HAL
open science

Formalizing Semantics with an Automatic Program Verifier

Martin Clochard, Jean-Christophe Filliâtre, Claude Marché, Andrei Paskevich

To cite this version:

Martin Clochard, Jean-Christophe Filliâtre, Claude Marché, Andrei Paskevich. Formalizing Semantics with an Automatic Program Verifier. 6th Working Conference on Verified Software: Theories, Tools and Experiments (VSTTE), Jul 2014, Vienna, Austria. hal-01067197

HAL Id: hal-01067197

https://inria.hal.science/hal-01067197

Submitted on 23 Sep 2014

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.

Formalizing Semantics with an Automatic Program Verifier*

Martin Clochard[1,2,3], Jean-Christophe Filliâtre[2,3], Claude Marché[3,2], and Andrei Paskevich[2,3]

[1] Ecole Normale Supérieure, Paris, F-75005
[2] Lab. de Recherche en Informatique, Univ. Paris-Sud, CNRS, Orsay, F-91405
[3] INRIA Saclay – Île-de-France, Orsay, F-91893

Abstract. A common belief is that formalizing semantics of programming languages requires the use of a *proof assistant* providing (1) a specification language with advanced features such as higher-order logic, inductive definitions, type polymorphism, and (2) a corresponding proof environment where higher-order and inductive reasoning can be performed, typically with user interaction.
In this paper we show that such a formalization is nowadays possible inside a mostly-automatic program verification environment. We substantiate this claim by formalizing several semantics for a simple language, and proving their equivalence, inside the Why3 environment.

1 Introduction

Interactive proof environments, also known as *proof assistants*, are tools of choice to formalize the semantics of programming languages and, more generally, to handle abstract syntax. For example, the project CompCert [1], formalizes in Coq [2] the semantics of a very large subset of the C programming language and verifies an optimizing compiler. Similarly, the L4.verified project [3] develops a verified operating system on top of a formalization of C in Isabelle/HOL [4], ACL2 [5] is used to formalize a Java virtual machine [6], and a complete semantics of JavaScript is formalized using Coq [7].

Typically, the formalization of a programming language makes heavy use of advanced logic constructs such as algebraic data types (e.g. to represent abstract syntax trees), inductive definition of predicates (e.g. to encode inference rules for typing or evaluation judgments), higher-order functions, dependent types, etc. Proving results on such a formalization (e.g. type soundness) thus typically involves complex reasoning steps, making use of the interactive tactics that a proof assistant provides for induction, case analysis, etc.

Automatic program verifiers aim at providing dedicated environments to verify behavioral properties of programs written in some specific programming language. Examples are Dafny [8], VeriFast [9], Frama-C [10], or SPARK [11]. They

* Work partly supported by the Bware project (ANR-12-INSE-0010, http://bware.lri.fr/) and the Joint Laboratory ProofInUse (ANR-13-LAB3-0007, http://www.spark-2014.org/proofinuse) of the French national research organization

differ from proof assistants in two ways. First, they allow one to verify programs written in some imperative language whereas proof assistants rely on purely functional programming. Second, the main technique for proving properties is via the use of automatic theorem provers, such as SMT solvers. The high level of proof automation in such environments is possible because the specification languages they propose are less expressive than those of proof assistants. They are typically limited to some flavor of first-order logic.

Why3 is a program verifier developed since 2011 [12] that proposes a rich specification language, which extends first-order logic with type polymorphism, algebraic data types, and inductive predicates [13]. Recently, it was augmented with some support for higher-order logic. It is thus a good candidate for experimenting in formalizing semantics using fully automated theorem proving. In this paper, we report on such an experiment. We consider exercises related to defunctionalization [14] proposed by O. Danvy at the JFLA'2014 conference. Along the presentation of our solution, we expose the techniques used inside Why3 to encode complex logic constructs into the logic of SMT solvers.

This paper is organized as follows. Section 2 defines a direct, big-step style semantics of a minimal arithmetic language; meanwhile we explain how we deal with algebraic data types, recursive functions, and pattern matching. Then Section 3 introduces another definition in continuation-passing style (CPS) and proves it equivalent to the first one; meanwhile we explain how we encode higher-order logic. Section 4 defunctionalizes the CPS code to get a first-order interpreter; meanwhile we explain how we deal with inductive predicates and complex termination proofs.

The complete solution in Why3, including the full source code and a dump of the proof results, is available at http://toccata.lri.fr/gallery/defunctionalization.en.html.

2 Direct Semantics

Our running example is a very simple toy language of arithmetic expressions, limited to literals and subtraction. The grammar for this language is as follows.

$$\begin{array}{lll} n & : & \mathit{int} \quad \text{integer constants} \\ e & : & \mathit{expr} \quad \text{expressions} \\ e & ::= & n \mid e - e \\ p & : & \mathit{prog} \quad \text{programs} \\ p & ::= & e \end{array}$$

Such abstract syntax trees are formalized with recursive algebraic data types. In Why3 such types are declared using an ML-like syntax:

```
type expr = Cte int | Sub expr expr
type prog = expr
```

Constructors `Cte` and `Sub` are used as regular function symbols. For instance, we can define an abstract syntax tree `p3` for the expression $(7-2)-(10-6)$ as follows:

```
function eval_0 (e: expr) : int =
  match e with
  | Cte n → n
  | Sub e1 e2 → eval_0 e1 - eval_0 e2
  end
```

Fig. 1. Direct, big-step semantics

```
constant p3: prog = Sub (Sub (Cte 7) (Cte 2)) (Sub (Cte 10) (Cte 6))
```

This language is given a big-step operational semantics, with inference rules defining a judgment $e \Rightarrow n$ meaning that expression e evaluates to the value n.

$$\frac{}{n \Rightarrow n} \qquad \frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2 \quad n_1 - n_2 = n_3}{e_1 - e_2 \Rightarrow n_3}$$

Since this semantics is deterministic and total, the simplest way is to encode it in Why3 as a logic function, as shown on Fig 1. This definition is recursive and makes use of pattern matching on its argument `e`. It is worth pointing out that, for any logic function definition, Why3 statically checks that it defines a total function: recursion must be on structurally smaller arguments and pattern matching must be exhaustive [13].

2.1 Interpreting and Proving

Before doing any proof, we can make some quick tests to check our definitions. For instance, we can define `v3` as the application of `eval_0` to `p3`

```
constant v3: int = eval_0 p3
```

and then ask Why3 to evaluate its value:

```
> why3 defunctionalization.mlw --eval v3
Evaluation of v3: 1
```

Of course, we can also *prove* that the value of `p3` is 1. We first state it:

```
goal eval_p3: v3 = 1
```

Then we ask Why3 to run the Alt-Ergo SMT solver [15] on this goal:

```
> why3 -P alt-ergo defunctionalization.mlw
defunctionalization.mlw eval_p3 : Valid (0.02s)
```

It is discharged in no time.

2.2 Encoding Algebraic Data Types and Pattern Matching

Proving such a lemma with the external provers available within Why3 requires encoding the constructs of algebraic data types and pattern matching into the logic of those provers.

The encoding is quite simple: the algebraic type is treated as an abstract type, and constructor symbols are treated as uninterpreted functions whose semantics is established via a number of axioms. Let us see how the `expr` type is transformed. First come the type and the constructors:

```
type expr
function Cte (n: int) : expr
function Sub (e1 e2: expr) : expr
```

Then Why3 defines a polymorphic selector function, which will later help us to encode pattern-matching constructions inside terms:

```
function match_expr (e: expr) (b1 b2: 'a) : 'a

axiom match_expr_Cte : forall b1 b2: 'a, n: int.
  match_expr (Cte n) b1 b2 = b1

axiom match_expr_Sub : forall b1 b2: 'a, e1 e2: expr.
  match_expr (Sub e1 e2) b1 b2 = b2
```

Here, `'a` denotes a type variable which can be instantiated with any type. Why3 natively supports ML-style prenex polymorphism and employs monomorphic instantiation with symbol discrimination, preservation of interpreted types, and lightweight polymorphism encodings [16,17] to efficiently translate polymorphic formulas for provers supporting only multi-sorted or mono-sorted logic.

The definition of `match_expr` suffices to prove that the ranges of `Cte` and `Sub` are disjoint. However, most SMT solvers will not deduce this fact automatically, which is why we also define an index function:

```
function index_expr (e: expr) : int

axiom index_expr_Cte : forall n: int [Cte n].
  index_expr (Cte n) = 0

axiom index_expr_Sub : forall e1 e2: expr [Sub e1 e2].
  index_expr (Sub e1 e2) = 1
```

The terms in square brackets that appear before the quantified expressions in these axioms are *instantiation patterns*, also known as *triggers*. First-order SMT solvers accept triggers as hints to control instantiation of universally quantified premises. A trigger prescribes to an SMT solver to produce a new instance whenever a term that matches the trigger occurs in the set of currently assumed facts. The precise semantics of triggers varies among solvers; in particular, a solver may ignore the patterns given by a user, invent its own triggers, or use an instantiation method not based on triggers, e.g. paramodulation.

In the definition of `index_expr`, the triggers are given to force the index calculation for every occurrence of a constructor, which is enough to deduce that no application of `Cte` can be equal to an application of `Sub`, provided that the prover supports integers and knows that 0 is not equal to 1. If we target a prover that has no support for integer arithmetic, Why3 does not generate the `index_expr` function and produces instead the axioms of pairwise dis-equality of constructors:

```
axiom Cte_Sub : forall n: int [Cte n]. forall e1 e2: expr [Sub e1 e2].
  Cte n ≠ Sub e1 e2
```

The index function is preferable, however, because an axiom like `Cte_Sub` would be instantiated for any two occurrences of constructors, producing a quadratic number of instances.

Then we introduce constructor elimination functions, or *projections*:

```
function Cte_proj_1 (e: expr) : int
function Sub_proj_1 (e: expr) : expr
function Sub_proj_2 (e: expr) : expr

axiom Cte_proj_1_def : forall n: int. Cte_proj_1 (Cte n) = n
axiom Sub_proj_1_def : forall e1 e2: expr. Sub_proj_1 (Sub e1 e2) = e1
axiom Sub_proj_2_def : forall e1 e2: expr. Sub_proj_2 (Sub e1 e2) = e2
```

And finally, we close the `expr` type with the inversion axiom:

```
axiom expr_inversion : forall e: expr.
  e = Cte (Cte_proj_1 e)  ∨  e = Sub (Sub_proj_1 e) (Sub_proj_2 e)
```

Notice that this translation is not complete. Indeed, we do not state (nor can we in a first-order language) that `expr` is the least fixed point of constructor application: our translation could be applied to a co-inductive type just as well. This is not an important shortcoming, since the automated provers we target do not perform reasoning by induction, and therefore do not need this property. We will see below how proofs by induction can be constructed directly in Why3.

Now, let us show how pattern-matching expressions are translated. In simple cases, such as function definition by pattern matching, Why3 just splits the premise into several instances. Thus, the definition of `eval_0` turns into the following conjunction:

```
function eval_0 (e: expr) : int

axiom eval_0_def :
  (forall n: int. eval_0 (Cte n) = n) ∧
  (forall e1 e2: expr. eval_0 (Sub e1 e2) = eval_0 e1 - eval_0 e2)
```

In the general case, a `match-with` expression in a formula is translated as a conjunction of implications. For example, a Why3 formula

```
match e with
| Cte n → n > 0
| Sub e1 e2 → eval_0 e1 > eval_0 e2
end
```

```
function eval_1 (e: expr) (k: int → 'a) : 'a =
  match e with
  | Cte n → k n
  | Sub e1 e2 →
      eval_1 e1 (\ v1. eval_1 e2 (\ v2. k (v1 - v2)))
  end

function interpret_1 (p : prog) : int = eval_1 p (\ n. n)
```

Fig. 2. Semantics encoded in CPS form

becomes

```
(forall n: int. e = Cte n → n > 0) ∧
(forall e1 e2: expr. e = Sub e1 e2 → eval_0 e1 > eval_0 e2)
```

When pattern matching happens in a term, we resort to the selector function. Here is how the `match-with` expression in the definition of `eval_0` is translated:

```
match_expr e (Cte_proj_1 e)
 (eval_0 (Sub_proj_1 e) - eval_0 (Sub_proj_2 e))
```

Like ML or Coq, Why3 admits complex patterns, containing wildcards and or-patterns. These are compiled internally into nested `match-with` expressions over simple patterns [18].

3 Continuation-Passing Style

The next exercise is to CPS-transform the function `eval_0`. It amounts to adding a second argument, the *continuation*, which is a function k to be applied to the result. This trick allows us to replace any recursive call in the body of a program f by a tail call: $C[f\ x]$ becomes $f\ x\ (\lambda v.\ C\ v)$. Of course, this can be done only if the language supports higher-order functions. Such a support was added to Why3 recently.

For the function `eval_0`, we obtain the function `eval_1` in Fig. 2. The new interpreter, `interpret_1`, calls `eval_1` with the identity continuation.

3.1 Soundness Lemma

The soundness of our CPS-transformed interpreter can be expressed by two lemmas relating the CPS-based semantics with the direct big-step semantics, one for `eval_1` and one for `interpreter_1`. The first lemma must naturally be generalized: for any continuation k, `eval_1 e k` returns the same result as the application of k to the direct value of e.

```
lemma cps_correct_expr:
  forall e: expr, k: int → 'a. eval_1 e k = k (eval_0 e)
```

The second lemma is an easy consequence of the first one, applied to the identity continuation.

```
lemma cps_correct: forall p. interpret_1 p = interpret_0 p
```

Proving the first lemma is not easy. It cannot be proved as such by SMT solvers, since it requires induction on e. Why3 provides two ways to perform such an induction.

Structural induction by a dedicated transformation. The first way is to apply a *Why3 transformation*, that is a kind of proof tactic directly coded in Why3's kernel, that reduces a given goal into one or several sub-goals that, all together, imply the original goal. Resulting sub-goals can in turn be solved by external provers, or subject to another transformation, etc.

A Why3 transformation is devoted to structural inductive reasoning on algebraic data-types, producing one sub-goal for each constructor of the considered data type. For our particular lemma, a heuristics selects structural induction on e, as the goal involves a function defined recursively over e, namely `eval_1`. Then the proof is easily completed using SMT solvers on the sub-goals. Using such a dedicated transformation to add support for induction in an automated verifier is originally due to Leino [19].

General induction via lemma functions. The second way to perform reasoning by induction, that would work for general recursion, is to use *lemma functions*. This concept, that should be classified as folklore, was first emphasized in the VeriFast verifier [9], where some inductive reasoning is required in the context of representations in separation logic. Such a notion also exists in Dafny [8], and now in Why3. It is based on the idea that a recursive program can be used to simulate induction, as soon as it is terminating and side-effect free. In Why3, such a lemma function is written as follows:

```
let rec lemma cps_correct_expr (e: expr) : unit
  variant { e }
  ensures { forall k: int → 'a. eval_1 e k = k (eval_0 e) }
= match e with
  | Cte _ → ()
  | Sub e1 e2 → cps_correct_expr e1; cps_correct_expr e2
  end
```

Here we are defining a *program*, using `let` instead of `function`. As for any program, we have to prove it correct, i.e., a verification condition is generated. When such a program is given the extra attribute `lemma`, its contract is then turned into a logical statement, which is added to the logical context. The statement is exactly that of lemma `cps_correct_expr` above. For that to be sound, a termination proof is mandatory, hence the `variant` clause. In this case, it means that recursive calls are made on arguments that are structurally smaller than e. (More details on variants and termination proofs are given in the next section.) The VC for this lemma function is easily proved by SMT solvers.

3.2 Encoding Higher-Order Functions

Why3 is essentially a first-order system. It treats the *mapping type* `'a → 'b` as an ordinary abstract type whose inhabitants are lambda-abstractions. Why3 desugars functional applications like `(k (eval_0 e))` using a binary "mapping application" operation: `k @ eval_0 e`. Partial applications of function and predicate symbols are replaced with suitable lambda-abstractions.

Why3 does not implement higher-order unification and provides no means of construction of new mappings beyond what can be derived from the user axioms using first-order logic. Thus, the only construction that requires special treatment is lambda-abstraction.

Why3 translates abstractions to first-order logic using lambda-lifting. This amounts to representing every lambda-term in the problem as a fresh top-level function which takes the free variables of that lambda-term as arguments. For example, here is how the definition of `eval_1` in Fig. 2 is translated:

```
function lam1 (k: int → 'a) (v1: int) : int → 'a

function lam2 (k: int → 'a) (e2: expr) : int → 'a

function eval_1 (e: expr) (k: int → 'a) : 'a =
  match e with
  | Cte n → k @ n
  | Sub e1 e2 → eval_1 e1 (lam2 k e2)
  end

axiom lam1_def : forall k: int → 'a, v1 v2: int.
  lam1 k v1 @ v2 = k @ (v1 - v2)

axiom lam2_def : forall k: int → 'a, e2: expr, v1:int.
  lam2 k e2 @ v1 = eval_1 e2 (lam1 k v1)
```

Here, `lam1` represents the inner lambda-term `\ v2. k (v1 - v2)`. The function `lam2` represents the outer term `\ v1. eval_1 e2 (\ v2. k (v1 - v2))`. Since `eval_1` and the introduced functions `lam1` and `lam2` are mutually recursive, Why3 puts the translated definition of `eval_1` between the declarations of uninterpreted symbols `lam1` and `lam2` and the axioms that define their semantics.

Why3 assumes extensional equality of mappings, even though it does not include the corresponding axiom in the translation (this may change in future versions). Because of this, lambda-terms that occur under equality are not lifted, but inlined. For example, an equality `id = \ n: int. n` is rewritten into `forall n: int. id @ n = n`. Then the axiom of extensionality can be directly proved in Why3.

The current implementation does not detect when two lambda-terms are instances of one common pattern. In particular, this makes it not robust with respect to inlining: if the definition of a function symbol contains a lambda-term and is inlined, then every occurrence of that term will be lifted separately, which

```
let rec continue_2 (c: cont) (v: int) : int =
  match c with
  | A1 e2 k → eval_2 e2 (A2 v k)
  | A2 v1 k → continue_2 k (v1 - v)
  | I → v
  end

with eval_2 (e: expr) (c: cont) : int =
  match e with
  | Cte n → continue_2 c n
  | Sub e1 e2 → eval_2 e1 (A1 e2 c)
  end

let interpret_2 (p: prog) : int = eval_2 p I
```

Fig. 3. De-functionalized interpreter

may affect the provability of the task. We intend to alleviate this shortcoming in future versions of Why3.

4 Defunctionalization

The next step of our case study is to "defunctionalize" the code of the previous section. The general idea of defunctionalization is to replace the functions used as continuations by some first-order algebraic data type, having as many constructors as the various lambdas in the CPS code. For our example, this data type is

```
type cont = A1 expr cont | A2 int cont | I
```

It has three constructors, corresponding to the three continuations of the code on Fig. 2, two on line 5 and one on line 8. The second continuation on line 5 is `(\ v2. k (v1 - v2))`. It has two free variables: `v1` of type `int` and `k` which is itself a continuation. The constructor `A2` associated to this continuation is thus given parameters of type `int` and `cont` (hence the algebraic data type is recursive). Similarly, the first continuation has both `e2` and `k` as free variables, so the corresponding constructor `A1` is given parameters of type `expr` and `cont`. Finally, the third continuation has no free variables. It is associated to the constant constructor `I`.

To derive a new interpreter `interpret_2`, we introduce an extra function `continue_2`, defined mutually recursively with function `eval_2`. It takes as arguments a continuation `c` and a value `v`, and evaluates `c` applied to `v`. Notice that we now define *programs*, introduced with `let`, instead of logic functions introduced with `function`. The reason is technological: the mutual recursion above cannot be statically checked terminating by Why3, the termination argument

being not structural. The termination has to be proved by theorem proving, as shown in Section 4.3 below.

It is worth pointing out that the code of `eval_2` and `continue_2` only make tail calls. Thus it can be seen as a small-step semantics for our language or, equivalently, as an abstract machine for interpreting programs in this language.

4.1 Soundness

Since we wrote a program and not a logic definition, the soundness of our defunctionalized interpreter is not expressed by a pure logical lemma but with a contract for the `interpret_2` program, where the post-condition tells that the result coincides with the big-step semantics.

```
let interpret_2 (p: prog) : int
  ensures { result = eval_0 p }
```

To achieve the proof that the interpreter satisfies this contract, we need to find appropriate contracts for the auxiliary programs `eval_2` and `continue_2`. For that purpose, a first idea would be to define a logic function `eval_cont (c:cont) (v:int)` returning the evaluation of `c` on `v`. Such a function would be a non-structurally recursive function, and would be rejected by Why3: logic functions and predicates must be guaranteed to terminate (see Section 5 for a discussion about this limitation of Why3). However, instead of a logic function, we can define a ternary predicate `eval_cont (c:cont) (v:int) (r:int)` expressing that `r` is the result of the evaluation of `c` on `v`. Such a predicate can be defined inductively as follows.

```
inductive eval_cont cont int int =
| a1 : forall e: expr, k: cont, v r: int.
         eval_cont (A2 v k) (eval_0 e) r → eval_cont (A1 e k) v r
| a2 : forall n: int, k: cont, v r: int.
         eval_cont k (n - v) r → eval_cont (A2 n k) v r
| i  : forall v: int. eval_cont I v v
```

Such a definition corresponds to a set of inference rules for an evaluation judgment $c\ v \Rightarrow r$ that means "the application of continuation c to the value v evaluates to r". To ensure the consistency of the definition, Why3 checks for the standard positivity conditions of occurrences of `eval_cont` in each clause. The contracts for our auxiliary programs can then be stated as follows.

```
let rec continue_2 (c: cont) (v: int) : int
  ensures { eval_cont c v result }

with eval_2 (e: expr) (c: cont) : int
  ensures { eval_cont c (eval_0 e) result }
```

Annotated as such, those programs are easily proved correct by automated provers. For example, Alt-Ergo, CVC3, and Z3 all discharge the VCs in no time.

4.2 Encoding of Inductive Predicates

Translation of inductive predicates for first-order provers is similar to that of algebraic types: Why3 declares an uninterpreted predicate symbol and then adds one axiom per clause of the inductive definition plus the axiom of inversion. Here is the translation of the `eval_cont` predicate.

```
predicate eval_cont (e: cont) (v r: int)

axiom a1 : forall e: expr, k: cont, v r: int.
  eval_cont (A2 v k) (eval_0 e) r → eval_cont (A1 e k) v r

axiom a2 : forall n: int, k: cont, v r: int.
  eval_cont k (n - v) r → eval_cont (A2 n k) v r

axiom a3 : forall v: int. eval_cont I v v

axiom eval_cont_inversion : forall k0: cont, v0 r0: int.
  eval_cont k0 v0 r0 →
    ((exists e: expr, k: cont.
        eval_cont (A2 v0 k) (eval_0 e) r0 ∧ k0 = A1 e k)  ∨
     (exists n: int, k: cont, v0 r0: int.
        eval_cont k (n - v0) r0 ∧ k0 = A2 n k)  ∨
     (k0 = I ∧ v0 = r0))
```

Just like in the case of algebraic types, this translation is incomplete, because we are unable to express in first-order logic that `eval_cont` is the least relation satisfying the axioms. If we declared `eval_cont` as a coinductive predicate (which is also supported by Why3), the translation for first-order provers would be exactly the same. This does not present a problem, since these provers do not perform reasoning by induction or by coinduction. However, it is desirable to be able to make proofs by induction over an inductive predicate inside Why3, using dedicated transformations and/or ghost lemmas, similarly to what we do for algebraic types (Sec. 3.1). This is one of the directions of our future work.

4.3 Termination

So far, we have proved the partial correctness of our interpreter. This is not fully satisfactory since an implementation that would loop forever would also satisfy the same contract. We thus aim at proving termination as well.

Because the two auxiliary programs are mutually recursive in a quite intricate way, it is not completely trivial to find a termination measure that decreases on each recursive call. One adequate measure is given by the following ad-hoc size functions:

```
function size_e (e: expr) : int =
  match e with
  | Cte _ → 1
  | Sub e1 e2 → 3 + size_e e1 + size_e e2
```

```
    end

  function size_c (c: cont) : int =
    match c with
    | I → 0
    | A1 e2 k → 2 + size_e e2 + size_c k
    | A2 _ k → 1 + size_c k
    end
```

The contracts of our auxiliary functions are then augmented with the following variants:

```
  let rec continue_2 (c: cont) (v: int) : int
    variant { size_c c }
    ...

  with eval_2 (e: expr) (c: cont) : int
    variant { size_c c + size_e e }
    ...
```

Generally speaking, a set of mutually recursive programs can be annotated as

$$
\begin{array}{l}
\texttt{let rec } f_1\ \boldsymbol{x_1} \\
\quad \texttt{variant } \{\ v_{1,1}(\boldsymbol{x_1})\ [\texttt{with } R_1],\ \ldots,\ v_{1,m}(\boldsymbol{x_1})\ [\texttt{with } R_m]\ \} \\
\vdots \\
\texttt{with } f_n\ \boldsymbol{x_n} \\
\quad \texttt{variant } \{\ v_{n,1}(\boldsymbol{x_n})\ [\texttt{with } R_1],\ \ldots,\ v_{n,m}(\boldsymbol{x_n})\ [\texttt{with } R_m]\ \}
\end{array}
$$

where each $v_{i,j}$ returns a value on some type τ_j and R_j is a binary relation on τ_j. The verification conditions are then:

1. Each relation R_j must be well-founded.
2. For each call $f_i(\boldsymbol{e})$ in the body of the program f_j, the vector $(v_{i,1}(\boldsymbol{e}), \ldots, v_{i,m}(\boldsymbol{e}))$ must be strictly less than the vector $(v_{j_1}(\boldsymbol{x_j}), \ldots, v_{j,m}(\boldsymbol{x_j}))$ with respect to the lexicographic combination of order relations $R_1, \ldots, R_m$.

These verification conditions ensure termination because if there were an infinite sequence of program calls, there would be an infinite decreasing sequence for $R_1, \ldots, R_m$.

Why3 assumes default relations for the variant clauses so that, most of the time, the user does not need to provide the relation. If the type τ_j is `int`, the default relation is

$$R_{int}\ y\ x := x > y \wedge x \geq 0.$$

If τ_j is an algebraic data type, then R_j is the immediate sub-term relation.

For our interpreter, the default relation R_{int} is used. The VCs related to termination cannot be proved automatically, since SMT solvers lack the information that sizes are non-negative. So we first state this as two lemmas:

```
  lemma size_e_pos: forall e: expr. size_e e ≥ 1
  lemma size_c_pos: forall c: cont. size_c c ≥ 0
```

Both require induction to be proved. As we did earlier in Section 3.1, we can either use Why3's transformation for structural induction or turn these lemmas into recursive lemma functions. The former is simpler. Once these two lemmas are proved, termination of `eval_2` and `continue_2` is proved automatically by SMT solvers.

5 Conclusions

Using the automatic program verifier Why3, we solved a student exercise aiming at illustrating, on a simple language, the relations between various semantics that were described in a research paper [14]. Unlike the students, we did not only *code* the interpreters and tested them, but also *proved them correct*. We formalized the big-step semantics of that language and proved correct its compilation, first into a CPS-style semantics and then into a small-step one, close to an abstract interpreter. The complete example available at http://toccata.lri.fr/gallery/defunctionalization.en.html also contains another variant of that language, forbidding negative numbers and possibly raising an "Underflow" exception. It also formalizes another semantics based on rewriting, making an analogy between reduction contexts and continuations, and an explicit abstract machine for executing the language. Even if the specifications and the proofs require advanced features such as algebraic data types, inductive predicates, and higher-order functions, we were able to prove our interpreters formally using only automated theorem provers.

Related Tools and Experiments. Why3 is not the only software verification tool where this kind of formalization can be done. For instance, the second Verified Software Competition [20] featured a challenge related to the semantics of S and K combinators and this challenge was successfully tackled by systems such as ACL2, Dafny, VCC [21], or VeriFast. Some features, such as algebraic data types and pattern matching, can be encoded without too much difficulty if a given system does not have a native support for them. Other features, such as type polymorphism, are much more difficult to simulate. Fortunately, polymorphism (parametric or ad-hoc) gained much traction in the mainstream programming languages (Java generics, C++ templates) and the verification tools follow the trail. For instance, both Dafny and VeriFast support generic types.

Perspectives. The question behind this experiment is whether an automatic program verifier can be a reasonable replacement for an interactive proof assistant for formalizing a programming language, its semantics, and its compilation. More generally, any program performing symbolic computation is a potential use case for the proposed techniques. Recently, we developed a generic approach for data types with binders in Why3. On top of it, we built a verified interpreter for lambda-calculus and a verified tableaux-based first-order theorem prover [22]. A couple of years ago, we formalized in Why3 a simple imperative while-language [23], with an operational semantics, some Hoare-style

rules proved correct with respect to the operational semantics, and (partially) a weakest precondition calculus. This former case study was involving a few Coq proofs and was missing support for higher-order logic, hence it would deserve to be revisited with the current Why3 environment.

Along these lines, we intend to improve Why3 in the following ways. In short term, we want to add support for non-structural recursion in pure functions, as it is done for programming functions. We also intend to allow lambda-expressions to be used in programs, provided they are terminating and side-effect free. A key challenge to apprehend large programs and large proofs is the ability to control the logical context. Indeed, automated provers are extremely sensitive to the number, size, and shape of logical premises. Why3 already provides a module system that allows the user to split specification and implementation into small interlinked components [13]. We are currently working on the principles of refinement for this module system (both for specification and program code). The idea is to verify program components in a most abstract and minimal context, and then to reuse them in a complex and refined development. Another way to minimize the context before invoking automated provers is to provide some limited interaction where the user filters out irrelevant concepts, e.g. everything which is related to the properties of sorted arrays if the conclusion under consideration does not require it.

Acknowledgments. We thank Olivier Danvy for proposing these exercises.

References

1. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43**(4) (2009) 363–446
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer (2004)
3. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. Communications of the ACM **53**(6) (June 2010) 107–115
4. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of Lecture Notes in Computer Science. Springer (2002)
5. Kaufmann, M., Moore, J.S., Manolios, P.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Norwell, MA, USA (2000)
6. Liu, H., Moore, J.: Java program verification via a JVM deep embedding in ACL2. In Slind, K., Bunker, A., Gopalakrishnan, G., eds.: Theorem Proving in Higher Order Logics. Volume 3223 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2004) 184–200
7. Bodin, M., Charguéraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A., Smith, G.: A trusted mechanised JavaScript specification. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, USA, ACM Press (January 2014)

8. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR-16. Volume 6355 of Lecture Notes in Computer Science., Springer (2010) 348–370
9. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R., eds.: NASA Formal Methods. Volume 6617 of Lecture Notes in Computer Science., Springer (2011) 41–55
10. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. In: Proceedings of the 10th International Conference on Software Engineering and Formal Methods. Number 7504 in Lecture Notes in Computer Science, Springer (2012) 233–247
11. Guitton, J., Kanig, J., Moy, Y.: Why Hi-Lite Ada? In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland (August 2011) 27–39
12. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In Felleisen, M., Gardner, P., eds.: Proceedings of the 22nd European Symposium on Programming. Volume 7792 of Lecture Notes in Computer Science., Springer (March 2013) 125–128
13. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland (August 2011) 53–64
14. Danvy, O., Nielsen, L.R.: Defunctionalization at work. In: Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming. PPDP '01, ACM Press (2001) 162–174
15. Bobot, F., Conchon, S., Contejean, E., Iguernelala, M., Lescuyer, S., Mebsout, A.: The Alt-Ergo automated theorem prover (2008) http://alt-ergo.lri.fr/.
16. Bobot, F., Paskevich, A.: Expressing Polymorphic Types in a Many-Sorted Language. In Tinelli, C., Sofronie-Stokkermans, V., eds.: Frontiers of Combining Systems, 8th International Symposium, Proceedings. Volume 6989 of Lecture Notes in Computer Science., Saarbrücken, Germany (October 2011) 87–102
17. Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Encoding monomorphic and polymorphic types. In Piterman, N., Smolka, S.A., eds.: TACAS. Volume 7795 of Lecture Notes in Computer Science., Springer (2013) 493–507
18. Augustsson, L.: Compiling pattern matching. In: Functional Programming Languages and Computer Architecture, Nancy, LNCS 201, Springer (September 1985)
19. Leino, K.R.M.: Automating induction with an SMT solver. In: Proc. 13th Int. Conf. Verification, Model Checking, and Abstract Interpretation (VMCAI 2012), Philadelphia, PA (2012)
20. Filliâtre, J.C., Paskevich, A., Stump, A.: The 2nd verified software competition: Experience report. In Klebanov, V., Grebing, S., eds.: COMPARE2012: 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems, Manchester, UK, EasyChair (June 2012)
21. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Theorem Proving in Higher Order Logics (TPHOLs). Volume 5674 of Lecture Notes in Computer Science., Springer (2009)
22. Clochard, M., Marché, C., Paskevich, A.: Verified programs with binders. In: Programming Languages meets Program Verification (PLPV), ACM Press (2014)
23. Marché, C., Tafat, A.: Weakest precondition calculus, revisited using Why3. Research Report RR-8185, INRIA (December 2012)