Formally Verifying Imperative Programs

José Pedro Correia, José Pedro Magalhães, and Jorge Sousa Pinto

Universidade do Minho, Escola de Engenharia, Departamento de Informática

Abstract. We present a view on the problem of formal verification of imperative programs. Formal verification is usually associated to functional programming languages, but a real-world application will probably be written in imperative main-stream languages. To have the power of formal verification in an imperative language, we present, for demonstrative purposes, an instance of a technique to extend a language with support for logical annotations. We present a short description of `LISS` (the imperative language chosen to be extended) and `Why` (the verification condition generator), and after a detailed analysis of the mechanisms used to generate proof-obligations, we conclude, analyzing the global quality of the result.

1 Introduction

1.1 Context

The term "program verification" is normally used with two meanings: on one hand, it refers to proving the correctness of intended algorithms with respect to a certain formal specification or property; on the other hand, it refers to obtaining guarantees of the program's behavior with respect to aspects such as termination, memory access, etc. On both cases, it's an instance of a recurring slogan in computer programming: it's not enough for the programmer to know how to create working programs; it is also necessary to know how to present the formal guarantees of quality and/or validity of the program. The work reported here intends precisely to explore that competence.

1.2 Case study

Since its introduction, in 1969, Hoare logic [5] is regarded as a very important tool for program verification. Based on this methodology, one commonly uses:

– A verification condition generator, and
– A theorem prover or proof-assistant

Here we analyze the use of the `Why` language [3] as a generic verification condition generator (both in the language to analyze and the prover/proof-assistant to use), stemming from the Caduceus [4] example as a verification condition generator for the `C` language. Thus, the objective is to explore all the steps in this process, allowing this work to be integrated into a more general initiative in the Proof-Carrying Code [7] field.

Our work basically consists of two distinct parts: firstly we explore the use of the `Caduceus` tool (which, given a properly annotated `C` program, allows to generate a set of verification conditions which can then be proved automatically or in an assisted fashion with a proof-assistant); secondly, we construct a tool similar to `Caduceus`, allowing the annotation of programs in a small but sufficiently expressive imperative language.

1.3 Structure of this document

The rest of this article is structured as follows: the next section presents a short description/introduction to the Caduceus tool, used as model of what we propose to do. In section 3 we present the `LISS` language (which we used, after extension with support for logical annotations, as input language), then a brief introduction to `Why` (the output language) in section 4, followed by a detailed analysis on the translation process (section 5). Finally, we present some examples of the translation (section 6) and conclude in section 7, also presenting possible directions for future work.

2 Analyzing Caduceus

`Caduceus` [4] is a verification tool for `C` programs at source level. Reading comments structured in a special syntax, this tool can generate proof-obligations which, after being proved in any of the supported provers, will ensure the validity and/or safety of the original `C` program. Apart from function specification properties, `Caduceus` also supports dynamic memory addressing issues, such as null pointer dereferencing and out-of-bounds array access. It uses the `Why` tool to generate the proof obligations, transforming the initial `C` program into input for `Why`.

```
/*@ requires \valid(p)
  @ ensures *p >= 0
  @*/
void abs(int *p) {
  if (*p < 0) *p = -*p;
}
```

Listing 1.1. Example of `Caduceus` annotations on a `C` program

Listing 1.1 shows an example of `Caduceus` annotations on a `C` program. These are:

- A precondition requiring the pointer `p` to be validly allocated when the function `abs` is called;
- A postcondition guaranteing the value pointed by `p` after the execution is greater than or equal to zero.

The translation process from Caduceus annotations to Why code is described in some detail in [4], together with the necessary axiomatic basis for array and pointer support (and a soundness analysis). Our tool will follow a similar structure, since it's built for a similar purpose.

3 From: `LISS`

`LISS`[1] (Language of Integers, Sequences and Sets) [2] is a programming language traditionally used by the DI-UM[2] and by the DI-UBI[3] as an instrument in the study of language processing and compilers. It clearly served our purposes, being a toy imperative programming language, so we used it as a basis to extend with logical annotations.

The syntax already defined for this language was not clear enough, and since it was our intention to have full control over it, we started by redefining `LISS` according to the techniques of [8], later extending it to serve our purpose[4].

Briefly, the language includes the following components[5]:

- Datatypes
  - Booleans
  - Integer numbers
  - Mutable integer sequences
  - Integer sets (defined by comprehension)
  - Static integer arrays
- Local functions with pre and post-conditions
- Control flow instructions
  - Attribution for all types (except arrays) and array positions
  - If-then-else
  - `while` loop with variant and invariant annotations
  - `for` loop which iterates a variable over a range of integer values
  - `foreach` loop which iterates a variable over the values in a sequence
- Basic input/output instructions
- A unitary instruction `skip`

4 To: `Why`

The `Why` tool[6] [3], developed at the French *Laboratoire de Recherche en Informatique* (LRI), takes an annotated program in a small imperative language of

[1] Compiler webpage at `http://www.di.uminho.pt/~gepl/LISS/`
[2] Webpage at `http://www.di.uminho.pt/`
[3] Webpage at `http://www.di.ubi.pt/`
[4] This implies that we cannot compare the behavior of a program compiled with the (old) compiler with the behavior of the same program while interpreted, since there are modifications to the language structure.
[5] A complete grammar of the language can be found on the supporting webpage at `http://twiki.di.uminho.pt/twiki/bin/view/Research/ALiss`.
[6] Webpage at `http://why.lri.fr/`

its own as input and generates verification conditions in diverse formats. These formats are compatible with different provers, including (but not restricted to) Coq[7], PVS, Isabelle, Hol (proof-assistants) and Simplify, CVC Lite and haRVey (automatic provers). Its main advantages are:

- It's a verification condition generator for a language specifically designed for the interpretation of already existing programming languages. It includes normal constructions for imperative programming languages and logical annotations, but also exceptions, recursive functions and polymorphism.
- Allows the declaration of new logical models (types, functions, predicates and axioms) which can then be used in the programs and annotations. In this way, adaptation to new types and constructions (for instance, an arithmetical type of limited precision) can be easily achieved, axiomatizing or leaving them to be proved on the prover side.
- Supports a wide range of existing provers, allowing for combination of their features in the proof of a single program.

Let us take a short example written in the Why language:

```
1 logic min: int, int -> int
2 axiom min_ax: forall x,y:int. min(x,y) <= x
3 parameter r: int ref
4 let f (n:int) = r := min !r n { r <= r@ }
```

Listing 1.2. Why example

Line-by-line, we have:

1. Declaring a function, together with its type and arity. Independently of how this function will be interpreted on the prover-side, its definition is enough for usage within Why.
2. Axiom introduction over the logical property min used in line 1.
3. A parameter is a value whose existence is assumed, i.e. belongs to the environment. Here the parameter is called r and its type is reference to integer (note the similarity to Ocaml's syntax).
4. Definition of a function f, without precondition and with a postcondition stating that the final value of r is less or equal than its initial value. The current value of a reference x is, inside annotations, referred to as x, while as part of an instruction is referred as !x. Inside postconditions, the notation x@ refers to the value of x in its initial state (i.e. in the precondition).

After processing by the tool, the code in listing 1.2 is transformed into suitable input for the prover/proof-assistant of choice.

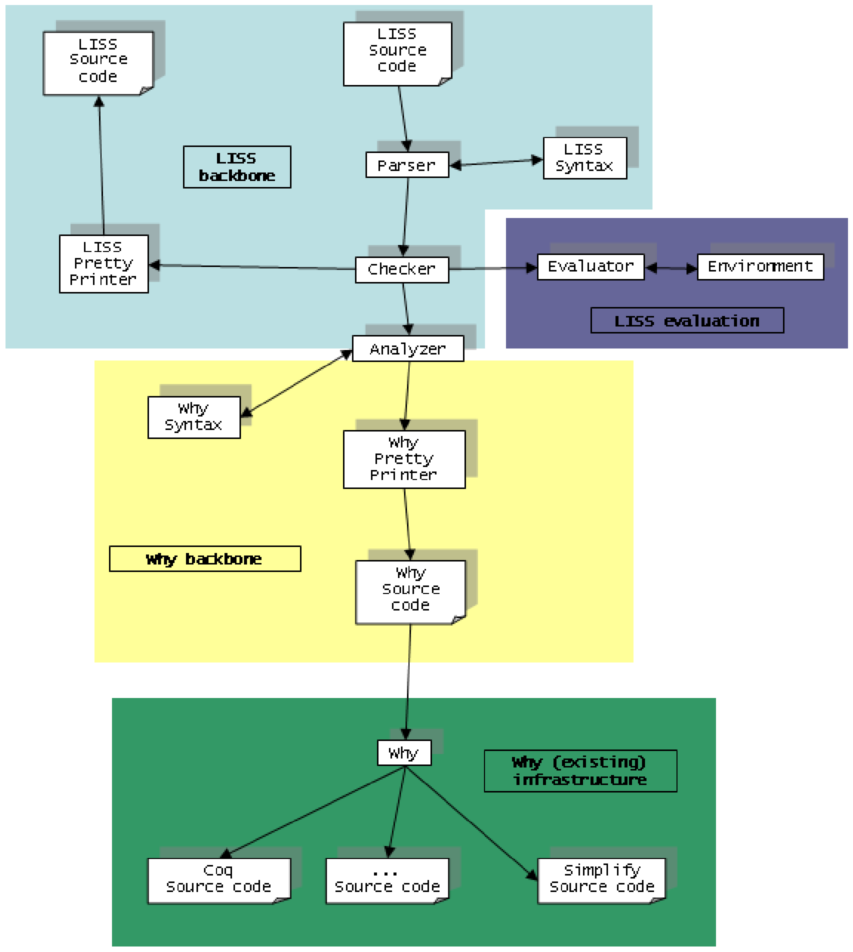

Fig. 1. Overall system architecture

5 Translation

In this section we present the general structure of our solution (figure 1), discussing all the involved components and their integration into a coherent tool (completely implemented in the `Haskell` language[8]).

5.1 Parsing and pretty-printing

Initially, all the infrastructure used for parsing and pretty-printing was written in the Utrecht University Parser Combinators style [9]. However, the parser

[7] Webpage at `http://coq.inria.fr/`
[8] Webpage at `http://www.haskell.org`

has since been rebuilt, this time using Parsec [6] (the pretty-printing, however, was maintained). The respective Abstract Syntax Trees (ASTs) have also been represented using Haskell data types.

5.2 `Checker`

The `Checker` is an intermediate processing step, intending to correct the output of the parser (which knows nothing about types). For instance, in an assignment of a variable to another variable, the parser cannot know what the types of these variables are, so it will just infer a generic assignment. The `Checker` looks at the types of the variables, checks that they match and turns the assignment into a typed assignment, providing information which might be necessary later in the translation (or evaluation) process.

In case there are type errors in the program (such as using an integer variable as the condition of an `if` expression), they are detected and reported at this step.

5.3 `Evaluator`

The existence of an evaluation mechanism for `LISS` expressions is important, since not only it allows for easier testing of programs, but also could be used for program simplification (like constant folding, for instance). Thus, we developed the `Evaluator`, coupled with its `Environment`, which can roughly be seen as a translation from `LISS` to `Haskell`, interpreting the former using the functionalities of the latter.[9]

5.4 Axiomatic basis

As with `Caduceus` and `C` pointers, there was a need to introduce new axioms regarding the `LISS` datatypes and operations not directly supported by `Why`.

Input/Output and `skip` Instructions `read` and `write` require a representation, as does the `skip` instruction. The `Why` source of these is represented in figure 1.3. Basically the interest of these is to correctly type the operations in `Why`.

```
parameter read :
v:int ref ->
{ } unit writes v { }

parameter write :
v:int ->
{ } unit { }

logic nop : unit
```

Listing 1.3. Input/Output operations

[9] Its denotational semantics are not included here, but the interested reader is referred to the (earlier referred) supporting webpage.

Arrays The built-in representation of arrays of `Why` was used, which contains all the common operations over arrays, and also logical operations like testing if an array is sorted and if an array is a permutation of another one. These can also be, in the future, included in `LISS`.

Sequences In order to support sequences, a new type is introduced and the necessary operations, like constructors `seqnil` and `cons`, destructors `head` and `tail`, and a `length` operation. Needed axioms for these operations are also introduced. The `Why` source for this type is not included here.

Sets As with sequences, an axiomatic basis is provided for this new type with operations like `inset` that tests if an integer is a member of a set, `union` and `intersect`, as well as a representation for an empty set. A fragment of the `Why` source is represented in figure 1.4. Unfortunately, this basis for sets is yet to be properly tested.

```
type set

logic inset : int, set -> bool
logic intersect : set, set -> set
logic union : set, set -> set
logic setnil : set

axiom set_inset_nil :
forall i:int.
inset(i,setnil) = false

axiom set_intersect_nil_l :
forall s:set.
intersect(s, setnil) = setnil

axiom set_inset_intersect1 :
forall s1:set. forall s2:set. forall i:int.
inset(i,intersect(s1,s2)) = inset(i,s1)

axiom set_comm_union :
forall a:set. forall b:set.
union(a,b) = union(b,a)

...
```

Listing 1.4. `Why` axiomatic support for sets

5.5 Analyzer

The objective of the translation is to know how to represent each element of the `LISS` syntax into a valid `Why` operation. Specification of this correspondence is represented as a sort of a "translation grammar", which for reasons of space limitation cannot be included here, but can be consulted online. There is also the need of an environment during the translation process in order to generate "fresh" names for variables and labels.

There is no interest in analyzing here the full translation grammar[10], so we'll just present the example in figure 2. This represents the translation of a `LISS` integer expression. As can be seen, most of the translation is quite straightforward. Things to be taken into account here are the use of `!` operator in `Why` to access the value of a variable and the use of our predefined operator `why_head` (refer to section 5.4), which introduces the precondition that the sequence isn't empty.

$$
\begin{aligned}
[\![i]\!]_{intexp}\sigma &= i \\
[\![var]\!]_{intexp}\sigma &= \texttt{!}var \\
[\![a\texttt{[}\ ie\ \texttt{]}]\!]_{intexp}\sigma &= a\texttt{[}\ [\![ie]\!]_{intexp}\sigma\ \texttt{]} \\
[\![-\ ie]\!]_{intexp}\sigma &= \texttt{-}\ [\![ie]\!]_{intexp}\sigma \\
[\![ie1 + ie2]\!]_{intexp}\sigma &= [\![ie1]\!]_{intexp}\sigma\ \texttt{+}\ [\![ie2]\!]_{intexp}\sigma \\
[\![ie1 - ie2]\!]_{intexp}\sigma &= [\![ie1]\!]_{intexp}\sigma\ \texttt{-}\ [\![ie2]\!]_{intexp}\sigma \\
[\![ie1 \times ie2]\!]_{intexp}\sigma &= [\![ie1]\!]_{intexp}\sigma\ \texttt{*}\ [\![ie2]\!]_{intexp}\sigma \\
[\![ie1\ /\ ie2]\!]_{intexp}\sigma &= [\![ie1]\!]_{intexp}\sigma\ \texttt{/}\ [\![ie2]\!]_{intexp}\sigma \\
[\![ie1\ \mathbf{rem}\ ie2]\!]_{intexp}\sigma &= [\![ie1]\!]_{intexp}\sigma\ \texttt{\%}\ [\![ie2]\!]_{intexp}\sigma \\
[\![\mathbf{head}\ se]\!]_{intexp}\sigma &= \texttt{why\_head}\ \texttt{(}\ [\![se]\!]_{seqexp}\sigma\ \texttt{)} \\
[\![\mathbf{call}\ name\ vars]\!]_{intexp}\sigma &= name\ \texttt{(}\ vars\ \texttt{)}
\end{aligned}
$$

Fig. 2. Integer expressions translation to `Why`

Other important choices in all of the translation process are the following:

- Functions are translated to `let ... in` statements in order to be faithful to the semantics of `LISS`
- The loops `for` and `foreach` are "syntatic sugar". They are translated to `while` loops
- For now, `for` and `foreach` have no invariant, only a variant to guarantee termination

5.6 Syntactic incompatibility

As referred before, the whole project was developed using the `Haskell` language. In this translation phase, we were confronted with a problem due to `Haskell`'s

[10] The complete translation grammar can be consulted online at the supporting webpage.

strong type system: as both ASTs of `Why` and `LISS` were defined as mutually recursive datatypes, a direct translation function posed as a complicated problem (see example on figure 3 — the code on the left column is the `Why` AST and the one on the right belongs to the `LISS` AST).

```
data Term =
        TConst Const
      | Sum Term Term
      | Sub Term Term
      | TNegation Term
      | ...

data Const =
        IntConst Int
      | BoolConst Bool
      | ...

data Prog =
        PConst Constant
      | PIdent Identifier
      | Assign Identifier Prog
      | ...
```

```
data IntExp =
        IntConst Int
      | Sum IntExp IntExp
      | Sub IntExp IntExp

data BoolExp =
        BoolConst Bool
      | Negation BoolExp

data Statement =
        IntAssign IntVar IntExp
      | ...
```

Fig. 3. As you can see it's impossible to safely define a function from `IntExp` always to the same type because depending on the context it could be a `Term` or a `Prog`

The solution was to "flatten" the `Why` AST representation into a single datatype. This solves the problem, but allows invalid `Why` code to be generated. A further step of checking would be advisable, but as the code generation is controlled by us, this was left undone.

6 Examples

A simple example to exemplify the translation process is presented in listing 1.5.

```
program sum {
    declarations
        a, b, c -> Integer;

        pre x >= 0 && y >= 0;
        post result == old(x) + old(y);
        subProgram sum(x -> Integer, y -> Integer) -> Integer {
            declarations
                ;
            statements
                invariant x == old(x)+(old(y)-y) && y >= 0;
                variant y;
                while (y > 0) {
                    x := x+1;
                    y := y-1;
                };
            return x;
        };

    statements
        write "a:\n"; read a;
        write "b:\n"; read b;
        if (a >= 0 && b >= 0) then { c := sum(a,b); write "a + b: ";
            write c; };
}
```

Listing 1.5. Inductive sum in `LISS`

It shows a possible implementation of an inductive definition of the sum of two natural numbers, including pre- and postconditions and a loop variant and invariant.

```
let example =
    let a = ref 0 in
    let b = ref 0 in
    let c = ref 0 in
    let sum(x : int ref) (y : int ref) =
        { ((x >= 0) and (y >= 0)) }
        begin
        a1:
        while ((!y>0)) do
               { invariant ((x = (x@a1 + (y@a1 - y))) and (y >= 0))
                    variant y }
                  x := (!x + 1);
                  y := (!y - 1)
        done;
        !x
        end
        { (result = (x@ + y@)) } in
 begin
   nop; read(a);
   nop; read(b);
   if (((!a>=0)&&(!b>=0)))
      then
         begin
            c := sum(a)(b); nop;
            write(!c)
         end
      else
         begin
            nop
         end;
   nop
 end
```

Listing 1.6. Translation in `Why`

Its translation in `Why` (listing 1.6) is not so easy to read, partly because of the introduction of superfluous `nop` instructions, but the semantics are preserved.

After running Why on this code, 3 proof-obligations are generated, corresponding to the loop initialization, preservation and termination. The precondition stated in the function is given as hypothesis for the proofs, and the postcondition is implied by the loop termination, so no proof is necessary. In `Coq` [1], the strategy `intuition` suffices to prove all the 3 lemmas.

7 Conclusion and future work

7.1 Limitations

Given the complexity of the translation and the exploratory nature of this work, some details were left without proper testing in the `Analyzer`, and this component is recognized as the "weakest link" in the solution, naturally being a good candidate for further refinement in the future. Those details include, for instance, recursive functions and set handling, the former (being allowed by `Why`) being translated directly and lacking effective testing, and the latter having an axiomatic basis constructed by us, but lacking proper integration with the `Analyzer`.

7.2 Future work

We present the directions of future work into two separate possibilities, one addressing this project as an isolated work, and another integrating it into a possible, forthcoming new project for the DI/CCTC[11] in the area of PCC.

On this project Naturally, future work on this tool should focus on its current limitations. For that end, together with further testing and refinement of the `Analyzer` component, we can suggest:

- Allowing for more functionalities in `LISS`, such as predefined functions for arrays and sequences;
- Considering semantical changes to the arrays, bringing them closer to the usual semantics on imperative languages (pass-by-reference) — this could imply the need for a complex memory model, as the one used by `Caduceus`;
- Lifting functionalities already available in `Why` to `LISS` (such as operations on sorted arrays and permutations).
- Constructing a complete PCC compilation environment, from the annotated `LISS` code to a final binary executable and additional safety proof. This, as was suggested to us, would add considerable interest to the solution and make it easier to use as teaching tool and laboratory for new verification techniques.

On a more generic initiative in the PCC field For this purpose, we think it's wise to report the main difficulties encountered during the development of the tool, intending to ease the future development of similar tools through the prediction of similar problems. Thus, we concluded that:

- The `Why` language is suited for this purpose. It is sufficiently documented and produces the expected output to several provers. Additionally, other

[11] The University of Minho research unit in Informatics — `http://cctc.di.uminho.pt/`

examples such as `Caduceus`[12] and `Krakatoa`[13] are already available and can be studied as models for new translations.

- The choice of the language for the development of the project is an important issue. We have used `Haskell` due to personal preference, but this implies having to represent the `Why` language's syntax and develop (at least) a pretty-printer. Such development is not necessary if one uses `OCaml` (which is the language used in `Why`).
- The translation process itself can be troublesome if the semantics of concepts (such as variable scope, pass-by-value vs. pass-by-reference, mutable vs. immutable references, etc.) are different in the languages. It is then advisable to have these concepts clearly defined in the source language first, and then analyze its existance/semantics in `Why` and finally construct an adequate translation, solving the semantical differences in a coherent fashion.

References

1. B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, et al. The Coq Proof Assistant Reference Manual. *INRIA. Version*, 6(11), 2000.
2. D. Cruz and P.R. Henriques. LISS — A Linguagem e o Compilador. CCTC Internal Report, Departamento de Informática, Universidade do Minho, Jan. 2007. (to be published).
3. J.-C. Filliâtre. Why: a Multi-Language Multi-Prover Verification Tool. Research Report 1366, LRI, Université Paris Sud, March 2003.
4. J.-C. Filliâtre and C. Marché. Multi-Prover Verification of C Programs. In *Sixth International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Seattle, November 2004. Springer-Verlag.
5. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
6. D. Leijen and E. Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.
7. G.C. Necula. Proof-Carrying Code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, Jan. 1997.
8. J.C. Reynolds. *Theories of Programming Languages.* Cambridge University Press, 1998.
9. S.D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. *Advanced Functional Programming*, 1129:184–207, 1996.

[12] A verification tool for `C` programs — `http://caduceus.lri.fr/`

[13] A verification tool for `Java` programs — `http://krakatoa.lri.fr/`