Why3 – Where Programs Meet Provers

Jean-Christophe Filliâtre, Andrei Paskevich

▶ To cite this version:

Jean-Christophe Filliâtre, Andrei Paskevich. Why3 – Where Programs Meet Provers. ESOP'13 22nd European Symposium on Programming, Mar 2013, Rome, Italy. hal-00789533

HAL Id: hal-00789533
https://inria.hal.science/hal-00789533
Submitted on 18 Feb 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Why3 — Where Programs Meet Provers

Jean-Christophe Filliâtre[1,2] and Andrei Paskevich[1,2]

[1] Lab. de Recherche en Informatique, Univ. Paris-Sud, CNRS, Orsay, F-91405
[2] INRIA Saclay – Île-de-France, Orsay, F-91893

Abstract. We present `Why3`, a tool for deductive program verification, and `WhyML`, its programming and specification language. `WhyML` is a first-order language with polymorphic types, pattern matching, and inductive predicates. Programs can make use of record types with mutable fields, type invariants, and ghost code. Verification conditions are discharged by `Why3` with the help of various existing automated and interactive theorem provers. To keep verification conditions tractable and comprehensible, `WhyML` imposes a static control of aliases that obviates the use of a memory model. A user can write `WhyML` programs directly and get correct-by-construction OCaml programs via an automated extraction mechanism. `WhyML` is also used as an intermediate language for the verification of C, Java, or Ada programs. We demonstrate the benefits of `Why3` and `WhyML` on non-trivial examples of program verification.

1 Introduction

`Why3` is a platform for deductive program verification [1]. It provides a rich language of specification and programming, called `WhyML`, and relies on external theorem provers, both automated and interactive, to discharge verification conditions. The tool comes with a standard library of logical theories (integer and real arithmetic, sets and maps, etc.) and of basic programming data structures. `WhyML` is used as an intermediate language for the verification of C, Java, or Ada programs [2], in a similar fashion to the Boogie language [3]. Besides, `WhyML` strives to be comfortable as a primary programming language and inherits numerous high-level features from ML, listed below.

The specification component of `WhyML`, used to write program annotations and background logical theories, is presented in [4], and here we only mention the most essential features. `Why3` is based on first-order logic with rank-1 polymorphic types and several extensions: recursive definitions, algebraic data types, and (co-)inductive predicates. Pattern matching, `let`-expressions, and conditional expressions are allowed both in terms and in formulas. A type, a function, or a predicate can be given a definition or just declared as abstract symbols and then axiomatized. The specification language of `Why3` does not depend on any features of the programming language, and can serve as a rich common format for theorem proving problems, readily suitable (via `Why3`) for multiple automated and interactive provers, such as Alt-Ergo, CVC3, Z3, E, SPASS, Vampire, Coq, or PVS. When a proof obligation is dispatched to a prover that does not support some language features, `Why3` applies a series of encoding transformations to, for example, eliminate pattern matching or polymorphic types [5].

2 Programming Language

`WhyML` can be seen as an ML dialect, with two important restrictions. Firstly, in order to generate first-order proof obligations, `WhyML` is also limited to the first order: Nested function definitions and partial application are supported, but higher-order functions are not. Secondly, in order to keep proof obligations more tractable for provers and more readable (hence debuggable) for users, `WhyML` uses no memory model and imposes a static control of aliases instead. Every l-value in a program must have a finite set of names and all these names must be known statically, at the time of generation of verification conditions. In particular, recursive data types cannot have mutable components. This restriction is not as limiting as it may seem, and we show in the next section that it does not preclude us from writing and verifying complex algorithms and data structures.

`WhyML` functions are annotated with pre- and post-conditions for normal and exceptional termination, and `WhyML` loops are annotated with invariants. Recursive functions and `while`-loops can be given variants (*i.e.* values that decrease at each recursive call or iteration) to ensure termination. Statically checked assertions can be inserted at arbitrary points in a program. Verification conditions are generated using a standard weakest-precondition procedure. Every pure type, function or predicate introduced in the logical component can be used in a `WhyML` program. For instance, the type of integers and basic arithmetic operations are shared between specifications and programs.

The mutable state of a computation is embodied in mutable fields of record data types. Mutable data types can be nested. For example, a polymorphic resizable array can be modeled by a record with a mutable field containing an ordinary fixed-size array:

```
type rarray 'a = { mutable data: array 'a; mutable size: int }
      invariant { 0 ≤ size ≤ data.length }
```

Here, the type is accompanied by an invariant, *i.e.* a logical property imposed on any value of that type. `Why3` assumes that any `rarray` passed as an argument to a program function satisfies the invariant and it produces a proof obligation every time an `rarray` is created or modified in a program. Notice that this requires that types with invariants not be used in recursive data structures, just as mutable types.

An important feature of `WhyML` is ghost code, *i.e.* computations that only serve to facilitate verification and that can be safely removed from a program without affecting its final result. A ghost expression cannot be used in a non-ghost computation, it cannot modify a non-ghost mutable value, and it cannot raise exceptions that would escape into non-ghost code. However, a ghost expression can use non-ghost values and its result can be used in program annotations. A classical use case for ghost code is that of step counters to prove time complexity of an algorithm. It also serves to equip a data structure with a ghost field containing a pure logical "view" for specification purposes.

3 Case Studies

We have used `WhyML` to verify a lot of non-trivial data structures and algorithms. Our gallery (`http://proval.lri.fr/gallery/why3.en.html`) currently contains 67 case studies. In this section, we illustrate three different kinds of verification.

Verification of an Algorithm. Let us consider the Knuth-Morris-Pratt algorithm for string searching [6]. A string is simply an array of characters. Arrays are imported from the `Why3` standard library. Conversely, the type of characters is declared as an abstract, uninterpreted type `character`. The Knuth-Morris-Pratt algorithm is then implemented as a function that receives two strings `p` and `t` and that returns, if any, the position of the first occurrence of `p` in `t` and, otherwise, the length of `t`:

```
let kmp (p a: array character)
  requires { 1 ≤ length p ∧ 0 ≤ length a }
  ensures  { first_occur p a result }        = ...
```

where `first_occur` is a predicate introduced earlier in the specification. To get an executable code, `Why3` translates `WhyML` to OCaml. In the process, uninterpreted `WhyML` types are either mapped to existing OCaml types or left as abstract data types. In the example above, this results into the following OCaml function:

```
val kmp: character array → character array → Num.t
```

where `array` is the OCaml built-in type, `character` is an abstract data type, and `Num.t` is the type of arbitrary precision integers from OCaml library. Such a mapping can be customized at the user level. The key point here is *genericity*. Extracted code is parameterized w.r.t. uninterpreted symbols, such as the `character` type from the above example. It is then possible to instantiate the extracted code in different ways, for example by wrapping it into an OCaml functor.

Verification of a Data Structure. Let us implement hash tables (associative arrays) in `WhyML`, using an uninterpreted type `key` for keys:

```
type t 'a = { mutable size: int;   (* total number of elements *)
              mutable data: array (list (key, 'a)); (* buckets *) }
```

where arrays and lists are imported from the `Why3` standard library. Field `data` is declared mutable, in order to allow dynamic resizing, for the case when the array holding the buckets is replaced by a new, larger array. This operation changes the current set of aliases and the type system of `WhyML` can detect and safely handle it. In particular, after the resize, one cannot use any stale pointer to the old value of `data`. Also, the new value of `data` must be fresh. The key point here is *modularity*: One can implement resizing in a separate function and call it, for instance, from the `add` function that inserts a new element in the table.

Specification of a Data Structure. There are data structures that cannot be implemented in `WhyML`. Simply speaking, these are pointer-based data structures where mutable nodes are arbitrarily nested, *e.g.* doubly-linked lists or mutable trees. Still we can easily *model* such data structures and then verify the programs that use them. Let us consider, for instance, a program building a perfect maze using a union-find data structure, as proposed in the VACID-0 benchmark [7]. A union-find can be implemented in `WhyML` using arrays. However, a more flexible implementation, with chains of pointers, is beyond the scope of `WhyML`, and is simply modeled as follows:

```
type uf model { mutable contents: uf_pure }
```

There are three ideas here. First, the keyword **`model`** replaces the equal sign. This means that type `uf` is not a record, as far as programs are concerned, but an abstract data type. Inside specifications, though, it is a record and its field `contents` may be accessed. Second, field `contents` is declared mutable, to account for the fact that `uf` is a mutable data structure. Last, a pure data type `uf_pure` represents the immutable snapshot of the contents of the union-find data structure.

We then declare and specify operations over type `uf`. For instance, the function `find` that returns the representative of the class of a given element and may modify the structure (*e.g.* for path compression) can be specified as follows:

```
val find (u : uf) (x : elt) : elt  writes {u}
  ensures { result = repr u x /\ same_repr u (old u) }
```

The key point here is *encapsulation*: Though we cannot implement the union-find data structure, we can declare an interface data type to model it and then verify a client code (in this case, a program building a maze). Any implementation of union-find could be used without compromising the proof of the client code.

4 Future Work

The most immediate direction of our future development is the ability to verify that a given implementation conforms to an interface. This amounts to establishing a refinement relation between `WhyML` modules, their data types and their functions, be they defined or merely specified. We also plan to introduce some higher-order features in the specification language, *e.g.* set comprehensions and sum-like operations, together with suitable encodings to first-order logic. A more ambitious goal would be to accept higher-order programs in `WhyML`, in order to bring it closer to functional programming. Finally, our long-term goal is to merge the specification and programming languages, in the spirit of PVS and ACL2. The challenge is two-fold. We want to allow imperative constructions in pure functions, provided they do not break referential transparency. Even more importantly, we want to state and prove theorems about `WhyML` programs, beyond what is possible to express using pre- and postconditions.

References

1. `Why3`, a tool for deductive program verification, GNU LGPL 2.1, `http://why3.lri.fr`
2. Guitton, J., Kanig, J., Moy, Y.: Why Hi-Lite Ada? In: Boogie. (2011) 27–39
3. Barnett, M., DeLine, R., Jacobs, B., Chang, B.Y.E., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: FMCO. Volume 4111 of LNCS. (2005)
4. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie. (2011) 53–64
5. Bobot, F., Paskevich, A.: Expressing Polymorphic Types in a Many-Sorted Language. In: FroCoS. Volume 6989 of LNCS. (2011) 87–102
6. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM Journal on Computing **6** (1977) 323–350
7. Leino, K.R.M., Moskal, M.: VACID-0: Verification of ample correctness of invariants of data-structures, edition 0. In: VSTTE. (2010)