Abstraction and Genericity in Why3

Jean-Christophe Filliâtre, Andrei Paskevich

Abstraction and Genericity in Why3

Jean-Christophe Filliâtre and Andrei Paskevich*

Université Paris-Saclay, CNRS, Inria, LRI, 91405, Orsay, France

Abstract. The benefits of modularity in programming — abstraction barriers, which allows hiding implementation details behind an opaque interface, and genericity, which allows specializing a single implementation to a variety of underlying data types — apply just as well to deductive program verification, with the additional advantage of helping the automated proof search procedures by reducing the size and complexity of the premises and by instantiating and reusing once-proved properties in a variety of contexts.

In this paper, we demonstrate the modularity features of WhyML, the language of the program verification tool Why3. Instead of separating abstract interfaces and fully elaborated implementations, WhyML uses a single concept of *module*, a collection of abstract and concrete declarations, and a basic operation of *cloning* which instantiates a module with respect to a given partial substitution, while verifying its soundness. This mechanism brings into WhyML both abstraction and genericity, which we illustrate on a small verified Bloom filter implementation, translated into executable idiomatic C code.

1 Introduction

When Alice writes code that uses hash tables, she does not need direct access to the actual implementation of that data structure — only to the handful of operations provided by it. Truth be told, she would rather not have that access: less risk to break her data structure by mistake, and she can also swap one implementation for another provided that the offered operations behave in the same way. What she needs, however, is hash tables for cabbages and hash tables for kings, and hash tables for whatever other data type she has in her code, for which she has written a hash function and an equality test[1].

If Alice also wants to formally verify her program, then *not having* access to the implementation may easily become a necessary requirement for her success. The automated provers are more stubborn than smart, and they will happily drown in all the minute properties of the implementation, whereas they could easily succeed in their proof, were they given just the simple specifications of hash table operations. The best way to get an automated proof of anything is to give the prover very little data written in very simple terms (incidentally, this

* This research was partly supported by the French National Research Organization (project VOCAL ANR-15-CE25-008).

[1] "Cabbage hash can be delicious," said Alice, "but I would never dare to hash a king."

also helps if at some later point you need to slightly change your problem and be able to prove it again). And then, if you do get the proof, just make sure that it somehow still holds if your terms are not as simple and your problem is actually much larger than what you let your prover believe.

What this means for Alice, is that she would prefer to verify her code without knowing anything about how hash tables are implemented, and if she also verified her implementation of hash tables, she would prefer to do it just once, without giving her prover any details about the type of the objects to store, only that there is an equality test and a hash function for them. If her verification framework is done right, this should be enough to guarantee that her final executable — where sophisticated and highly performant hash tables are reused for ships and shoes and sealing-wax sticks — is flawless.

Probably, any approach invented for modularity in programming can be adapted to program verification. After all, functional properties of programs are just types, if you take a sufficiently expressive type system. The purpose of this work is to show how we do it in WhyML, the language of the program verification tool Why3 [6].

Our framework is inspired by theorem proving just as much as by programming. In classical, non-constructive, logic, the difference between full implementation and partial specification is just how much you say about your type, function, or program. Also, apart from some symbols that are given a fixed meaning in your formalism (equality predicate or integer type), everything else is just an identifier bound by some quantifier, explicitly or implicitly, up in the scope. And finally, we need to break our formalizations in many small pieces, to keep the proof tasks within reach of automated provers.

This has led us to quite a minimalistic system of modules, which are simple collections of specifications and code, with only two basic operations: (a) link module A from module B so that B can have access to the contents of A, and (b) put a fresh copy of module A inside module B, while replacing some symbols introduced in A with the symbols from B. The second operation we call *module cloning*, and it turned out to be surprisingly (not that surprisingly, if one comes from classical logic) versatile. One of the first cloning instructions we wrote was in the standard library of integers, where we imported the ring axioms by cloning the generic library of rings, replacing the abstract domain `t` with `int`, abstract function `plus` with `+`, etc. We did not need to say that the module of generic rings was a functor parametrized by that type and those operations. Instead we simply declared an abstract type and three abstract functions on it. And Why3 allows us to instantiate any abstract symbol (or none at all) when cloning a module, on condition that we respect its properties.

Module cloning can help us create abstraction barriers. Write a module A with abstract types and abstract functions, described only by their specifications — this is your interface. Client code may link to A or clone it (to have a fresh instantiated copy), and be verified without knowing anything about the implementation. Write an implementation — a module B with fully defined types and fully implemented operations, and then clone A into B while instantiating every abstract symbol from the interface with its implementation. Why3 will check the types and the side effects, and will generate verification conditions for you to prove in order to ensure that the implementation respects the interface.

Module cloning can help us implement generic code. Rings and integers cited above are just one example. Write a module A with all the parameters as abstract symbols — this is your functor. State (and prove) all the generic properties you may need. Clone A into the client code while instantiating the abstract symbols with concrete types and concrete operations. Why3 will transfer all the properties you have proved in the generic module to the client code without requiring you to reprove them.

The idea is simple but imperative programs are complex things. In the rest of the paper, we describe how modules are implemented in Why3 (Section 2), and illustrate their use on an extended example of Bloom filters (Section 3), where proofs performed in minimal contexts lead us to a fully implemented correct-by-construction C program (Section 4). The complete formalization is available at the companion web page http://why3.lri.fr/isola-2020/.

2 WhyML Modules

A building block of a WhyML development is a *declaration*. A declaration can introduce a data type, a mathematical symbol, a logical proposition or a program function. Some declarations provide full information about the symbols they introduce: the structure of a data type is fully exposed, a mathematical symbol is given a sound definition, a proposition is proved, a program function is implemented. Other declarations give us a partial view: we only get to know some fields of a data type, a mathematical symbol is only given a name and a type signature, a proposition is posited without a proof, and a program function shows its specification but the actual implementation is unknown. Mixing concrete and abstract declarations is best suited for program verification, as we get to freely choose the level of abstraction for each element involved. Of course, if we intend to obtain executable code at the end, we must be able to refine the abstract portions into concrete implementations, while preserving the properties obtained through proof.

Declarations are structured using *scopes* and *modules*. Scopes help us to manage namespaces. Let us say, we declare a function symbol `f` in a scope `S`:

```
scope S
  function f ...
end
```

After closing the scope `S`, we can refer to `f` by using a qualifier:

```
lemma L: ... S.f ...
```

or by temporarily opening `S` inside a WhyML expression:

```
predicate p = ... S.( ... f ... ) ...
```

or by importing `S` into the current namespace until the end of the current scope:

```
import S
constant c = ... f ...
```

Sometimes we want to import a scope right away:

```
scope import T     (* the same as writing ‘import T’... *)
  predicate q ...  (* ...right after closing the scope  *)
end
```

This is useful if there is some other symbol named `q` declared in the current scope. WhyML forbids giving the same name to two symbols declared in the same scope, but permits shadowing with imported names.

Scopes can be nested and reopened. They are used for name resolution only and do not affect the logical or operational semantics of WhyML declarations.

Modules, on the other hand, provide the semantic structure of a WhyML program. Each module contains a sequence of declarations and scopes and references to other modules. These references are of two kinds.

First, a module `N` can bring another module `M` in its logical context, and thus get access to the contents of `M`, through the operation **`use`**:

```
  module M                             module N
    type t                               use M
    function h (x: int): t               constant d: t = h 5
  end                                  end
```

Module `M` shares its contents with all modules that **`use`** it, either directly or indirectly. For example, if some third module **`use`**s both `M` and `N`, it will get access to the same type `t` and function `h` through both of them.

The other way to reference a module is by *cloning* it. This operation makes a full copy of the contents of the cloned module while simultaneously replacing some of its abstract symbols with suitable refinements:

```
module P
  clone M
  constant e: t     (* this is not the same type as t in M *)
end

module Q
  clone M with type t = int
  lemma idem: forall z: int. h (h z) = h z  (* h returns int *)
end
```

When cloning `M` in the module `P` above, the programmer does not specify any substitution to be performed. Thus the contents of `M` is copied into `P` verbatim. However, the copied declarations are now part of `P` and they are distinct from the original declarations in `M`. If some other module **`use`**s both `M` and `P`, it will get two different types named `t`: one from `M` and another from `P`.

As for the module Q in the same example, it copies the contents of M while replacing every occurrence of the type t with int. Since type t is abstract, this substitution is allowed. Still, if M contains any axioms about t, they may come in contradiction with the properties of type int (e.g., t could be axiomatized as a finite type in M), thus creating an inconsistency. This is why all axioms from a cloned module appear by default as *lemmas* in the cloning module, obliging the programmer either to prove them or to deliberately override the default.

Cloning a module does not affect the symbols that were added to it through the **use** command. For example, if we clone module N, we get access to the same type t and function h as if we have **use**d module M directly. Informally, one can see **use** as creating a window into another module. On the other hand, the symbols introduced with **clone** belong to the cloning module (and Why3 does actually put the instantiated declarations inside the cloning module), and thus can be further instantiated during subsequent **clone** commands. For example, one can write `clone P with type M.t = real`.

When we **use** or **clone** a module, we introduce new symbols to the logical context and thus, new names. In their shortest form, with no modifiers, both **use** and **clone** will put these names in a new scope, named after the module in question, and import that scope. Operations **use export** and **clone export** do not open a new scope, and put all the new names in the current namespace instead. For example, module P above can be equivalently written as follows:

```
module P
  scope import M    (* gets the name of the cloned module *)
    clone export M
  end
  constant e: t     (* we can also write 'M.t' here *)
end
```

We can choose a different name for the new scope by writing `use M as A` or `clone M as B`. In this form, the new scope is not imported automatically:

```
module P_alt_1
  clone M as B      (* scope B is not imported *)
  constant e: B.t   (* qualifier is required *)
end
```

unless we add the **import** modifier:

```
module P_alt_2
  clone import M as B   (* scope B is imported *)
  constant e: t         (* both 't' and 'B.t' work *)
end
```

It is important to note that module names can only appear in **use** or **clone** operations. In particular, it is impossible to refer to a symbol from a module that has not been added to the current context either through **use** or through **clone**. Once it is done, the scope structure will determine the fully qualified name for each symbol that came with that module.

In what follows, we discuss in more detail various aspects of cloning, paying most attention to the checks and verifications required to ensure the soundness of symbol instantiations. The cloning mechanism guarantees that all properties that have been established in the module being cloned — proved lemmas, verified program contracts, etc. — stay valid after instantiation and can be incorporated into the cloning module without creating a contradiction. This does not mean that cloning a module is always a conservative extension: as we have seen earlier, Why3 does not guarantee that the instantiated *axioms* of the cloned module are consistent with the current logical context (which is why it incites the programmer to prove them after instantiation). However, whatever has been *proved* in the cloned module must stay provable after cloning.

We call "original" the module being cloned and the symbols declared in it: type symbols, mathematical symbols, program symbols, etc. The substitution in a `clone` operation we call a "refinement", the original symbols on the left-hand side being "refined", and the ones on the right-hand side, which replace the originals in the cloned declarations, being "refining". Symbols that are given a full definition in the original module cannot be refined, and are simply transferred into the new context. Their definitions, however, are still instantiated with respect to the cloning substitution, similarly to how in module `Q` above, type `t` is replaced with `int` in the signature of the cloned function `h`.

Type declarations. Fully defined types in Why3 are sum types, non-private records, type aliases, and special numeric types:

```
type option 'a = None | Some 'a
type ref 'a = { mutable contents : 'a }
type point = (real, real)
type int8 = <range -128 127>
```

Being fully defined, these types cannot be replaced by cloning instructions. The only refinable types are private records:

```
type queue 'a = private { ghost mutable elts: list 'a }
```

WhyML programs can read the values of private records' fields, but cannot directly construct such records or modify their mutable fields. Instead, the necessary operations must be provided via abstract program functions.

A type without definition is considered to be a private record with no fields:

```
type t  (* the same as 'type t = private {}' *)
```

A private type whose fields are all ghost (meaning that they can only be used in specifications and in ghost computations, but cannot influence the observable program behaviour) is called "abstract". For example, the definition of type `queue` above can be equivalently written as follows:

```
type queue 'a = abstract { mutable elts: list 'a }
```

Both private and non-private records can be equipped with a type invariant:

```
type char = abstract { contents: string }
            invariant { length contents = 1 }
```

A type invariant is essentially an axiom that restricts possible values of the fields of a record type. Why3 requires type invariants to be satisfiable and generates appropriate proof obligations. Private records, records with mutable fields, and records with type invariants cannot be recursive in WhyML.

A cloning operation can instantiate a private record with a different type. The following restrictions apply:

1. The refining type must have the same number of type parameters as the original type.
2. All fields of the original type must be present in the refining type and have the same type. Here, as before, "the same type" is meant modulo instantiation: that is, if the field's type in the original record is `ref t` and the cloning substitution replaces `t` with `int`, the corresponding field in the refining type must have type `ref int`.
3. A mutable field in the original type must be mutable in the refining type.
4. A ghost field in the original type may become non-ghost in the refining type but not vice-versa.
5. New fields can be added, which can be mutable and/or ghost. Mutable fields, however, can only be added when the original type is explicitly declared as mutable or has mutable fields of its own.
6. The (instantiated) original invariant must hold for each value of the refining type; Why3 generates an appropriate proof obligation. One possible way to satisfy this requirement is to include the original invariant in the invariant of the refining type.
7. An original field with a mutable type that is not mentioned in the original type invariant can not occur in the invariant of the refining type either.

The last item deserves some discussion. Let us consider the following declaration:

```
type ptr 'a = private { segment: array 'a;
                        mutable offset:  int }
```

and a variable `p` of type `ptr`. Since `ptr` is private, modification of the mutable field `p.offset` is only possible through abstract functions operating on values of type `ptr`. What about the array in `p.segment`? One possibility is to treat it in the same way as `p.offset`, that is, to forbid writes into the array. Another is to allow modifications of `p.segment`. In the latter case, however, we must ensure that any given write into `p.segment` does not break the invariant of the `ptr` type. The problem, of course, is that `ptr` is a private type and its invariant can be strengthened during refinement. Since we do not know the full invariant of `ptr` right now, we cannot formulate an invariant preservation condition for the writes into `p.segment`. We can work around this problem by forbidding to constrain the values of the `segment` field in the current and *all future* type invariants of `ptr`, so that no state of `p.segment` can break the integrity of `p`. An easy way to ensure this is to forbid mentioning the field in the invariant altogether.

Thus, the presence or the absence of a field with a mutable type (such as `segment`) in the type invariant of a private type serves as an indication of the user intention: If the field is mentioned in the type invariant, then it becomes non-modifiable[2]; otherwise, it can be written into, but must not appear in the invariants of the refining types, ensuring that modifications are always safe.

Mathematical functions and predicates. Here, the rules are simple, because functions and predicates in WhyML are either provided with a definition:

```
predicate mem (x: 'a) (l: list 'a) = match l with
  | Nil      -> false
  | Cons y r -> x = y \/ mem x r
  end
```

or declared as abstract symbols, with only their name and type signature:

```
function length (s: string) : int
```

The defined functions and predicates cannot be refined and their definitions are simply transferred to the current module. An abstract function or predicate is refinable, and the refining symbol must have the exact same type signature modulo instantiation.

For example, the following module clones module `M` above, and refines both type `t` and function `h`:

```
module R
  use list.List
  function singleton (n: int) : list int = Cons n Nil
  clone M with type t = list int, function h = singleton
end
```

Refinement of functions and predicates does not produce proof obligations.

Logical propositions. Axioms, lemmas, and goals are not refinable: they cannot be replaced with some other propositions. However, Why3 allows the programmer to specify how they should be treated in the cloning module.

The goals in the original module are not transferred to the current module at all. Indeed, they have already been proved in the original module and thus do not need to be reproved after instantiation. And since they are not added to the logical context as premises (contrary to lemmas), the cloning module has no need for the original goals.

The original lemmas are cloned as lemmas; however, since they have already been proved in the original module, Why3 will not generate a proof obligation for them. If we do not want to keep a cloned lemma as a premise in the logical context (e.g., because it duplicates an existing premise), we can “recast” it as a

[2] In this case, due to the specifics of state handling in WhyML, not even abstract functions are allowed to announce a potential write in the `segment` field, which limits the usefulness of this kind of construction. This may be relaxed in future.

goal by writing **with goal** `L` in the cloning substitution (where `L` is the name of the original lemma). Then lemma `L` will not be copied to the current module.

Axioms require caution, because, as we have noted above, simply copying an original axiom into the new context may create an inconsistency. To prevent this from happening, WhyML clones axioms as lemmas by default (and generates proof obligations for them), and the programmer must explicitly specify which axioms of the original module are to be kept as axioms:

```
clone relations.PreOrder with axiom Refl, axiom Trans
```

This instruction clones the `PreOrder` module from the file `relations.mlw` from the standard library of Why3. It adds to the current module declarations of a new abstract type `t` and a new binary relation `rel` on `t` together with the axioms of reflexivity and transitivity of `rel`.

When cloning modules with numerous axioms, listing all of them would be tedious. Therefore, WhyML provides a shortcut **with axiom** `.` which preserves every axiom in the original module unless it is converted into a lemma or a goal elsewhere in the cloning substitution.

Program functions. Only the abstract program functions, characterized by their type signature and their contract, can be refined in a cloning substitution. However, due to the large variety of possible side effects in the original and refining functions the required checks are rather complex. For example, consider the following modules (we omit the references to the standard library of lists):

```
module Queue
  type queue 'a = abstract { mutable elts: list 'a }

  val enqueue (q: queue 'a) (x: 'a) : unit
    writes   { q.elts }
    ensures  { q.elts = (old q.elts) ++ Cons x Nil }
end

module TwoListQueue
  type queue 'a = { mutable front: list 'a;
                    mutable back:  list 'a;
              ghost mutable elts:  list 'a }
    invariant { elts = front ++ reverse back }

  let enqueue (q: queue 'a) (x: 'a) : unit
    writes   { q.back, q.elts }
    ensures  { q.elts = (old q.elts) ++ Cons x Nil }
  =
    q.back <- Cons x q.back;
    q.elts <- q.elts ++ Cons x Nil

  clone Queue with type queue = queue, val enqueue = enqueue
end
```

The first module requests the standard library of linked lists and declares an abstract type of mutable queues and an abstract enqueuing function. The second module implements the queue type and the enqueue operation, and then clones the first module, refining the two symbols.

After checking the correctness of the type refinement (remember that in an abstract record all fields are ghost and thus field `elts` is allowed to stay ghost in the implementation), Why3 proceeds to the refinement of `enqueue`.

The procedure starts with instantiating the prototype of the original abstract function. This step does not take the refining function into consideration; in fact, the same rules are applied when we simply transfer an abstract function into the cloning module without refining it. Prototype instantiation is non-trivial because the types in the original type signature, notably those involved in the side effects, may have been refined, revealing new mutable fields and new fields with mutable components. This is the case in our example: the modified parameter `q` has gained two new mutable fields, `front` and `back`.

Why3 applies the following rules when instantiating the "write" effect annotations on the mutable values whose type is refined (remember that all side effects in these annotations are latent and do not have to actually happen in any implementation):

1. All original mutable fields marked as written in the original prototype are considered written in the instantiated prototype.
2. All original fields not marked as written in the original prototype are not considered written in the instantiated prototype.
3. All new mutable fields are considered written in the instantiated prototype.
4. All mutable components of the new fields are considered written in the instantiated prototype.

All mutable values that are not modified in the original prototype, are not modified in the instantiated prototype either, regardless of how their type is refined.

According to these rules, the instantiated prototype of the original function `enqueue` is as follows:

```
let enqueue (q: queue 'a) (x: 'a) : unit
  writes   { q.front, q.back, q.elts }
  ensures  { q.elts = (old q.elts) ++ Cons x Nil }
```

Indeed, `q.elts` is considered written, as it was already marked as such in the original prototype (rule 1). The fields `front` and `back` are added to the "write" effect, since they are new mutable fields in a modified parameter `q` (rule 3). If the added fields `front` and `back` were not mutable but had a mutable type (say, `array 'a`), they would also appear in the instantiated effect by rule 4.

To sum up, the effect annotation in the original prototype does not change with respect to what is already known to the original module (rules 1, 2, and 5). However, each announced write effect extends to all added fields of the affected parameters. This allows the implementations of the original `enqueue` function to modify the new fields `front` and `back`.

Now that we obtained an instantiated prototype of the original abstract function, we need to compare it with the proposed refinement and verify that instantiation is legal. This requires multiple checks:

1. The type signatures must coincide.
2. Ghost parameters of the original should be ghost in the refinement (an implementation cannot depend on ghost data passed from the client code).
3. Ghost results of the refinement should be ghost in the original (an implementation cannot pass ghost data to client code unbeknown to it).
4. The refining function must not have effects unlisted in the instantiated prototype of the original.
5. The refining function must not create memory aliases that are not required by the original.
6. The refinement must satisfy the instantiated contract of the original.

In order to check these conditions, Why3 creates and verifies (and then throws out) a WhyML function whose specification comes from the instantiated original prototype and whose implementation consists in calling the refining function with the same parameters. The type-checking system of Why3 and its verification condition generator perform the necessary checks and produce an appropriate proof obligation for the last item. In the case of `enqueue`, this proof obligation is an easily provable tautology.

While the rules for prototype instantiation introduce new latent write effects, these effects only concern the values that are already marked as modified in the original prototype, and they are limited to the new fields. Since a caller of the abstract function only knows the fields in the original type declaration, it can only observe a modification in the new fields as a non-specific change of the whole value — which is covered by the effect annotation in the original prototype.

It is crucial for the soundness of cloning that no aliases exist between the values accessible to the caller and the "hidden state" represented by the added fields. Such an alias can only be created through a refinement of an abstract program function in the original module, and this is prevented by rule 5 above.

In the next section, we show how to use modules in a fully developed example, going from abstract specifications to executable C code. In particular, we demonstrate how module cloning expresses: (a) the relation between an interface and an implementation; (b) the relation between a generic module and its parameters; and (c) specialization of a generic module.

3 Example: Bloom Filters

A Bloom filter [4] is a data structure that implements a set and provides two operations: one to insert an element into the set and one to query the presence of an element in the set. The latter must always give a correct positive answer for elements that have been indeed inserted into the set, but it may return a false answer for the elements not in the set. In other words, false positives are allowed but false negatives are not.

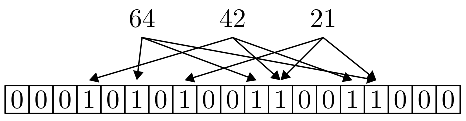

Fig. 1. A Bloom filter for integers, using $m = 19$ and $k = 3$.

A Bloom filter makes use of a bitmap (a Boolean array) of a given size m and of k hash functions $h_1, \ldots, h_k$ mapping the elements to integers between 0 and $m - 1$. When inserting an element x, we set the bits at indices $h_1(x), \ldots, h_k(x)$. When querying the presence of x, we return true if and only if *all* bits at indices $h_1(x), \ldots, h_k(x)$ are set.

Figure 1 illustrates a Bloom filter for integer elements where we use an array of 19 bits and 3 hash functions $h_1(x) = 34x$, $h_2(x) = 55x$, and $h_3(x) = 89x$ (all considered modulo 19). We insert three elements into the set, namely 21, 42, and 64. It results in seven bits being set (bits at indices 3, 5, 7, 10, 11, 14, and 15 in the array). If we now query the filter for the element 82, it reports that it is not in the set. Indeed, element 82 is mapped to bits 2, 7, and 14 and, though bits 7 and 14 are set, bit 2 is not and thus 82 does not belong to the set. But if we now query the filter for the element 80, it checks for bits 3, 11, and 14, which are all set, and thus reports that 80 belongs to the set. This is a false positive. If we query the filter for all elements between 0 and 99, it reports 17 positives: the three elements we added and 14 false positives. For the remaining 83 elements, we know for sure they do not belong to the set.

Despite being imprecise, a Bloom filter is a genuinely useful data structure. One good application is the following. Say we are implementing a storage whose operations are expensive, because they involve disk or network access. A Bloom filter can be conveniently placed between the storage and its client. When an element is added to the storage, it is added to the filter as well. Whenever the storage needs to be queried, we first query the filter. If the filter reports that the element is not in the storage, the answer is guaranteed to be correct and we avoid a costly operation. By themselves, Bloom filters are efficient data structures, in both space and time. With suitable choices of m and k, a Bloom filter can achieve an error ratio less than 1% with less than 10 bits per element [13].

It is worth pointing out that unlike a traditional hash table, a Bloom filter cannot be resized to accommodate an increasing number of elements. Indeed, the elements themselves are not kept in the Bloom filter and thus there is no way to rehash them into a larger array. This means that we must make an estimation of the expected size of the element set in advance, and pick the value of m accordingly. Similarly, there is no way to remove an element from a Bloom filter. Indeed, by clearing the bits corresponding to an element, we could remove other elements from the set, which would make the filter unsound. This is not a problem, since removing an element from the actual storage without updating the Bloom filter would merely lead to another false positive, which is allowed.

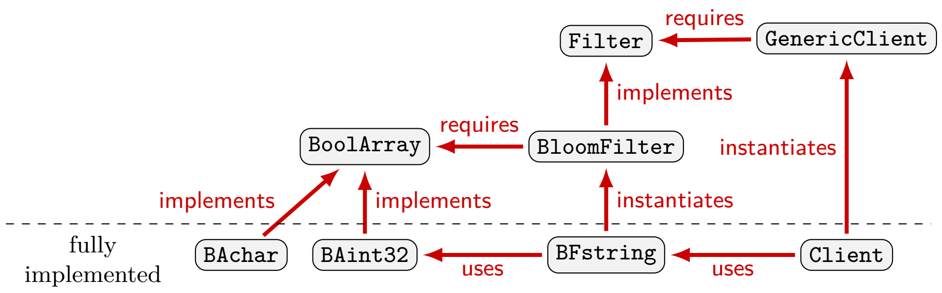

Fig. 2. Bloom filters in Why3: the module map.

If need be, the filter can be reconstructed from the storage on regular intervals, without compromising the asymptotic complexity.

Let us implement a Bloom filter with Why3. Our final objective is to get a verified C library of string filters. We decompose the Why3 code into eight different modules, depicted in Fig. 2. Here is a short overview:

– On the left side of the figure, we have three modules related to Boolean arrays. Module `BoolArray` is an interface and modules `BAchar` and `BAint32` are two implementations of this interface. The former implements a Boolean array rather naïvely, using one byte per element. The latter uses an array of 32-bit integers, using one bit per element.
– On the central part of the figure, we have three modules related to filters. Module `Filter` is an interface. It provides an abstract data type and three operations `create`, `add`, and `mem`. Module `BloomFilter` implements Bloom filters on the basis of various parameters: an implementation of Boolean arrays, values for m and k, a data type for the elements, and a set of hash functions. Then module `BFstring` instantiates all these parameters to get a fully implemented Bloom filter for strings.
– Finally, on the right side of the figure, we have two modules to make a quick test of the library. Module `GenericClient` uses the interface `Filter` to build a filter and perform a few additions and membership queries. Then module `Client` instantiates this generic client using module `BFstring`.

The four modules at the bottom of the figure are fully implemented. This means they can be translated to C (this is described in the next section). The red arrows in the figure illustrate the various ways in which the modules depend on each other, either through **`use`** or **`clone`** commands.

Boolean arrays. We start with an interface `BoolArray`, which declares a type `t` together with three operations `create`, `get`, and `set`.

```
module BoolArray
  type t = abstract { mutable contents: seq bool; }
  val create (size: uint32) : t
```

```
    val get (a: t) (i: uint32) : bool
    val set (a: t) (i: uint32) : unit
  end
```

We omit various details here, such as the modules imported from the standard library and the contracts for the three operations. The type `t` is an abstract record data type, with a single field named `contents`. Since all fields of an abstract type are ghost, the field `contents` can be used within any specification element, such as a function contract, but cannot be used in actual computation in the code. In other words, it serves as a *model* for the type `t`, but not as a part of its implementation. This model is a sequence of Boolean values (type `seq`, from Why3 standard library, can be seen as a purely applicative array) and this is all we need to provide suitable contracts to our three operations. For instance, operation `get` is given the following contract:

```
    val get (a: t) (i: uint32) : bool
      requires { i < length a.contents }
      ensures  { result = a.contents[i] }
```

For convenience, WhyML allows us to declare `contents` a coercion symbol, so that we can write simply `a` instead of `a.contents`. Module `BoolArray` does not incur any verification condition.

We now provide two different implementations of this interface. We start with a rather simple implementation with one byte per bit. We do this in a separate module `BAchar`. It also contains declarations for a type `t` and three operations `create`, `get`, and `set`.

```
  module BAchar
    type t = { mutable ghost contents: seq bool;
                                 arr: ptr uchar; }
      invariant { ... }
    let create (size: uint32) : t = ...
    let get (a: t) (i: uint32) : bool = ...
    let set (a: t) (i: uint32) : unit = ...
    ...
```

This time, however, our types and functions are fully implemented. Type `t` is still a record data type with a ghost field `contents`. But it also contains a non-ghost field `arr` that holds a pointer to an array of bytes (type `uchar` from Why3 standard library). A gluing invariant (omitted here) makes the connection between field `contents` (the model) and field `arr` (the implementation). The type `t` is not abstract anymore, which means we are now allowed to construct instances of that type. This is precisely what function `create` does.

Operations `create`, `get`, and `set` are now given definitions (omitted here). Their contracts are identical to those of module `BoolArray`. In particular, they only refer to field `contents`. Their definitions, of course, do make use of field `arr`. Why3 generates suitable verification conditions for these three definitions to be correct with respect to their contracts.

Finally, we show that module `BAchar` is indeed an implementation of the interface `BoolArray`, *i.e.*, that it *refines* module `BoolArray`. This is done with the help of a **clone** instruction:

```
    ...
    clone BoolArray with type t, val create, val get, val set
  end
```

Here, we use a syntactical shortcut that allows us to write only the left-hand side of the substitution when the refining symbol has the same name as the original. That is, we substitute the type `t` of module `BoolArray` with the type `t` we just defined, and similarly for the three operations.

This **clone** command generates several verification conditions. They are all rather trivial, as there is no invariant on type `BoolArray.t` and the contracts for the three operations are the same in the interface and the implementation.

We also provide a second implementation of interface `BoolArray` in a module called `BAint32`. It is a more efficient implementation that uses an array of 32-bit integers, where each element packs 32 Boolean values.

Filters. We proceed in a similar way for filters, though using two layers of refinement instead of one. We start with an interface, `Filter`, which declares types for elements and filters and three operations:

```
  module Filter
    type elt
    type filter = abstract { mutable contents: fset elt; }
    val create (m: uint32) : filter
    val add (x: elt) (s: filter) : unit
    val mem (x: elt) (s: filter) : bool
  end
```

This is similar to what we did earlier with module `BoolArray`. Here, the contents of type `filter` is modeled using a finite set. Then we implement Bloom filters in a second module `BloomFilter`. We start by introducing parameters for the type of elements and the family of `k` hash functions.

```
  module BloomFilter
    type elt
    val constant k: uint32
    val function hash (i: uint32) (e: elt) : uint32
```

The individual hash functions are identified with an index `i` in $0..\texttt{k}-1$. Then we move to the implementation of type `filter`. For that, we need a Boolean array, and so we bring a copy of `BoolArray` into the context.

```
    clone BoolArray
```

It is worth pointing out that this module is merely an *interface* for Boolean arrays. This means that our implementation of Bloom filters does not depend on a particular implementation of that data structure, and can be instantiated to use any of them. We can now define type `filter` on top of `BoolArray.t`.

```
    type filter = {
      mutable ghost contents: fset elt;
                          m: uint32;
                       barr: BoolArray.t; }
    invariant { length barr = m > 0 }
    invariant { forall x. mem x contents ->
                forall i. i < k -> barr[(hash i x) % m] }
```

The gluing invariant makes the connection between the model field `contents` and the implementation fields `m` and `barr`. Now we can implement the three operations over Bloom filters:

```
    let bloom_filter (m: uint32) : filter = ...
    let add (x: elt) (s: filter) : unit   = ...
    let mem (x: elt) (s: filter) : bool   = ...
```

Note that, despite being defined, these functions still depend on parameters `elt`, `k`, and `hash`. Thus, they are not executable.

Last, as we did with module `BAint32`, we check that this module refines module `Filter`, using a **`clone`** command.

```
    clone Filter with type elt, type filter,
      val create = bloom_filter, val add, val mem
  end
```

Again, this generates VCs that are all easily discharged.

In order to obtain executable code, we further refine module `BloomFilter` to produce a filter for strings. Here, we choose to use three hash functions.

```
  module BFstring
    type elt = string
    let constant k: uint32 = 3
    let function hash (h: uint32) (x: elt) : uint32 = ...
```

The actual implementation of `hash`, omitted here, is based on Fowler-Noll-Vo hash functions, following Louridas [12]. The remaining part is a **`clone`** command to instantiate `BloomFilter` with these parameters and with module `BAint32`:

```
    use BAint32
    clone export BloomFilter with val k, type elt, val hash,
      type BoolArray.t = BAint32.t,
      val BoolArray.create = BAint32.create,
      val BoolArray.get = BAint32.get,
      val BoolArray.set = BAint32.set
  end
```

Though for Why3 this **`clone`** command is no different from the previous two, from the programmer's point of view it is of a rather different flavor. Instead of claiming that module `BFstring` implements `BloomFilter`, it rather *instantiates* module `BloomFilter` with actual parameters. Notice that we write **`export`** in order to have the Bloom filter operations in the top namespace of `BFstring`.

Client. We conclude this example with a tiny client code. The main purpose is to check the usability of our contracts before going any further. We start with a client for module `Filter`, which we instantiate on string elements.

```
module GenericClient
  clone Filter with type elt = string
```

Then a test function builds a filter of a given size, inserts some strings, and checks for membership:

```
    let main () =
      let f = Filter.create 0x10000 in
      Filter.add "foo" f;
      Filter.add "bar" f;
      let b = Filter.mem "foo" f in
      assert { b };
      ...
  end
```

Once this is done, and verified, we can **clone** this generic client with a specific implementation of `Filter`, namely module `BFstring` we built earlier.

```
module Client
  use BFstring
  clone export GenericClient with type Filter.filter = filter,
    val Filter.create = bloom_filter, val Filter.add = add,
    val Filter.mem = mem
end
```

This verification passes, too, as we have already checked that `BFstring` implements `Filter`. (As for now, Why3 unnecessarily generates the same VCs a second time; this will be improved in the future.) Module `Client` is fully implemented and we will be able to translate it into executable C code, as shown in the next section. If we look again at the right-hand size of Fig. 2, we can see that the correctness of the `Client` module is ensured by the correctness of `GenericClient` and the fact that `BFstring` correctly refines `Filter`. A similar relation exists between modules `BFstring`, `BloomFilter`, `BAint32`, and `BoolArray`.

4 C Library

Once verification is complete, Why3 can automatically translate WhyML code to C [14]. The resulting C code is composed of three files:

- `baint32.c`, a translation of module `BAint32`;
- `bfstring.c`, a translation of module `BFstring`;
- `client.c`, a translation of module `Client`.

File `bfstring.c` makes use of functions from `baint32.c` and file `client.c` makes use of functions from `bfstring.c`. Each C file comes with a corresponding

```
uint32_t * create(uint32_t size) {
  uint32_t n, i, o;
  uint32_t * p;
  n = 1U + (size - 1U) / 32U;
  p = malloc(n * sizeof(uint32_t));
  assert (p);
  o = n - 1U;
  for (i = 0U; ; ++i) {
    p[(int32_t)i] = 0u;
    if (i == o) {
      break;
    }
  }
  return p;
}
```

Fig. 3. Generated C code for function `create` from module `BAint32`.

header file (`.h`). These files are available at `http://why3.lri.fr/isola-2020/`. Figure 3 contains the C code for function `create` from file `baint32.c`, resulting from the translation of function `create` from module `BAint32`. We can make two comments regarding this code. First, `assert` is used so that we can assume that the value returned by `malloc` is not `NULL` in the following, without having to test it. This is reflected on the Why3 side with an `assert` function that ensures (in its postcondition) that `p` is not `NULL`. Second, the rather unusual form of the `for` loop, using a `break` statement, ensures that even a loop up to the maximum representable value is sound with respect to the WhyML semantics. In this case, the loop is bounded by `n-1` so a traditional loop would be fine but Why3 does not make any effort to figure that out.

It is worth pointing out that each generated header file exposes *all* declarations from the corresponding WhyML module. For instance, file `baint32.h` declares the functions `create`, `get`, and `set`, as expected, but also "internal" functions `one_bit`, `bit_set`, and `set_bit`. Similarly, file `bfstring.h` declares the structure `filter` and the functions `bloom_filter`, `add`, and `mem`, but also the global variable `k` and the functions `hash` and `bit`. We translate all declarations because the translation is made on per-module basis. In WhyML, modules do not have dedicated interfaces and any module using module `Baint32` has access to all of its declarations. Thus, this ability must not be lost in translation.

An argument can be made that it is not crucial to ensure any abstraction barrier in the translated code since we have already made use of it on the WhyML side. This argument is less applicable when we develop a verified library for the target language, which is the case of `bfstring`. Indeed, the development will be pursued in the target language and thus it would be nice to hide the translated code behind a suitable interface. The simplest way to achieve it is just to remove unnecessary declarations from the generated header files. Finally, when translating to C, the whole discussion is moot since there is no proper encapsulation in C (it is always possible to bypass header files).

5 Related Work

The idea of conducting verification through stepwise refinements is not new. It is at the basis of Abrial's B method [1] for instance. In this context, abstract machines, which can be seen as interfaces, are gradually refined into fully executable machines, which are implementations. This is quite close to what we do: for instance, when we start with an interface `Filter` and refine it into an implementation `BFstring` in two steps. Proper modularity is also offered by the B method, as a machine is referring to the abstract version of another machine (its interface) and not to its refinements. Again, this is similar to what we do, for instance with our `GenericClient` referring to the interface `Filter`. Yet, there are fundamental differences between B machines and Why3 modules, the main being that B machines are state machines. Though Why3 modules can definitely be used to specify and implement state machines, they are not limited to this usage. Why3 modules may provide data types (as Boolean arrays and filters in our example) and this has no counterpart in the B method.

Abstraction and genericity are handled in programming languages in various ways. Most of these solutions can be readily used or adapted for use in program verifiers. When a program verifier is built for an existing programming language, such as Java for instance, it is natural to apply the abstraction and genericity mechanisms (*e.g.*, object-oriented programming, visibility modifiers, generic types) to the specification/verification level. This is done in tools such as VeriFast [7] or KeY [2] for instance. When a program verifier is providing its own programming language, it is nonetheless possible to reuse mechanisms from the programming community. The Coq proof assistant, for instance, implements both a module system inspired by that of OCaml [11,5] and type classes inspired by those of Haskell [17,16].

Why3 modules are not a direct implementation of a concept from any programming language. Yet, they have obvious connections with traits [15] and mixins [3], even if they are not cast in some object-oriented context. Indeed, Why3 modules mix declarations and definitions, may require parameters to come with some operations (by cloning suitable “interfaces”), and may provide new definitions on top of these parameters (in modules to be later cloned in suitable contexts). The comparison stops at some point, however, as Why3 modules are not centered around types. A parameter of a Why3 module can be a constant, a function, etc., which means more flexibility. On the other hand, Why3 modules cannot be used to require that a type parameter of a polymorphic type or function provides some operations, contrary to traits or type classes.

Closest to our work is likely to be Dafny [9], where modules are used to organize the namespace and to restrict visibility of symbols or symbol definitions [10]. Thus it provides adequate abstraction during the verification of client code, though this is done by hiding implementation details rather than having the client exposed to an interface only. There is a notion of module refinement in Dafny [8]. As in Why3, it allows declarations to be refined with definitions and it permits data refinement, though it is class-based in Dafny and record-based in Why3. Dafny goes a step forward in program refinement, allowing reduction of nondeterminism in program statements during refinement.

6 Conclusion

We have shown how abstraction and genericity are provided in the Why3 program verifier through a notion of modules and a module cloning operation. The latter performs a partial substitution on a module, replacing some of its abstract declarations with concrete ones, and generates suitable verification conditions to guarantee correctness. In this paper, we demonstrated this mechanism on a library of Bloom filters, using several modules and refinement steps.

The module system of Why3, despite being usable (and extensively used), can still be improved in several regards.

First, we should avoid redundant verification conditions (such as ones generated for the `clone` instruction in the `Client` module) by taking into account the previously made refinements. In practice, these redundant VCs are usually easy to discharge, but it is preferable not to produce them at all.

Second, we should add support for scope-level cloning substitutions, which would allow us to write simply `clone GenericClient with scope Filter = BFstring`, and avoid long and tedious enumeration of individual refinements.

Third, it would be convenient to annotate an "implementation" module with its designated interface, e.g., by writing `module BAint32 : BoolArray`. This notation should automatically add an appropriate cloning instruction at the end of `BAint32`, ensuring that it indeed refines `BoolArray`. Furthermore, any subsequent `use` of `BAint32` in the client code should only add to the logical context the contents of `BoolArray`, acting as an abstraction barrier (of course, translation into executable code would still use the concrete definitions from `BAint32`). This can be achieved by implicitly replacing such `use` instructions with cloning of `BoolArray`, like we did in the `BloomFilter` module above; also, renaming substitutions should be applied to ensure symbol sharing where necessary.

Acknowledgments. We are grateful to Claude Marché, Jacques-Henri Jourdan, and Rustan Leino for their insightful remarks and suggestions.

References

1. Jean-Raymond Abrial. *The B-Book, assigning programs to meaning.* Cambridge University Press, 1996.
2. Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science.* Springer, 2016.

3. Davide Ancona and Elena Zucca. An algebraic approach to mixins and modularity. In M. Hanus and M. Rodrìguez Artalejo, editors, *5th Intl. Conf. on Algebraic and Logic Programming*, number 1139 in Lecture Notes in Computer Science, pages 179–193, Berlin, 1996. Springer.
4. Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.
5. Jacek Chrząszcz. *Modules in Type Theory with Generative Definitions*. PhD thesis, Warsaw University, Poland and Université de Paris-Sud, January 2004.
6. Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.
7. Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.
8. Jason Koenig and K. Rustan M. Leino. Programming language features for refinement. In John Derrick, Eerke A. Boiten, and Steve Reeves, editors, *Proceedings 17th International Workshop on Refinement, Refine@FM 2015, Oslo, Norway, 22nd June 2015.*, volume 209 of *EPTCS*, pages 87–106, 2015.
9. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.
10. K. Rustan M. Leino and Daniel Matichuk. *Modular Verification Scopes via Export Sets and Translucent Exports*, pages 185–202. Springer International Publishing, Cham, 2018.
11. Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.
12. Panos Louridas. *Real-World Algorithms: A Beginner's Guide*. The MIT Press, 2017.
13. Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, USA, 2005.
14. Raphaël Rieu-Helft, Claude Marché, and Guillaume Melquiond. How to get an efficient yet verified arbitrary-precision integer library. In *9th Working Conference on Verified Software: Theories, Tools, and Experiments*, volume 10712 of *Lecture Notes in Computer Science*, pages 84–101, Heidelberg, Germany, July 2017.
15. Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In Luca Cardelli, editor, *ECOOP 2003 – Object-Oriented Programming*, pages 248–274, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
16. Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In Otmame Aït-Mohamed, César Muñoz, and Sofiène Tahar, editors, *21th International Conference on Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*. Springer, August 2008.
17. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.