s\ f$) characterizes a final state s for a tree t (resp. a forest f):

$$\frac{\texttt{F}\ s\ t \quad \texttt{even}\ t \quad \texttt{I}\ s\ f}{\texttt{F}\ s\ (t::f)} \qquad \frac{\texttt{F}\ s\ t \quad \texttt{odd}\ t \quad \texttt{F}\ s\ f}{\texttt{F}\ s\ (t::f)}$$

$$\frac{}{\texttt{F}\ s\ []} \qquad \frac{s(i) = \texttt{Black} \quad \texttt{F}\ s\ f}{\texttt{F}\ s\ (\texttt{Node}\ (i,f))}$$

The precondition of `enum_forest` requires each tree of the forest to be either in an initial or final state, which can be defined as follows:

$$\texttt{any}\ s\ [f_1; \ldots; f_n] \overset{\text{def}}{=} \forall i,\ \texttt{I}\ s\ f_i \vee \texttt{F}\ s\ f_i$$

More precisely, the precondition requires that `any` holds on the concatenated forest `append f f0` (line 2). We also require that `valid` and `any` are sufficient conditions to ensure the precondition of `k` (line 3). Finally, the new continuation `k` which is built in `enum_forest` is given the same requirement (line 8).

We now turn to the postcondition of `enum_forest`. Simply speaking, we want to state that it switches the coloring of the forest from initial to final and conversely. As for the precondition, it would be a too strong requirement and we need to characterize the effect of `enum_forest` more subtly. Again, the parity of the number of colorings is playing a role, since an even number of colorings for the first tree would result in an unchanged coloring for the remaining of the forest and, conversely, an odd number for the first tree would result in a switch for the remaining of the forest. To denote unchanged colorings, we introduce the following predicate over two different states s_1 and s_2:

$$\texttt{same}\ s_1\ s_2\ f \overset{\text{def}}{=} \forall i,\ i \in f \Rightarrow s_1(i) = s_2(i)$$

Then we can define the effect of `enum_forest` between pre-state s_1 and post-state s_2, as the following, inductively defined predicate `mirror`:

$$\frac{\texttt{I}\ s_1\ t \quad \texttt{F}\ s_2\ t}{\texttt{mirror}\ s_1\ s_2\ t} \qquad \frac{\texttt{F}\ s_1\ t \quad \texttt{I}\ s_2\ t}{\texttt{mirror}\ s_1\ s_2\ t}$$

$$\frac{}{\texttt{mirror}\ s_1\ s_2\ []}$$

$$\frac{\texttt{mirror}\ s_1\ s_2\ t \quad \texttt{odd}\ t \quad \texttt{mirror}\ s_1\ s_2\ f}{\texttt{mirror}\ s_1\ s_2\ (t::f)}$$

[6] In the Coq files, we distinguish `validt` for trees and `validf` for forests, since there is no overloading; in this paper, we simply write `valid` for greater clarity. We proceed similarly for other predicates defined on trees and forests.

$$\frac{\texttt{mirror}\ s_1\ s_2\ t \quad \texttt{even}\ t \quad \texttt{same}\ s_1\ s_2\ f}{\texttt{mirror}\ s_1\ s_2\ (t :: f)}$$

This is the expected postcondition for `enum_forest` (line 18), and thus also a requirement over continuation `k` (lines 4 and 10).

As such, the specification of `enum_forest` is incomplete, as it does not say anything about the colors for the indices which are not in `f`. Since the state of `bits` is considered as a whole, these colors could have been changed. But for the correctness proof of `enum_forest`, it is necessary to know that recursive calls will not modify the color of node `i`. Thus we need a stronger postcondition, which "frames" the effects on array `bits`. For that purpose, we introduce the predicate

$$\texttt{eqout}\ s_1\ s_2\ f \overset{\text{def}}{=} \forall i,\ i \notin f \Rightarrow s_1(i) = s_2(i)$$

and use it in the postcondition of `enum_forest` (line 18) and `k` (lines 4 and 10).

4.4 Formal Proof

When processed with Who, the code in Figure 7 results in 17 proof obligations. They have been discharged using the Coq proof assistant, with the use of several auxiliary lemmas over predicates `even`, `any`, `mirror`, etc. Completing the proofs of these lemmas is still work in progress, as is the proof that `enum_forest` indeed realizes a Gray code enumeration. Anyhow, the Who tool already proved very useful in the process of obtained the verification conditions and debugging the specification.

5. Related Work

The Pangolin system, the implementation of the theoretical system of (Régis-Gianas and Pottier 2008), can also be used to reason about purely functional programs. It cannot deal with side effects. However, Pangolin has been one of the starting points of our work.

Systems for verification of first-order programs like Why (Filliâtre 2003) or Spec# (Barnett et al. 2004) are the other starting points of this work. These systems generally deal very well with usual features of first-order programs, for example arrays, use SMT solvers to discharge proof obligations and strive for the best possible automation. We try to improve on this work by adding higher-order features, trying to maintain the degree of automation and ease of use.

Honda, Yoshida and Berger (Honda et al. 2005; Berger et al. 2005) present a logic for imperative higher-order programs, first without, then with aliasing, but without allocation. They obtain strong theoretical results (e.g. a completeness result of their program logic). However, they describe the system in Hoare logic style with many non-structural rules, which seems to render it difficult to implement. The Who implementation has been straightforwardly derived from the syntax-directed formulation of a weakest-preconditions calculus.

The Ynot System (Nanevski et al. 2008) is an extension to the Coq proof assistant, capable of reasoning about imperative higher-order programs, including effectful functions as arguments, using a monad in which effectful computations can take place. Ynot is strictly more expressive than our system and features modular reasoning (abstracting over the specification and effect of a function as argument) while being able to reason about aliasing situations thanks to separation logic. We believe however, that one can obtain much simpler proof obligations, without separation logic, in the case of alias-free programs. We leave the work of substantiating this claim for future work.

Another line of work consists in giving a translation from an imperative to a functional language, with the aim of applying one of the techniques for purely functional programs to the translation. O'Hearn and Reynolds (O'Hearn and Reynolds 2000) present a translation from Algol with first-class references and dynamic allocation to a linear λ-calculus. They impose an alias-avoiding restriction which is similar to ours, by the notion of syntactic interference. It is not clear to us how this restriction compares to ours. Charguéraud and Pottier (Charguéraud and Pottier 2008) give a translation from Core ML with first-class references, dynamic allocation and aliasing to a functional language, using a very expressive type system with linear capabilities. Both systems have not yet been connected to a program logic, so it is unclear what kind of proof obligations one would obtain. Although we use a simpler programming language than Charguéraud and Pottier, we present a complete cycle from imperative programs to proof obligations.

6. Future Work

The current prototype implementation is still very limited. We plan to extend it in several ways. First, we intend to add creation of reference cells to the language. Using now-standard techniques of alias tracking this can be done in a straightforward way. Second, an important feature of functional languages are algebraic datatypes. We plan to add them natively to the language. This will result in programs much simpler to write and understand, and can also be exploited to obtain simpler proof obligations.

All the proofs presented in this paper have been done manually, using the Coq proof assistant (The Coq Development Team 2006). With the exception of the proof obligations for the Koda-Ruskey algorithm, all proofs are actually very simple and, after some structural reasoning, can be discharged by built-in automatic tactics (in particular the `omega` tactic for arithmetic reasoning) . This fact, along with the experience of the Pangolin system, makes us think that the resulting proof obligations are suited for SMT provers (Ranise and Tinelli 2006), which would avoid manual proofs in many cases. We are currently investigating the possibility to use SMT provers, using an encoding of higher-order logic into first-order logic.

Effectful computations with aliasing, mutable data structures and sharing is currently out of the scope of Who. We plan to build a more sophisticated analysis on top of the presented tool. A suitable memory model along with a translation, guided by this model, to the Who language, could deal with these difficult issues independently of considerations of a weakest precondition calculus.

7. Conclusion

We have presented Who, a tool to specify and prove effectful higher-order functions, as they occur very often in language mixing functional and imperative features. We have demonstrated its usefulness on several small and one more complex example. The current limitations are either subject to (relatively straightforward) future work, like algebraic datatypes and even the treatment of local allocation, or motivated by its status as an intermediate language, target of a tool applying a more sophisticated aliasing analysis.

Acknowledgments

We would like to thank the anonymous referees for their helpful comments.

References

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

Mike Barnett, Robert DeLine, Bart Jacobs, Bor-Yuh Evan Chang, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium*, volume 4111 of *LNCS*, pages 364–387, 2005.

Martin Berger, Kohei Honda, and Nobuko Yoshida. A logical analysis of aliasing in imperative higher-order functions. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 280–293, New York, NY, USA, 2005. ACM.

Gérard Boudol. Fair cooperative multithreading or typing termination in a higher-order concurrent imperative language. In *Proceedings, 18th International Conference on Concurrency Theory*, 2007.

Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ACM International Conference on Functional Programming (ICFP)*, pages 213–224, September 2008.

J.-C. Filliâtre and F. Pottier. Producing All Ideals of a Forest, Functionally. *Journal of Functional Programming*, 13(5):945–956, September 2003.

Jean-Christophe Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13 (4):709–745, July 2003.

Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.

Kohei Honda, Nobuko Yoshida, and Martin Berger. An observationally complete program logic for imperative higher-order functions. In *In Proc. LICS'05*, pages 270–279, 2005.

Johannes Kanig and Jean-Christophe Filliâtre. Proof of higher-order programs with effect polymorphism. Technical report, march 2009. URL `http://www.lri.fr/~kanig/files/who-theory.pdf`.

Donald E. Knuth. *The Art of Computer Programming*, volume 4, Pre-Fascicle 2a: A Draft of Section 7.2.1.1: Generating all n-tuples. Addison-Wesley, September 2001. Circulated electronically. `http://www-cs-staff.stanford.edu/~knuth/news.html`.

Yasunori Koda and Frank Ruskey. A Gray code for the ideals of a forest poset. *Journal of Algorithms*, 15(2):324–340, September 1993. `http://csr.csc.uvic.ca/home/fruskey/Publications/ForestIdeals.ps`.

P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *ICFP '08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, September 2008.

Peter W. O'Hearn and John C. Reynolds. From Algol to polymorphic linear lambda-calculus. *Journal of the ACM*, 47(1):167–223, 2000.

Silvio Ranise and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.smtcomp.org`, 2006.

Yann Régis-Gianas and François Pottier. A Hoare logic for call-by-value functional programs. In *Proceedings of the Ninth International Conference on Mathematics of Program Construction (MPC'08)*, pages 305–335, July 2008.

The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.1*, July 2006. `http://coq.inria.fr`.

Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.