A Why3 Framework for Reflection Proofs and its Application to GMP's Algorithms

Guillaume Melquiond[1] and Raphaël Rieu-Helft[2,1]

[1] Inria, Université Paris-Saclay, Palaiseau F-91120
[2] TrustInSoft, Paris F-75014

Abstract. Earlier work showed that automatic verification of GMP's algorithms using Why3 exceeds the current capabilities of automatic solvers. To complete this verification, numerous cut indications had to be supplied by the user, slowing the project to a crawl. This paper shows how we have extended Why3 with a framework for proofs by reflection, with minimal impact on the trusted computing base. This framework makes it easy to write dedicated decision procedures that make full use of Why3's imperative features and are formally verified. We evaluate how much work could have been saved when verifying GMP's algorithms, had this framework been available. This approach opens the way to efficiently tackling the further verification of GMP's algorithms.

Keywords: Decision procedures, Proofs by reflection, Deductive program verification, Nonlinear integer arithmetic

1 Introduction

The Why3 software-verification tool[3] offers an ML-like language (WhyML) that makes it possible to write programs and to specify the functional behavior of these programs using pre- and post-conditions, and loop invariants [7]. The tool then turns programs and specifications into theorem statements that can be sent to external provers, be they automated (e.g., SMT or TPTP solvers) or interactive (e.g., Coq, Isabelle/HOL, PVS). Once these theorems have been proved, and assuming that Why3 and the external provers are sound, the programs are known to satisfy their specification.

In an earlier work, we used Why3 to implement algorithms from the GNU Multi-Precision library,[4] GMP for short, to prove them correct, and to generate a compatible C library [12]. The proofs were done using automated provers only, mostly SMT ones. While some algorithms are extremely intricate (e.g., division [11]), we ended up having to litter the code with many more assertions than we initially envisioned, as exemplified on Figure 4, line 24. Seemingly trivial theorems were confusing solvers to no end. Indeed, they involved nonlinear integer

[3] `http://why3.lri.fr/`
[4] `http://gmplib.org/`

arithmetic and large proof contexts. For some theorems (e.g., for the naive multiplication algorithm), we had to write several 100-line assertions, which defeats the point of using automated tools rather than an interactive theorem prover. Thus, we decided to put that experiment on hold, until we got a way to make the proof of these theorems straightforward.

When one wants to extends a theorem prover with new capabilities (e.g., an inference rule dedicated to the problem at hand), one way is to "incorporate a reflection principle, so that the user can verify within the existing theorem proving infrastructure that the code implementing a new rule is correct, and to add that code to the system" [9]. This article shows how we have modified Why3 to offer computational reflection. It was especially important to make the user process straightforward, so that reflection can be routinely used whenever external provers get lost. As an illustration, this paper shows how we made use of our approach to design and prove a decision procedure suitable for verifying GMP-like algorithms.

In Section 2, we illustrate computational reflection on the correctness of Strassen's algorithm for matrix multiplication in Why3. While straightforward to verify by hand, this algorithm already exceeds the capabilities of SMT solvers [5]. So we perform a reflection-based proof in a traditional way: we represent logical propositions about matrix polynomials by inductive objects, we define functions over these objects in the logical system, we prove some lemmas about them, and we use these functions and lemmas to prove the correctness property of Strassen's algorithm.

This approach does not require any modification to Why3 or to the external provers, but we have not yet explained how to reify logical propositions into inductive objects that can be manipulated inside Why3's logic. Section 3 shows how we have extended Why3 to do so.

Traditionally, computational reflection performs proofs by evaluating some pure terms occurring in logical propositions. Yet, Why3's programming language is much richer: mutable variables, arrays, exceptions, loops, and so on. Section 4.1 shows how the designer of decision procedures can benefit from the whole extent of WhyML. This required us to add a WhyML interpreter to Why3 (Section 4.2).

While the reification component does not extend the trusted computing base of Why3 at all, the interpreter does, albeit in a minimal way. We discuss the soundness of our approach in Section 5.

Given the ability to write decision procedures in WhyML, to verify them using Why3, and to execute them inside Why3 on reified logical propositions, we have all the tools to design and use a decision procedure dedicated to verifying GMP's algorithms. Section 6 presents this procedure. While it might look like a naive procedure for solving systems of linear equalities, the coefficients it manipulates are not simple rationals, they are products of rationals by powers with symbolic exponents, e.g., $-5/3 \cdot \beta^{i+j-2}$. These powers occur because we are proving the soundness of algorithms manipulating power series $\sum a_i \beta^i$.

This work is part of Why3 and the examples presented in this article are available at `http://toccata.lri.fr/gallery/reflection.en.html`.

2 Computational reflection proofs

When designing a decision procedure by reflection, one first finds an embedding of the propositions P of interest into the logical language of the formal system. Let us denote $\ulcorner P \urcorner$ the resulting term, e.g., the abstract syntax tree of P. Then one proves that, if $\ulcorner P \urcorner$ satisfies some property φ, then P holds. Thus, when one wants to prove that some proposition P holds, one just has to check that $\varphi(\ulcorner P \urcorner)$ does. If φ is designed so that $\varphi(\ulcorner P \urcorner)$ can be validated just by computations, then we have a proof procedure by computational reflection. This approach has been used in various contexts [9,8,1,3,4,5].

Let us illustrate this process on a toy example: the correctness of Strassen's matrix multiplication algorithm. Among other properties, one has to prove four matrix equalities such as the following one:

$$\mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} = \mathbf{M}_{1,1},$$

with

$$\begin{aligned}\mathbf{M}_{1,1} = {} & (\mathbf{A}_{1,1} + \mathbf{A}_{2,2}) \cdot (\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) + \mathbf{A}_{2,2} \cdot (\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\ & - (\mathbf{A}_{1,1} + \mathbf{A}_{1,2}) \cdot \mathbf{B}_{2,2} + (\mathbf{A}_{1,2} - \mathbf{A}_{2,2}) \cdot (\mathbf{B}_{2,1} + \mathbf{B}_{2,2}).\end{aligned}$$

By the group laws of matrix addition and by distributivity of matrix multiplication, one easily shows that the right-hand side of the equality can be turned into the left-hand side. Unfortunately, in practice, SMT solvers (Alt-Ergo, CVC4, Z3) and TPTP solvers (Eprover) fail to prove such a proposition. There are two reasons. First, a solver should instantiate the above algebraic laws on the order of one hundred times, assuming they apply them in an optimal way. Second, when verifying programs, the proof context is usually filled with hundreds of other instantiable theorems, which will delay applying the algebraic laws. As a consequence, unless an automated prover implements a dedicated decision procedure for this kind of property, there is no way its proof can be found.

Let us see how to supplement the lack of such a dedicated decision procedure. While this paper presents it in the context of Why3, the exact same process could be followed in any formal system with some computational capabilities.

2.1 Embedding terms

The first step is to embed $\mathbf{M}_{1,1}$ into the logical language of Why3. We define the following inductive type `t` to represent its abstract syntax tree:

```
type t = Var int | Add t t | Mul t t | Sub t t | Ext r t
```

Matrices appearing at the leaves of the expression (e.g., $\mathbf{A}_{2,1}$) are assigned a unique integer identifier and are represented using the `Var` constructor. The sum, product, and differences of two matrices, are represented using the constructors `Add`, `Mul`, and `Sub`. Finally, the `Ext` constructor represents the external product (by a value of type `r`), which is not needed in the case of Strassen's algorithm.

```
type vars = int → a
let rec function interp (x: t) (y: vars) : a =
  match x with
  | Var n → y n
  | Add x1 x2 → aplus  (interp x1 y) (interp x2 y)
  | Mul x1 x2 → atimes (interp x1 y) (interp x2 y)
  | Sub x1 x2 → asub   (interp x1 y) (interp x2 y)
  | Ext r x → ($) r (interp x y)
  end
```

Fig. 1. Interpreting the abstract syntax tree of a polynomial

Note that the function $\mathbf{M} \mapsto \ulcorner\mathbf{M}\urcorner$ cannot be expressed in the logical language, but its inverse can. We thus define a function that maps a term of type `t` into a matrix, as shown in Figure 1. That definition causes Why3 to create a recursive function `interp` inside the logical system, since its termination is visibly guaranteed by the structural decrease of its argument `x`.

When `aplus`, resp. `atimes`, is instantiated using matrix sum, resp. product, one can prove that the Why3 term

```
interp (Mul (Add (Var 0) (Var 1)) (Var 7)) y
```

is equal to $(\mathbf{A}_{1,1} + \mathbf{A}_{1,2}) \cdot \mathbf{B}_{2,2}$, assuming that `y` maps 0 to $\mathbf{A}_{1,1}$, 1 to $\mathbf{A}_{1,2}$, and 7 to $\mathbf{B}_{2,2}$. This proof can be done by unfolding the definition of `interp`, by reducing the `match with` constructs, and by substituting the applications of `y` by the corresponding results. Why3 provides a small rewriting engine that is powerful enough for such a proof, but one could also use an external prover.

2.2 Normalizing terms

Let us suppose that we now have two concrete expressions `x1` and `x2` of type `t` and a single map `y` of type `vars` and that we want to prove the following equality:

```
goal g: interp x1 y = interp x2 y
```

The actual value of `y` does not matter, but the facts that `aplus` is a group operation and that `amult` is distributive do. In other words, we want to see `x1` and `x2` as non-commutative polynomials and we want to prove that they have the same monomials with the same coefficients. To do so, let us turn them into weighted lists of monomials. Figure 2 shows an excerpt of the code. For example, the term $(\mathbf{A}_{1,1} + \mathbf{A}_{1,2}) \cdot \mathbf{B}_{2,2}$ gets turned into the list

```
Cons (M 1 (Cons 0 (Cons 7 Nil))) (Cons (M 1 (Cons 1 (Cons 7 Nil))) Nil)
```

Note that we have introduced a new interpretation function `interp'` and we have stated the postcondition of `conv` accordingly. Why3 requires us to prove that this postcondition holds. The proof is straightforward, even in the multiplication case. Once done, we obtain the following lemma in the context:

```
lemma conv_def: forall x y. interp x y = interp' (conv x) y
```

```
type m = M int (list int)
type t' = list m

let rec function interp' (x: t') (y: vars) : a =
  match x with
  | Nil → azero
  | Cons (M r m) l → aplus (($) r (mon m y)) (interp' l y)
  end

let rec function conv (x:t) : t'
  ensures { forall y. interp x y = interp' result y }
= match x with
  | Var v → Cons (M rone (Cons v Nil)) Nil
  | Add x1 x2 → (conv x1) ++ (conv x2)
  | Mul x1 x2 → ...
  end
```

Fig. 2. Converting a polynomial to a list of monomials

We define one last function, `norm`, which sorts a weighted list of monomials by insertion using a lexicographic order, merging contiguous monomials along the way. Its postcondition, once proved, leads to

```
lemma norm_def: forall x y. interp' x y = interp' (norm x) y
```

Note that we do not even need to prove that `norm` actually sorts the input list or that it merges monomials, so the proof is again trivial. If there is some bug in `norm`, it only endangers the completeness of the approach, not its soundness. For example, defining `norm` as the identity function would ultimately be fine but pointless.

By composing `norm` and `conv` and equality, we get our decision procedure φ dedicated to verifying Strassen's algorithm. Indeed, to prove the goal `g` above, we just need to prove the following intermediate lemma:

```
lemma g_aux: norm (conv x1) = norm (conv x2)
```

As with `interp` before, `norm` and `conv` are logic functions defined by induction on their argument, so there is no difficulty in proving `g_aux` using the rewriting engine of Why3 or an external automated prover.

2.3 Advantages

There are several advantages to this approach. The most important one is that the user can easily design a decision procedure dedicated to the problem at hand. Indeed, the inductive type for representing expressions does not have to handle the full extent of the language, but can focus on the constructions that matter (e.g., addition). Moreover, the soundness of the system is not endangered, since the user has to prove the correctness of the procedure (e.g., the lemmas

`conv_def` and `norm_def`). Finally, since the procedure is ad hoc, performances in the general case do not matter much, so one can write it so that both the code and the proof are straightforward. For instance, in the example above, the sorting algorithm has quadratic complexity and one only has to prove that the interpretation of the list is left unchanged. Thus, SMT solvers quickly discharge all the verification conditions that Why3 generates to guarantee that the implementation of the decision procedure satisfies its specification.

Even if this normalization procedure is dedicated to proving Strassen's algorithm, we took advantage of Why3's module system to make it generic: coefficients are in an arbitrary commutative ring and variables are in a (noncommutative) ring. Both rings are potentially different, as in the case of matrices. The genericity of the presented decision procedure does not extend to supporting variables in a commutative ring, but it is just a matter of duplicating the code of the decision procedure to modify the ordering relation, which we did.

A very similar reflection-based tactic is used by the Coq proof assistant to formally verify equalities in a commutative ring or semi-ring [8]. This tactic was implemented, part as an OCaml plugin for Coq, part in the meta-language Ltac of Coq. Rather than lists of monomials, that work uses Horner's representation of polynomials: $p_0 + x_1 \cdot (p_1 + x_1 \cdot (p_2 + x_1 \cdots))$ with $(p_i)_i$ being polynomials where variable x_1 does not occur.

3 Reification

We have not yet explained how one obtains the inductive objects used to instantiate the decision procedure. Without modifying Why3, it is up to the user to provide them. Even for an algorithm as simple to verify as Strassen's, the user might forfeit before finishing to translate all the terms of the algorithm.

3.1 Possible approaches

To circumvent this issue, the original Why3 proof of Strassen's algorithm uses a clever approach [5]. The type of matrices has been modified so that a matrix contains not only the values of its cells but also the normalized list of monomials representing all the operations performed to obtain the matrix. In other words, the decision procedure has been split and embedded into all the matrix operations and it is executed symbolically along them. The lists of monomials (and the operations to build them) are declared *ghost*, so they do not interfere with actual matrix computations and can be erased from the final algorithm, which is therefore still fundamentally the same. Nonetheless, this approach forces the user to instrument the matrix operations, and while these modifications are suitable to prove Strassen's algorithm, they might be useless when verifying another matrix algorithm, if not detrimental by polluting the proof context with all the symbolic computations.

Thus, for a reflection-based decision procedure to be useful, we have to provide some ways to automate the *reification* process, that is, the conversion of expressions into their inductive representation.

As mentioned above, one difficulty lies in defining ⌜⌝, which is an inverse function of `interp`. This inverse is usually written using the meta-language of a formal system to parse the term and to produce the corresponding inductive object. Since Why3 can load plugins written in OCaml, one could certainly use OCaml as a meta-language for Why3. This unfortunately requires the user to learn the inner workings of Why3.

Another possibility would be to use WhyML as a meta-language by providing some primitives to visit the abstract syntax trees of expressions and by making Why3 able to interpret it. As is the case for other formal systems [13,6], any WhyML function using such primitives would no longer be meaningful for the remainder of the logical system, so as to avoid inconsistencies. The user would thus no longer need to leave the confines of WhyML, but this is still not completely satisfactory. Indeed, as written before, ⌜⌝ is the inverse of `interp`, so any explicit definition seems superfluous. Instead, we could have some OCaml code, e.g., a Why3 plugin, that inverts `interp` on the fly.

3.2 Inversion of the interpretation function

Consider the following function, which is just a variant of the decision procedure for Strassen's algorithm:

```
let norm_f (x1 x2: t) : bool
  ensures { forall y:vars. result = True → interp x1 y = interp x2 y }
= match norm (conv (Sub x1 x2)) with
  | Nil → True (* the difference evaluates to the empty polynomial *)
  | _ → False
  end
```

Whenever the user wants to use this decision procedure to prove a goal, we would like Why3 to automatically find `x1`, `x2`, and `y`, so that the right-hand side of the post-condition matches the goal. This is done by a straightforward recursive walk of the goal. Let us illustrate this walk with `foo a + b = c`. This goal is an equality, and so is the right-hand side of the postcondition of `norm_f`, so Why3 proceeds recursively on each side of the equality. The left-hand side starts with an addition, while there is an application of `interp` in the postcondition, so Why3 assumes that `interp` is an interpretation function.

This function starts with a pattern matching on its first argument, so Why3 looks at all of the branches. The second branch starts with an addition (i.e., `aplus`, which we assume was instantiated with `+`). So Why3 registers that `x1` should start with the constructor `Add`. And so on, recursively. Eventually, Why3 has to match `foo a` against a branch. None of them matches, but the one for the `Var` constructor returns `y n`, with `y` a variable of type *arrow*. So Why3 selects a fresh integer for `n`, e.g., 0, and remembers that it should choose `y` so that it maps 0 to `foo a`.

3.3 Extensions

The previous process works fine when a goal has to be proved in isolation, irrespective of the proof context. To remove this limitation, Why3 also recognizes the presence of an implication inside a branch of an interpretation function. In that case, it tries to match a hypothesis of the proof context against the left-hand side of the implication, and it does so recursively until all the hypotheses of the context have been tried. The following functions illustrate this behavior. They serve as interpretation functions of a decision procedure that needs to consider all the equalities from the proof context. In this example taken from the verification of GMP's algorithms, the fact that the goal also has to be an equality is a coincidence.

```
function interp_eq (g:equality) (y:vars) (z:C.cvars) : bool
= match g with (g1, g2) → interp g1 y z = interp g2 y z end

function interp_ctx (l:list equality) (g:equality) (y:vars) (z:C.cvars)
      : bool
= match l with
  | Nil → interp_eq g y z (* goal *)
  | Cons h t → (interp_eq h y z) → (interp_ctx t g y z)
  end
```

Notice that, since Why3's logical system does not permit functions returning logical propositions, we have defined these interpretation functions as returning Boolean values. But this has no impact on the way reification proceeds.

While the decision procedures presented in this paper ignore quantified formulas, our reification transformation does support them. For example, the excerpt below would handle universal quantifiers in a nameless fashion, using negative indices to store the depth of the quantifier:

```
function interp_fmla (f:fmla) (l:int) (b:vars) : bool
= match f with
  | Forall f' → forall v. interp_fmla f' (l-1) b[l ← v]
  | ...
  end
```

A current limitation of our approach is the purely syntactic nature of the reification step. For example, for an uninterpreted function `foo`, the terms `foo (a+b)` and `foo (b+a)` are mapped to distinct variables, even though they are provably equal. This requires a significant amount of extra work from the user. However, we are optimistic that this can be mitigated either in the reification step itself or by composition with another decision procedure (as shown in Section 6.3).

4 Effectful decision procedures

Computations in the reflection-based proof from Section 2 are all done in logic functions, which are unfolded by automated provers or Why3's rewriting engine. A limitation of this approach is that Why3's language of logic functions is not

very expressive, as they must be side effect-free and their termination must be guaranteed by a structurally decreasing argument.

In this section, we show how we can instead write decision procedures as regular WhyML programs, making full use of the language's imperative features such as loops, references, arrays, and exceptions. These decision procedures are proved correct using Why3 and some automated theorem provers. Their contract can then be instantiated by reification of the goal and context, and used as a cut indication.

4.1 Running example: systems of linear equalities

As an example, let us consider a decision procedure for linear equation systems in an arbitrary field (code excerpts in Figure 3). Given some assumed-valid linear equalities in the context, the procedure attempts to prove a linear equality by showing that it is a linear combination of the context.

This is done by representing the context and goal by a matrix and performing a Gaussian elimination (function `gauss_jordan`). In case of success, we obtain a vector of coefficients and we check whether the corresponding linear combination of the context is equal to the goal (function `check_combination`). Otherwise, the procedure returns `False` and proves nothing, since its postcondition has `result = True` as premise.

As is done in Coq with the tactics `lia` and `lra` [1], this is a proof by certificate, since we check if the linear combination of the context returned by `gauss_jordan` matches the goal. There is no need to prove the Gaussian elimination algorithm itself, nor to define a semantics for the matrix passed to it as a parameter. In fact, we do not prove anything about the content of any matrix in the program. This makes the proof of the decision procedure very easy in relation to its length and intricacy.

Let us now examine the contract of the decision procedure. The postcondition states that the goal holds if the procedure returns `True`, for any valuations `y` and `z` of the variables such that the equalities in the context hold. The `valid_ctx` and `valid_eq` preconditions state that the integers used as variable identifiers (second argument of the `Term` constructor) in the context and goal are all non-negative. This is needed to prove the safety of array accesses. The nature of the reification procedure ensures that these preconditions will always be true in practice, but as reification is not trusted, the user has to verify them explicitly; SMT solvers do this very easily. Finally, the `raises` clause expresses that an exception may escape the procedure (typically an arithmetic error, as we allow the field operations to be partial). In that case, nothing is proven.

Notice that the decision procedure is independent from Why3 (apart from the fact that it is formally verified), in the sense that it does not contain meta-instructions for reification or anything linked to Why3 internals. One could easily imagine finding the same kind of code in an automatic prover.

```
clone LinearEquationsCoeffs as C with type t = coeff

type expr = Term coeff int | Add expr expr | Cst coeff
type equality = (expr, expr)

let linear_decision (l: list equality) (g: equality) : bool
  requires { valid_ctx l }
  requires { valid_eq g }
  ensures { forall y z. result = True → interp_ctx l g y z }
  raises  { C.Unknown → true }
= ...
  fill_ctx l 0;
  let (ex, d) = norm_eq g in
  fill_goal ex;
  let ab = m_append a b in
  let cd = v_append v d in
  match gauss_jordan (m_append (transpose ab) cd) with
    | Some r → check_combination l g (to_list r 0 ll)
    | None → False
  end
```

Fig. 3. Decision procedure for linear equation systems

4.2 Interpreter

Due to their side effects, functions from WhyML programs only have abstract declarations in the logical world (as opposed to the concrete logic functions used in Section 2). Therefore, they cannot be unfolded by automatic provers or by Why3's rewriting engine. In order to compute the results of decision procedures such as the previous one, we have added an interpreter to Why3. It operates on an ML-like intermediate language that corresponds to WhyML programs from which logic terms, assertions, and ghost code, were erased, thereby assuming that the program was proved beforehand and that the preconditions are met. This intermediate code is produced by the existing extraction mechanism, which is used to produce OCaml and C programs from proved WhyML programs.

Our interpreter provides built-in implementations for some axiomatized parts of the Why3 standard library, such as integer arithmetic and arrays. For performances purposes, we also chose to implement references as a builtin rather than interpret their WhyML definition (records with a single mutable field), in order to reduce the number of indirections. To ease debugging decision procedures, we have added to Why3's standard library a `print` function of type `'a → unit` and without effects. It is interpreted as a polymorphic `printf` function.

There have been few works on computational reflections using effectful decision procedures. One may cite Claret *et al* [4]. They use a monadic encoding of effectful computations in Coq (e.g., non-termination). Monadic decision procedures are turned into impure programs that are executed outside of Coq. The

result of these external computations is used as a "prophecy" to simulate the execution of the decision procedure inside of Coq. Since we are working with Why3, which natively supports impure computations, we sidestep the need for a heavyweight simulation mechanism.

5 Soundness

The implementation of our framework requires two additions to Why3: a reification transformation and an interpreter of WhyML programs. Let us discuss the soundness of our approach.

First, the rather large and intricate code needed for reification is not part of the trusted computing base of Why3. Indeed, the reification merely guesses values for all the relevant variables and asks Why3 to instantiate the contract of the decision procedure with them. Assuming the user has proved the soundness of the decision procedure, this instantiated proposition holds, whether the reification algorithm is correct or not. A reification failure would either prevent a well-typed instantiation of the post-condition, or the resulting cut would be useless for proving the current goal.

Contrarily to the reification code, our interpreter is part of the trusted computing base. Fortunately, it is very simple, since it only manipulates concrete values. There is no need for partial evaluation nor symbolic execution nor polymorphic equality, which makes this new interpreter much simpler than the existing rewriting engine. Another reason for its simplicity is that the intermediate language has relatively few constructions, since program transformations performed by the existing extraction mechanism eliminate potentially confusing behaviors from the surface language such as parallel assignation.

6 Application: GMP

In this section, we briefly present our verified multiprecision library [12] and show how we eliminated a large number of assertions by implementing a dedicated reflection-based decision procedure.

6.1 A GMP function

In GMP, natural integers are represented as little-endian buffers of unsigned machine integers called *limbs*. We set a radix β (typically 2^{64}). Any natural number N has a unique radix-β decomposition $\sum_{k=0}^{n-1} a_k \beta^k$, which is represented as the buffer $a_0 a_1 \dots a_{n-1}$.

In the low-level functions, there is almost no memory management; operands are specified as pointers to their least significant limb and a size of type `int32`. Given such a pointer `a` and a size `n`, provided the pointer is valid over the size `n`, we denote

$$\texttt{value a n} = \overline{a_0 \dots a_{n-1}} = \sum_{k=0}^{n-1} a_k \beta^k.$$

```
1  (** [add_limbs r x y sz] adds [x[0..sz-1]] and [y[0..sz-1]]
2      and writes the result in [r].
3      Returns the carry, either [0] or [1].
4      Corresponds to the function [mpn_add_n]. *)
5
6  let add_limbs (r x y: ptr limb) (sz: int32) : limb
7    ensures { 0 ≤ result ≤ 1 }
8    ensures { value r sz + (power radix sz) * result =
9              value x sz + value y sz }
10 =
11   let limb_zero = Limb.of_int 0 in
12   let i = ref (Int32.of_int 0) in
13   let lx = ref limb_zero in
14   let ly = ref limb_zero in
15   let c = ref limb_zero in
16   while Int32.(<) !i sz do
17     invariant { value r !i + (power radix !i) * !c =
18                 value x !i + value y !i }
19     lx := get_ofs x !i;
20     ly := get_ofs y !i;
21     let res, carry = add_with_carry !lx !ly !c in
22     set_ofs r !i res;
23     c := carry;
24     assert { value r (!i+1) + (power radix (!i+1)) * !c
25              = value x (!i+1) + value y (!i+1)
26              by value r (!i+1) + (power radix (!i+1)) * !c
27                = value r !i + (power radix !i) * res
28                  + (power radix !i) * c
29                = ... (* 10+ subgoals *) };
30     i := Int32.(+) !i (Int32.of_int 1);
31   done;
32   !c
```

Fig. 4. Addition of two integers

As an example, Figure 4 shows the function that adds two natural integers of identical limb count. Part of the specification and most invariants and assertions have been omitted for readability. The algorithm is the schoolbook addition: starting from the least significant limb, the input numbers are added limb by limb, keeping track of the carry.

Unfortunately, even such a simple algorithm somewhat stumps the SMT solvers. In order to prove the loop invariant, we needed the assertion at line 24. Its proof consists in a sequence of about ten rather simple steps (rewrite an equality in the context, use distributivity, etc.) but the large search space prevents the automatic provers from succeeding. Therefore, we had to provide many cut indications by hand using the `by` construct.

Yet, with a judicious choice of coefficients, this goal (and many others in the proofs of our library) can be seen as a linear combination of the context. Therefore, we should be able to use the decision procedure from Section 4 to prove the assertion in one go.

6.2 Coefficients

The following is a simplified version of the context and goal obtained for the assertion of the main loop of `add_limbs` (Figure 4, line 24). Variables `r1` and `c1` denote the values of `r` and `c` at the start of the loop (before the modifications that occur at lines 22 and 23).

```
axiom H:  value r1 i + (power radix i) * c1 = value x i + value y i
axiom H1: res + radix * c = lx + ly + c1
axiom H2: value r i = value r1 i
axiom H3: value x (i+1) = value x i + (power radix i) * lx
axiom H4: value y (i+1) = value y i + (power radix i) * ly
axiom H5: value r (i+1) = value r i + (power radix i) * res
goal g:   value r (i+1) + power radix (i+1) * c
          = value x (i+1) + value y (i+1)
```

Notice that the linear combination $\texttt{H5} - \texttt{H4} - \texttt{H3} + \texttt{H2} + \beta^{\texttt{i}} \cdot \texttt{H1} + \texttt{H}$ simplifies to an equality equivalent to `g`. In order to prove this, our decision procedure has to include powers of β (`radix` in the WhyML code) in its coefficients, and to support symbolic exponents (as `i` is a variable).

More precisely, the coefficients of our decision procedure are the product of a rational number and a (symbolic) power of β. Figure 5 is an excerpt of the WhyML implementation of the coefficients. The decision procedure of Figure 3 is instantiated with `type coeff = t`.

One can define addition, multiplication, and multiplicative inverse over these coefficients. Addition is partial, since one may only add two coefficients with equal exponents. If this is not the case, the addition raises an exception, which is accounted for in the specification of the decision procedure (exception `C.Unknown` in Figure 3). Note that exponents do not have to be structurally equal, only to have equal `interp_exp` interpretations for all values of `y`, which can be automatically proved within the decision procedure.

6.3 Modular decision procedures

The coefficients above are expressive enough to prove assertions such as the one in Figure 4. However, notice that their interpretation (function `interp` in Figure 5) is expressed in terms of real numbers (this is needed because the Gaussian elimination algorithm used in the decision procedure needs to compute the multiplicative inverse of some coefficients), while the context and goal consist in equalities over integers. Moreover, the inductive type for expressions that is used in the decision procedure (type `expr` in Figure 3) is quite restrictive, which avoids repetitions in the code of the decision procedure. However, this is

```
type exp = Lit int | Var int | Plus exp exp | Minus exp | Sub exp exp
type rat = (int, int)
type t = (rat, exp)

function qinterp (q:rat) : real
= match q with (n,d) → from_int n /. from_int d end

function interp_exp (e:exp) (y:vars) : int
= match e with
  | Lit n → n
  | Var v → y v
  | Plus e1 e2 → interp_exp e1 y + interp_exp e2 y
  | Sub e1 e2 → interp_exp e1 y - interp_exp e2 y
  | Minus e' → - (interp_exp e' y)
  end

function interp (t:t) (y:vars) : real
= match t with
  (q,e) → qinterp q *. pow radix (from_int (interp_exp e y))
  end
```

Fig. 5. Definition of the coefficients

problematic for the user, since a term such as `2 * 3 * x` cannot be reified by inversion of `interp`.

These constraints can be lifted thanks to an approach similar to the `conv` function in Section 2. We compose the decision procedure `linear_decision` with a function that converts integer-valued coefficients to real-valued coefficients, and a function that converts from a more expressive expression type to the `expr` type (code excerpts in Figure 6).

The conversion procedure from integer-valued to real-valued coefficients is only sound when the exponents of β are nonnegative. This is always the case for GMP algorithms. Due to the symbolic exponents, it is not yet possible to automatically prove this property within the decision procedure, so we instead add it as an extra precondition (the `pos_*` predicates in `mp_decision`). In practice, SMT solvers prove it easily.

While the final decision procedure is specialized for GMP goals, almost all the reasoning is done in the generic linear decision procedure `linear_decision`, which we did not modify at all. We expect that, for other use cases than GMP, users will also be able to develop their own interpretation and conversion layers and reuse the primary linear decision procedure as is.

7 Conclusion

This paper presents two contributions. First, we have developed a framework for proofs by reflection that uses effectful WhyML programs as decision procedures.

```
let decision (l:list equality') (g:equality') : bool
  requires { valid_ctx' l ∧ valid_eq' g }
  ensures { forall y z. result = True → interp_ctx' l g y z }
  raises  { Unknown → true }
= let sl, sg = simp_ctx l g in
  linear_decision sl sg

let mp_decision (l: list equality'') (g: equality'') : bool
  requires { valid_ctx'' l ∧ valid_eq'' g }
  ensures  { forall y z. result = True → pos_ctx'' l z → pos_eq'' g z
             → interp_ctx'' l g y z }
  raises  { Unknown → true }
= decision (m_ctx l) (m_eq g)
```

Fig. 6. Composition of decision procedures

Second, we have implemented and verified a procedure for automatically solving systems of linear equalities with symbolic coefficients. We have used this decision procedure to prove many goals throughout our formalization of GMP algorithms.

As a point of comparison, we have revisited all our existing proofs of addition, subtraction, and multiplication algorithms, which previously required numerous user-supplied assertions. The decision procedure was able to discharge all the large assertions (in the vein of Figure 4, line 24). This section of our library was previously about 1660 lines long. The 660 lines of program code were obviously left unchanged, but the 1000 lines of specifications and proofs were halved. Moreover, a large part of the remaining 500 lines consists in function contracts and loop invariants, which are essentially incompressible.

The hardest goal we have successfully used our decision procedure on (an assertion in the proof of the generic-case long division) involves Gaussian elimination on a matrix of size about 150×90, and it terminates in about 3 seconds, which is acceptable from a user-experience standpoint. Should larger matrices become problematic, one option to improve performance would be, instead of using a WhyML interpreter, to extract the decision procedure to OCaml and execute the resulting binary.

Note that while our decision procedure only deals with linear equation systems, we have successfully used it to prove goals in the proofs of multiplication, division, and logical shifts that, at first glance, are completely nonlinear. In these cases, we had to supply one or two cut indications that took care of the nonlinear part of the reasoning, but this is very acceptable considering that many of these goals previously required more than fifty user-supplied cut indications each. We are optimistic that this new tool will allow us to verify new GMP algorithms much more efficiently than we used to.

The approach presented in this article is not limited to Why3 in principle. All that is required to develop a similar framework is the capability to specify and prove the correctness of decision procedures, and the capability to execute

verified programs. As such, it would likely not take much work to adapt our framework to verification platforms such as Leon [2] and Dafny [10]. For example, Leon is already able to compile ground terms to Java bytecode and execute them.

References

1. Besson, F.: Fast reflexive arithmetic tactics the linear case and beyond. In: Altenkirch, T., McBride, C. (eds.) International Workshop on Types for Proofs and Programs. Lecture Notes in Computer Science, vol. 4502, pp. 48–62. Nottingham, UK (2007)
2. Blanc, R.W., Kneuss, E., Kuncak, V., Suter, P.: An overview of the Leon verification system: Verification by translation to recursive functions. In: 4th Annual Scala Workshop (2013)
3. Chaieb, A., Nipkow, T.: Proof synthesis and reflection for linear arithmetic. Journal of Automated Reasoning 41(1), 33–59 (2008)
4. Claret, G., González Huesca, L.d.C., Régis-Gianas, Y., Ziliani, B.: Lightweight proof by reflection using a posteriori simulation of effectful computation. In: 4th International Conference on Interactive Theorem Proving. Lecture Notes in Computer Science, vol. 7998, pp. 67–83. Springer (Jul 2013)
5. Clochard, M., Gondelman, L., Pereira, M.: The Matrix reproved. Journal of Automated Reasoning 60(3), 365–383 (2017)
6. Ebner, G., Ullrich, S., Roesch, J., Avigad, J., de Moura, L.: A metaprogramming framework for formal verification. In: 22nd ACM SIGPLAN International Conference on Functional Programming. pp. 34:1–34:29. Oxford, UK (Sep 2017)
7. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) 22nd European Symposium on Programming. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (Mar 2013)
8. Grégoire, B., Mahboubi, A.: Proving equalities in a commutative ring done right in Coq. In: Hurd, J., Melham, T. (eds.) 18th International Conference on Theorem Proving in Higher Order Logics. pp. 98–113. Oxford, UK (Aug 2005)
9. Harrison, J.: Metatheory and reflection in theorem proving: A survey and critique. Tech. Rep. CRC-053, SRI International Cambridge Computer Science Research Centre (1995)
10. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR-16. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010)
11. Moller, N., Granlund, T.: Improved division by invariant integers. IEEE Transactions on Computers 60(2), 165–175 (2011)
12. Rieu-Helft, R., Marché, C., Melquiond, G.: How to get an efficient yet verified arbitrary-precision integer library. In: 9th Working Conference on Verified Software: Theories, Tools, and Experiments. Lecture Notes in Computer Science, vol. 10712, pp. 84–101. Heidelberg, Germany (Jul 2017)
13. Ziliani, B., Dreyer, D., Krishnaswami, N.R., Nanevski, A., Vafeiadis, V.: Mtac: A monad for typed tactic programming in Coq. Journal of Functional Programming 25 (2015)