An AST for Representing Programs with Invariants and Proofs

Guillaume Bertholon, Arthur Charguéraud

An AST for Representing Programs with Invariants and Proofs

Guillaume Bertholon[1,2] and Arthur Charguéraud[2,1]

[1] Université de Strasbourg, CNRS, ICube, France
[2] Inria, Strasbourg, France

Abstract

Deductive verification enables one to check that a program satisfies its specification. There are mainly two approaches: either the user provides invariants in the form of annotations and use a tool to extract proof obligations, like in, e.g., Why3; or the user verifies the program through interactive proofs, like in, e.g., CFML, by providing invariants during the proof steps.

We are interested in expressing in Coq the representation of a program, accompanied with not only its invariants but also its proof terms. Concretely, we present an AST for representing source code and specification in a deep embedding style, and embedded lemmas in shallow embedding style. Such lemmas can be established using the full capabilities of the prover.

We develop a way to build these ASTs from source code using CFML-style interactive tactics. We also develop a way to build these ASTs by extracting proof obligations from source code already annotated with its invariants. Besides, we provide a way to validate our ASTs by reifying them as Coq proof terms.

This work is a first step towards a long term project to devise a trustworthy, user-guided, source-to-source optimization framework. On the one hand, we may need to exploit invariants to justify the correctness of code transformations. On the other hand, to be able to chain transformations, we also need every transformation to update the program annotations.

1 Introduction

Producing programs without bugs is known to be a hard task. In order to guarantee the absence of bugs, programmers can use deductive verification. Deductive verification is a technique that allows programmers to state the formal specification, and statically check that they hold on the program. The link between the code and the specification must be guaranteed by a proof. In practice however, verifying a program requires a lot of effort. Indeed, the amount of effort grows even larger when targeted program grow in size or in in complexity. The formal verification of realistic high performance code, appears prohibitively costly.

In our long term project, we aim to develop a methodology for deriving programs that are both highly optimized and formally verified. Our plan is to start from unoptimized code that is easier to verify, and progressively refine this code into a more performant version. Such source-to-source code refinement process has already been shown in the OptiTrust framework to be expressive enough to match the performance of a numerical simulation optimized by hand [3]. However, OptiTrust does not yet provide formal guarantees about the correctness of the output code. To ensure that a source-to-source transformation preserves correctness, it may be needed to exploit the existing invariants of the input code. Therefore to support chain of transformation, each transformation will need to maintain program invariants.

In this paper, we present an abstract syntax tree (AST) data structure for representing programs, together with their invariants and proofs. We also give practical methods to build these

ASTs, and to validate the proof they carry. However, we leave to future work the implementation of practical transformations, e.g., function inlining, arithmetic rewriting (such as replacing a division by a shift), loop reordering, or even replacing a part of the program by a semantically equivalent but arbitrary code. Nonetheless, our AST provides a way for transformations to access those invariants and proofs and maintain both while changing the code.

Before explaining in more details how our AST is represented, let us recall the ways in which invariants are provided and proofs are constructed in state-of-the-art deductive verification tools. Typically, the specification of a program is expressed with a Hoare triple stating what pre-condition must be true before the execution, and what post-condition is true after the execution. In addition, one needs to describe the mutable state, and express the fact that pointers may or may not overlap, i.e., do not alias. These memory properties can be stated concisely using Separation Logic [11]. Given a specification, the verification of a program includes three kind of steps: (1) semantic steps, which process one construct of the source code; (2) structural steps, which refine invariants at a given program point, possibly exploiting the rules of separation logic; and (3) pure steps, which do not involve the program or the data it manipulates directly but are mathematical arguments needed to conclude. Throughout the paper, we call such pure steps *proof leaves*, since they can only appear at the end of proof branches.

These proof steps can be organized in different ways depending on the approach followed to construct a proof that a program satisfies its specification. There are mainly two kind of approaches.

- One can use an annotation strategy. In this case, the user starts with adding annotations to the program. These annotations must contain the specification but also useful invariants that are hard to deduce (e.g. loop invariants). Then, a tool extracts proof leaf goals to ensure that assertions are satisfied using the semantics of the programming language. Doing so, it automatically applies semantics and structural proof steps. These generated proof leaves can then be discharged using automated provers such as SMT solvers. If the solvers are unable to conclude, the user can add more annotations, as intermediate assertions. This is the approach used by, for instance, Why3 [5].

- Or, the user can verify the program through an interactive proof. There, the tool let the user interactively choose a semantic or structural step and update the remaining program code and pre-condition accordingly. In particular, at any point the user can either weaken the pre-condition or strengthen the post-condition. Some of the steps generate proof leaves to justify their correctness. Such proof leaves can also be proven interactively; they correspond to standard Coq lemmas. This second approach is used, for instance, by CFML [4].

Both approaches have their strengths for representing program with invariants and proofs. The annotation approach anchors the invariants directly at the relevant place in the code, in a very natural way. The interactive proof approach gives a fully interactive construction method. Moreover it produces statements that can be easily related to the formal semantics of the programming language, in a *foundational* way.

In Section 2, we explain how programs with invariants can be described in both approaches. In Section 3, we present our AST data structure as a Coq inductive type. As we will argue, our AST can be either viewed as a *deep embedding* of a proof derivation; or as an annotated source code, furthermore decorated with proof environments and proof terms. All proof leaves remain represented by means of a *shallow embedding*, that is, as Coq proof terms. In Section 4, we explain how instances of our AST can be built using either of the two aforementioned approaches: either via annotation and extraction of proof obligations, or via fully interactive processing. In section §5, we present a *reification* technique for validating that an instance of our AST indeed satisfies the reasoning rules of the program logic. Concretely, the AST is translated into a Coq proof term using lemmas and semantic definitions from CFML. This reification procedure will be helpful in future work to validate the correctness of any source-to-source transformation step.

2 Representing Proofs

As said in the introduction, our AST for representing program with invariants and proofs will be inspired by the two classical approaches for doing program verification, namely program annotation and interactive proofs.

Before getting into the details of our AST, let's review how proofs that an imperative program satisfies a specification are usually made in both styles.

2.1 Program Annotations

The program annotation approach gives a direct link between code and attached invariants. This is an important property for transformations because we need to find which invariants need a modification when updating the code.

Let's consider as example program, the function `ref_move`.

```
let ref_move r n =
    let m = get r in
    let s = ref m in
    set r n;
    s
```

This function can be specified by the following lemma:

$$\forall r, n, m.\ \{r \hookrightarrow m\}\ (\texttt{ref\_move}\ r\ n)\ \{\lambda s.\ r \hookrightarrow n \star s \hookrightarrow m\}$$

This lemma indicates that under the pre-condition that the cell at address `r` contains the value `m`, the function will return a new pointer `s` that points to `m` and that `r` will now point to `n`. Here, the symbol $\star$ denotes the separating conjunction of the logic. It implies in particular that $r \neq s$. Following the annotation approach, we then decorate the source code with separation logic assertions at all key program points, yielding the program below.

```
let ref_move r n =
        $r \hookrightarrow a$
    let m = get r in
        $[m = a] \star r \hookrightarrow a$
    let s = ref m in
        $[m = a] \star r \hookrightarrow a \star s \hookrightarrow a$
    set r n;
        $r \hookrightarrow n \star s \hookrightarrow m$
    s
        $\lambda s.\ r \hookrightarrow n \star s \hookrightarrow m$
```

With such an annotated program, one can run a tool to extract, for each block of instruction between annotation, a set of proof leaf obligations that is sufficient for guaranteeing the specification. Typically, such tools are implemented using weakest precondition (WP) computation. With separation logic style specifications, this technique is used by tools such as VeriFast [12], and Viper [10].

In more complicated examples, the memory state contains representation predicates to express recursive data-structures. For instance, a mutable linked list with head at address p, storing the values described by the list L, could be described by a predicate $p \rightsquigarrow \mathsf{Mlist}\, L$ recursively defined as:

$$\begin{array}{rcl} p \rightsquigarrow \mathsf{Mlist}\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\ & & \mid \mathsf{nil} \;\Rightarrow\; [p = \mathsf{null}] \\ & & \mid x :: L' \;\Rightarrow\; \exists q.\;\; (p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q) \star q \rightsquigarrow \mathsf{Mlist}\, L'. \end{array}$$

To manipulate those representation predicates, one may need to add some ghost instructions in the annotated program specifically to change the vision of the memory state, by unfolding or folding back a Mlist predicate.

The advantage of the annotated program representation for program transformations is that intermediate assertions are attached to program points, so it is easy to tell which assertions need to be updated when local changes are applied to the code. For example, for a transformation that inlines a function, we can specialize the invariants of the body of the function and use these specialized invariants for the inlined code. All the rest of the code can keep its invariants unchanged. As another example, consider the reordering of independent let-bindings. We need to generate new intermediate annotations, but all the annotations before the first let-binding and in the continuation of the second let-binding can be directly reused. In that example, checking that we are able to generate the intermediate annotations effectively proves the absence of dependence between the two bindings. In most cases, such a check is purely syntactic and only use the commutativity of the separating conjunction.

2.2 Interactive Construction of Derivations

The interactive proof strategy provides a way to build a *derivation tree*. These trees perfectly capture the proof steps applied, their required hypotheses, and the links with the program semantics. This is an important property for proof validation, and gives a practical method for the construction of the AST. In this approach, proof are carried out inside an interactive proof assistant, by exploiting reasoning rules of the program logic expressed as lemmas.

Foundational systems such as CFML [4], VST [6] or Iris [7] are built using rules stated as lemmas. These lemmas are proved with respect to the programming language semantics and the logical framework. This foundational approach reduces the trusted code base.

In the derivation trees produced by such frameworks, every node corresponds to a logic rule. Reasoning rules with hypotheses correspond to nodes with sub-trees in the derivation. Interactive verification frameworks typically provide tactics for applying the reasoning rules. They thereby give the illusion to the user of following the code line by line, while building the proof. When invoking these tactics, the user provides the program invariants that cannot be automatically inferred.

Our AST features constructors to match both *structural reasoning rules* and *semantic reasoning rules*. In these reasoning rules, we choose to make the context, written Γ, explicit. This context consists of a list of bindings, for variables and pure facts (i.e., hypotheses). In many presentation, the context, which corresponds to the Coq context, is left implicit, but we find is useful to make context explicit in order to better explain the transitions that affect the context.

Below, the symbol $\vdash$ denotes the heap entailment: $H \vdash H'$ holds iff all heaps satisfying H also satisfy H'. Post-conditions are represented by functions from program return values to heap predicates and we define their point-wise entailment $Q \mathrel{\dot\vdash} Q'$ as $\forall v : \mathsf{Val},\ Q\ v \vdash Q'\ v$. The notation $[P]$ corresponds to the pure heap predicate P. The judgement $\Gamma \vdash \{H\}\ t\ \{Q\}$ asserts that the term t admits H as a pre-condition and Q as a post-condition, in the environment Γ. For details, we refer to Charguéraud's survey [2].

We consider the 4 following structural rules of Separation Logic.[1]

$$\textsc{consequence}\ \dfrac{\Gamma \models H \vdash H' \quad \Gamma \models \{H'\}\ t\ \{Q'\} \quad \Gamma \models Q' \mathrel{\dot\vdash} Q}{\Gamma \models \{H\}\ t\ \{Q\}} \qquad \textsc{frame}\ \dfrac{\Gamma \models \{H\}\ t\ \{Q\}}{\Gamma \models \{H \star H'\}\ t\ \{Q \mathbin{\dot\star} H'\}}$$

$$\textsc{prop}\ \dfrac{\Gamma, \_ : P \models \{H\}\ t\ \{Q\}}{\Gamma \models \{[P] \star H\}\ t\ \{Q\}} \qquad \textsc{exists}\ \dfrac{\Gamma, x : T \models \{H\}\ t\ \{Q\}}{\Gamma \models \{\exists x : T.\, H\}\ t\ \{Q\}}$$

There is one semantic rule per term construct. We show below a few of them.

$$\textsc{val}\ \dfrac{}{\Gamma \models \{\,[\,]\,\}\ v\ \{\lambda r.\,[r = v]\}} \qquad \textsc{let}\ \dfrac{\Gamma \models \{H\}\ t_1\ \{Q'\} \qquad \Gamma, v : \mathsf{Val} \models \{Q'\ v\}\ ([v/x]\,t_2)\ \{Q\}}{\Gamma \models \{H\}\ (\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2)\ \{Q\}}$$

$$\textsc{if}\ \dfrac{\Gamma, \_ : b = \mathsf{true} \models \{H\}\ t_1\ \{Q\} \qquad \Gamma, \_ : b = \mathsf{false} \models \{H\}\ t_2\ \{Q\}}{\Gamma \models \{H\}\ (\mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)\ \{Q\}}$$

3 Definition of an AST for Representing Programs with Invariants and Proofs

We want to benefit from the strength of both program annotation and proof derivation in our AST. From the annotated program vision we want to keep the ability to read back the program and find where to apply a transformation, and which invariants need a change. From the logical derivation vision, we want to keep the ability to give the invariants along the proof, and the foundational relation with the semantics of the programming language. In this section, we present our AST data structure expressed a Coq inductive type, which can be interpreted both as an annotated program and as a derivation.

In order to manipulate and transform invariants, we will need an explicit representation not only of the program syntax, but also of program invariants and program proofs that we can manipulate. Therefore, we need program invariants to be represented using a *deep embedding.* On the contrary, the proof leaves (i.e., proof elements that do not refer to program code) correspond to Coq proof terms, and are encoded using *a shallow embedding.* The next section explains what is a deep embedding and how it compares to a shallow embedding.

3.1 Shallow vs Deep Embeddings

To represent the syntax of a programming language in Coq (or other similar proof assistants), there are two methods. Either one defines the language as a new inductive type with a fixed

[1] We can notice that the PROP and the EXISTS rules are essentially the same, we can exploit this fact encoding $[P]$ by $(\exists \_ : P.\, [])$.

set of constructors, or one directly use Coq functions. The first style is called *deep embedding* and the second one is called *shallow embedding*. The question of whether to use a deep or a shallow embedding actually applies not only to the representation of program syntax, but also to the representation of program invariants, and to the representation of proofs.

In a deep embedding, the predefined list of constructors allows for pattern matching. However, binders and contexts need to be manually managed, for example with a definition of substitution and an explicit encoding of variable (either De Brujn or identifier strings). Similarly, typing is not provided unless explicitly implemented. In a shallow embedding however, since one defines usual Coq terms, binders and typing are directly provided by the proof assistant. As these terms can contain arbitrary elements with arbitrary types, we cannot pattern-match them, and thus have no way to inspect, modify or reuse sub-terms.

Regarding program syntax, a shallow embedding represents program functions as Coq functions, whereas a deep embedding represents it as data constructor carrying name of its arguments and a description of the body. Regarding program invariants, a shallow embedding corresponds to a term in `Prop`, whereas a deep embedding correspond to a value from a custom inductive type. Regarding proofs, shallow embedded proofs correspond to the usual Coq proof terms, whereas deeply embedded proofs correspond to trees in a custom inductive data type, where each constructor correspond to a reasoning rule. Note that deeply embedded proofs are not natively verified.

Recall that our motivation is to define program transformation. Such transformations need to manipulate the program, its invariants and its proofs. Such manipulations can include the identification or the modification of sub-terms that cannot be done with a shallow embedding. We therefore need to use deep embeddings in this work.

In our AST, program syntax is essentially represented using a deep embedding. Program invariants are also represented using a deep embedding. This corresponds to a (simplified) representation of Coq terms, as we show in Section 3.2. For representing the proof terms, we need to be able to manipulate the part of the proofs that correspond to the application of the reasoning rules of the program logic. We represent the application of these rules using a deep embedding. Regarding proof leaves, we have a choice: we can use either a deep or a shallow embedding. It would be technically possible to encode the full calculus of construction in a deep embedded way, as it is done in "Coq in Coq" [1]. However, providing a construction method for such proofs would require reimplementing all Coq tactics. Instead, we choose to represent the proof leaves as *shallowly embedded closed lemmas with deeply embedded arguments*, as we detail in Section 3.4.

In Section 3.3, we will show how we manage the explicit binders in our deep embedded proofs. In Section 3.5, we present our AST as a deep embedding of the proof with strict restrictions on the rules allowed inside the derivation. Those restrictions make the proof look like an annotated program.

3.2 A Deep Embedding of Coq Terms

As we have seen in the previous section, we need a deep embedding for expressing program invariants. Those invariants might contain arbitrary logical formulae. Therefore, we choose to represent them with a simplified deep embedding of Coq terms, as shown below.

```
Inductive coq :=
  | coq_sort (S: sort)
  | coq_val (v: val)
  | coq_var (x: var)
  | coq_app (u1 u2: coq) (* <[ u1 u2 ]> *)
  | coq_forall (z: bind) (Uz P: coq) (* <[ ∀z: Uz, P ]> *)
  | coq_fun (z: bind) (Uz u: coq) (* <[ fun z: Uz, u ]> *)
  | coq_eq (u1 u2: coq) (* <[ u1 = u2 ]> *)
.
```

This deep embedding mostly consists of lambda calculus, with an additional constructor (`coq_val`) to quickly include values of the manipulated programming language. The variables (type `var`) are defined as Coq strings. We prefer explicit naming to De Bruijn indices to ease printing of the terms and retain user given names. This means, however, that we need to compare terms up to alpha equivalence. The binders are defined with the type `bind` and can either contain a fresh variable name, or be anonymous and not introduce any new variable in the context. Free variables are interpreted in a global environment. This environment contains the encoding of the separation logic, and all the relevant predicates for expressing the specification, with definitions close to their CFML analogues. Advanced constructions, such as recursive functions are not handled because we are not yet sure that they will be useful for our application.

For the goals of this paper, we do not need to formalize the typing judgement of these deep embedded Coq terms. As explained in Section 5, proofs and the invariants they refer to will be ultimately type checked during their reification.

Our embedded terms can be written in a human readable manner thanks to the use of notation. For example, the predicate $\exists L.\ [\forall i, 0 \leq i < 5 \rightarrow L[i] = 0] \star p \leadsto \mathsf{Mlist}\, L$ can be written as follows[2]:

```
<[ ∃'L: "list", \[ ∀'i: "nat", 0 ≤ 'i < 5 → "get" 'L 'i = 0 ] ⋆ "MList" 'L 'p ]>.
```

3.3 Manipulation of the Proof Environment

In our AST which represents proof trees, we call *environment* the context of all entities syntactically introduced by the parents of a node and therefore available at that point. Since this AST represents a proof that a program respects a specification, we have several kinds of entities in the context:

- Program variables, which are manipulated by the program instructions;
- Pure facts, which represent mathematical properties true at a given program point;
- Ghost variables, which are additional variables used to state specifications;
- Linear resources, which are linear heap predicates in Separation Logic; each resource can be consumed only once.

These environments can be represented by 3 distinct binding lists. One is used for program variables and is called *program environment*. Another is used for ghost variables and pure facts (which can be seen as pure variables with a type in `Prop`) and is called *pure environment*. The last one is for linear resources and is called *linear environment*.

Each element of the environment can be accessed by a name. This includes linear ressources as in Iris [7]. The type of each environment entity is defined using the previously introduced deep embedding, and can refer to the entities defined before itself (considering that linear resources always come last).

[2] With notation such as `H ∗ H' ≡ "hstar" H H'`, and `∃x: ux, H ≡ "hexists" ux (fun x: ux ⇒ H)`

Our AST will not contain an explicit representation of environments because their contents can be computed by traversing the AST. However, such explicit representation of the environments is used for the construction of our AST. Moreover, since nodes in the AST correspond to reasoning rules, they act on this environment, either by using or adding entities in it.

Environments are extended with bindings when, for example, entering the scope of a let-construct. Consider the program snippet `let p = ref 2 in c`, in a given environment E. Suppose that the body `ref 2` is specified using the post condition $\exists n, [\mathsf{even}\ n] \star p \hookrightarrow n$. In the continuation `c` the environment E is extended with a program variable p, a ghost variable n, a pure fact of type `even` n and its linear environment is replaced by a single binding of type $p \hookrightarrow n$.

To assign names to these bindings, in our AST we need to introduce annotations for the binders, with square brackets to denote pure arguments and curly braces to denote linear resources.

```
let p [n: nat] [Pn: even n] {Hp: p↪n} = ref 2 in c
```

This annotated let-construct, adds `p` as a program variable, `n` and `Pn` to the pure environment and sets the pure environment to only contain `Hp` when processing the `c` continuation branch. Dually, it requires that the let body returns a value `p` such that the environment `[n: nat] [Pn: even n] {Hp: p↪n}` can be built.

Concretely, we will represent these annotations in our AST using the record `binds` shown below.

```
Record binds := {
    binds_vars: list bind;
    binds_pure: list (bind ∗ coq);
    binds_linear: list (bind ∗ coq)
}.
```

Two instances of this `binds` record are also involved in function specifications: one describes the input of the function, including its pre-condition, the other describes its output, including its post-condition. Annotated functions are represented using the record shown below, where `annot_trm` is explained further on.

```
Record annot_fn := {
  annot_fn_input: binds;
  annot_fn_body: annot_trm;
  annot_fn_output: binds
}.
```

3.4 Shallow Embedding of Proof Leaves

Proof leaves appear as hypotheses of the derivation rules applied in our AST, by definition, they correspond to the proof obligations that do not directly involve the program.

Proving these leaves can involve the application of arbitrary mathematical theorems. Therefore, it is very likely that all code transformations would consider all the proof leaves as axioms, and at most reuse, maybe combine, but never modify or inspect what is inside the leaf. Because of that, we choose to encode proof leaves with a shallow embedding in our AST. This brings two main advantages: it removes the need to support most of the features of the calculus of construction in the deep embedded proof rules, and it gives the user the full set of Coq tactics to build the lemmas in the leaves.

However, incorporating a shallow embedded proof inside a deep embedded one brings some issues. Most of the proof leaf obligations refer to the deep embedded proof environment. There is no hope to use the deep embedded environment from a shallow embedded proof since the shallow proof needs to be closed to pass the type checking.

One approach could be to build a shallow lemma taking the environment as argument. This has the main drawback that environment can contain bindings of any type, and we then need to guarantee that all the accesses fetch a variable with the right type, which would be quite heavy or impractical. We prefer to implement shallow embedded proof leaves as Coq functions that take all the relevant elements of the environment as arguments.

Since our shallow embedded goals have arbitrary types, we are forced to store their proofs in dependant pairs. This dependant pair does not enforce by itself a link between the proven lemma type and the deep embedded proof obligation but we will show in Section 5 how to check this property.

Concretely, we represent prove leaves as follows, where we call `pure_lemma` a dependant pair, and `proof_leaf` a shallow embedded lemma instantiated with deep embedded arguments. These arguments are directly given in the Coq term deep embedding..

```
Record pure_lemma := {
    pure_lemma_stmt: Type;
    pure_lemma_proof : pure_lemma_stmt
}.

Record proof_leaf := {
    proof_leaf_lemma: pure_lemma;
    proof_leaf_args: list coq
}.
```

To keep maximal freedom in transformations, lemmas should only take as argument useful hypotheses; indeed, a modified piece of code could have a smaller environment than the original.

3.5 An AST Representing Both a Proof Derivation and an Annotated Program

We are now ready to present an AST that can be viewed either as a deep embedding of a proof derivation, or as an annotated program decorated with its invariants and proofs.

In our AST, we allow incomplete proofs. This means that annotations are optional. This includes, for instance, all the proof leaves.

When interpreting our AST as a proof derivation, all the constructors in our AST correspond to one derivation rule, either structural or semantic. On the contrary, when interpreting our AST as an annotated program, semantic constructors become annotated program syntax, and structural constructors are ghost instructions. In particular, if we ignore all the annotations and skip all the structural rules, we can directly recover the source code of the program.

The AST can be given by the following type, where `val_or_var` contains either a program value or a program variable name and `hyp` and `hvar` are aliases to `var` to respectively denote a name bound in the pure environment and a name bound in the linear environment.

```
Inductive annot_trm :=
    (*** Semantic ***)
  | annot_trm_val (tv: val_or_var)
  | annot_trm_call (fn: var) (args: list val_or_var) (spec: option (hyp*list coq))
```

```
 | annot_trm_let (binds: binds) (body: annot_trm) (cont: annot_trm)
 | annot_trm_if (uvc: val_or_var) (hyp_name: option bind) (brt: annot_trm) (brf:
    annot_trm)
 | ... (* One rule for each construction in the program language *)
  (*** Structural ***)
 | annot_trm_frame (frame: list hvar) (cont: annot_trm)
 | annot_trm_change_pre (pure_clear: list hyp) (linear_in: list hvar)
          (output: binds) (change_lemma: option proof_leaf) (cont: annot_trm)
 | annot_trm_change_post (pure_clear: list hyp) (linear_in: list hvar)
          (output: binds) (change_lemma: option proof_leaf) (cont: annot_trm)
.
```

The semantic constructors correspond both to annotated program constructions and to the application of a semantic rule.

- The constructor annot_trm_val represents a return value at the end of a program expression. It corresponds as well to the VAL semantic rule.

  $$\textsc{val}\qquad \frac{}{\Gamma \models \{\,[\,]\,\}\ v\ \{\lambda r.\,[r = v]\}}$$

  As this rule has no parameters except the return value itself, there is no need for any annotation on the annot_trm_val constructor. However, it only occurs in fully proven programs with an empty linear environment and a matching post-condition: in particular, all linear resources must have been framed out above in the AST.

- The constructor annot_trm_let represents a let-construct. Without annotation its binds argument contains only program variables. However, when annotated, it contains as well new environment bindings for the continuation, that correspond to the environment specification of the body. The annot_trm_let constructor also corresponds to the following LET semantic rule where Q' is given in the binds argument.

  $$\textsc{let}\qquad \frac{\Gamma \models \{H\}\ t_1\ \{Q'\} \qquad \Gamma, v : \mathsf{Val} \models \{Q'\,v\}\ ([v/x]\,t_2)\ \{Q\}}{\Gamma \models \{H\}\ (\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2)\ \{Q\}}$$

- Similarly annot_trm_if, is both an if-construct and the application of the IF rule, where hyp_name is the name given to the new hypothesis p generated by that rule application.

  $$\textsc{if}\qquad \frac{\Gamma, p : b = \mathsf{true} \models \{H\}\ t_1\ \{Q\} \qquad \Gamma, p : b = \mathsf{false} \models \{H\}\ t_2\ \{Q\}}{\Gamma \models \{H\}\ (\mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)\ \{Q\}}$$

- annot_trm_call encodes a function call with a provided list of arguments. It does not have a corresponding rule because it is supposed to use a function specification. This specification can be given in the annotation spec as an instantiated lemma from the pure environment (either local or global).

Structural constructors can appear anywhere in a derivation and correspond to structural rules, or ghost instructions:

- `annot_trm_frame` constructor encodes an application of the FRAME rule.

$$\textsc{frame}\quad \frac{\Gamma \models \{H\}\ t\ \{Q\}}{\Gamma \models \{H \star H'\}\ t\ \{Q \mathbin{\underset{\cdot}{\star}} H'\}}$$

  The `frame` argument contains a list of linear resource bindings that correspond to H in the rule above. H' can be automatically deduced by taking the remaining elements in the current context. Q is similarly deduced by subtracting H' from the output specification at this point.

- There is no constructor for the rules PROP or EXISTS since we apply them implicitly when adding bindings to the pure environment.

- To be able to pinpoint easily which parts of the environment are modified when using the CONSEQUENCE rule, we split it into two different constructors `annot_trm_change_pre` and `annot_trm_change_post`, one for each side of the Hoare triple. Moreover, we identify with `linear_in` which subset of the linear environment will be modified, and with `pure_clear` which pure bindings will be cleared in the process, while giving binding names to all the introduced entities with `output`. This corresponds to the following rules that can be derived from the CONSEQUENCE rule, where Γ is a pure environment, and Γ' is a subset of Γ.

$$\textsc{change-pre}\quad \frac{\Gamma \models H \vdash H' \quad \Gamma' \models \{H' \star R\}\ t\ \{Q\}}{\Gamma \models \{H \star R\}\ t\ \{Q\}} \qquad \textsc{change-post}\quad \frac{\Gamma' \models \{H\}\ t\ \{Q' \mathbin{\underset{\cdot}{\star}} R\} \quad \Gamma \models Q' \mathrel{\vdash\!\!\cdot}\ Q}{\Gamma \models \{H\}\ t\ \{Q \mathbin{\underset{\cdot}{\star}} R\}}$$

  This is the only construction introducing a shallow proof leaf, to prove the heap entailment hypothesis. With an empty `linear_in` list, it can be used to add arbitrary assertions in the environment at any program point, as long as it is provable.

With Coq notation, and if we mask proof leaves, we can actually print a term of type `annot_trm` as an annotated program.

This representation of a proof derivation is a good candidate for the kind of proof manipulation required by source-to-source transformations because by replacing any sub-tree in a derivation with another valid sub-tree of the same specification, we maintain the validity of the full tree, while changing the source code of a branch. It means that transformations can be local and ignore all the derivation rules above a relevant application point.

The AST does not by itself enforce its validity: at this stage, nothing checks the typing of the environment variables or that proof leaves assert what they should. This is not a problem since we show in the two next sections how to build a self-consistent tree and how to verify its validity.

4 Building Annotated Programs

In the previous section, we described an AST that can be seen either as an annotated program or as a proof derivation. This section show that we can use these two views to build the trees. Given a program annotated with invariants but without proofs, one may want to build the same annotated program with its proof leaves filled in, using the annotation and extraction workflow presented in section §2.1. Besides, given a program without any annotation, one may want to *interactively* build, in CFML fashion, the proof derivation that ensures the program follows its specification, as presented in section §2.2.

We explain how to perform both kinds of tasks, interactively in Coq. We start by presenting the CFML approach. Indeed, we will see how the case where program invariants are already present can also be handled with CFML style tactics, with the only difference being taking into account the existing invariants.

4.1 Building Interactively

We first describe how to interactively build a derivation, in CFML fashion, since we reuse most of the components in our proposed system.

From the user point of view, a CFML proof follow a simple workflow. At each step, the theorem prover shows the program that still need to be processed along with the environment at this program point. It also shows what invariants need to hold after the execution. Seeing these goals, the user will apply CFML tactics to apply structural of semantic rules at the current program point. These tactics produce sub-goals corresponding either to sub-programs verification or proof leaves.

Internally, these CFML tactics call shallow embedded lemmas. These lemmas produce a Coq proof term corresponding to a proof in the shallow embedding of Separation Logic and program semantics. Since this output proof term is in `Prop`, there is no simple way to inspect it, to extract an instance of our AST. Technically Ltac manipulation, to transfer this shallow proof in a deep embedding of Coq is possible (as it is done in the MetaCoq [13] plugin for instance), but the proof terms would have no canonical form anyway, so we cannot convert them into our AST.

Instead, we provide our own set of tactics. These new tactics display the same kind of goals as CFML and generate the same kind of proof obligations. However, their output is an AST in our format. This AST corresponds to a deep embedding of the derivation, except for proof leaves which are shallowly embedded.

The challenge in designing those tactics is that the proof obligations that we would like to show to the user are propositions, in `Prop`. However, their output is an instance of our AST, in `Type`. There is no fundamental difficulty, we simply need to find an appropriate way to state the proof obligations on which the tactics apply. To that end, we introduce a predicate, written `refine_pred` $E\ t\ F\ \hat{t}$, where t is the input program, E and F correspond to pre- and post-conditions represented as environments with named elements; and $\hat{t}$ is an *evar* that gets refined along the proof. When the proof is completed the $\hat{t}$ variable gets instantiated with the desired output.

We display the `refine_pred` $E\ t\ F\ \hat{t}$ predicate to the user using a notation that hides $\hat{t}$. What remains looks seemingly like a CFML proof obligation. We show below an example where E and F are displayed with a notation that separates the three components of the environments (program variable, pure facts, and linear resources).

```
PRE
   ’x, ’p
   ———
   [’n : int]
   ["Px": ’x = 3]
   ["Pn": ’n = 1]
   ———
   {"Pp": ’p ↪’n}
```

```
CODE
   let ’y = get ’p in
   ’x + ’y
POST
   ’r
   ———
   ["Pr": 4 = ’r]
   ———
   {"Pp": ’p ↪1}
```

This example corresponds to the following Hoare triple:

$$\{\exists n.[x=3] \star [n=1] \star p \hookrightarrow n\}\ \texttt{let y = get p in x + y}\ \{\lambda r.[4=r] \star p \hookrightarrow 1\}.$$

During an interactive proof all goals except proof leaves are displayed using such notation for `refine_pred`.

Reasoning steps of separation logic in such proofs are carried out using CFML style tactics. For instance, the tactic `xlet ["Py": ’y = 3] {"Pp": ’p ↪1}` applies to the proof obligation shown above. As in CFML, this tactic generates two sub-goals (one for the body, one for the continuation). The argument provided to `xlet` describes the post-condition for the body. It is optional, and if not provided, an evar is introduced for this post-condition, which like in CFML gets instantiated during the verification of the body.

Internally, `xlet` applies to a goal of the form `refine_pred` E (`let x = t_body in t_cont`) F $\hat{t}$. It produces two sub goals `refine_pred` E `t_body` E' $\hat{t_b}$ and `refine_pred` E'' `t_cont` F $\hat{t_c}$, where $\hat{t_b}$ and $\hat{t_b}$ are newly generated evars, and E'' corresponds to the concatenation of E with E' in the program and pure environments and only E' in the linear environment. While doing so, it instantiates $\hat{t}$ as (`let x E’ =` $\hat{t_b}$ `in` $\hat{t_c}$).

At some points in the proof, the user reaches proof leaves, that is, proof obligations not establishing Hoare triples. Such proof leaves are represented in our AST using the type `proof_leaf` introduced in Section 3.4. Recall that `proof_leaf` contains a *closed* shallowly embedded lemma and a list of deeply embedded arguments.

Our framework present proof leaves obligations as the predicate `prove` E G $\hat{p}$, where E is an environment without linear ressources, G is the deep embedding of the property we want to prove and $\hat{p}$ is an evar for the desired value of type `proof_leaf`. We provide a tactic `xprove` that applies to a goal of the form `prove` E G $\hat{p}$. This tactic takes as argument a subset of the keys of E that corresponds to the hypotheses the user wants to exploit to establish a proof of G. The tactic then leaves a standard Coq proof obligation of a shallow version of G obtained using the reification mechanism described further on in Section 5. The resolution of this goal yields a proof term, which ends up being recorded in our AST.

4.2 Completing an Annotated Program with Proofs

In the previous section, we saw how to build an AST fully annotated with invariants and proofs starting from an unannotated program. In this section, we explain how to build a similar AST starting from a program annotated with invariants. Concretely, we need to fill the missing proof leaves. This consists of extracting proof obligations and proving them, like e.g. in Why3.

Our implementation leverages the same mechanisms based on `refine_pred` as in the previous section. The key difference is that the source program contains invariants. In particular, there is no need to provide a post-condition when calling tactics such as `xlet` because this post-condition already appears as a program annotation. More generally, CFML style tactics, can all be invoked without arguments simply following the structure of the annotated program. As a result, all the tactics processing the code can be automatically handled.

A number of these tactics produce proof leaves. These proof leaves correspond to what is called extracted proof obligations in the tools taking the annotation and extraction approach.

There is another application of refining a partially complete AST into a fully complete one, related to program transformations that we intend to work on in the long term. The user may wish to introduce an arbitrary piece of code into an already verified program. Then, thanks to this construction method, we can extract proof obligations concerning exclusively the new piece of code.

5 A Reification Process to Validate our AST

The AST we presented in Section 3, does not include any guarantees about the validity of the proof it represents. Indeed, nothing prevents using an unbound environment variable or using a rule that do not apply at a given program point. In Section 4, we gave a methodology to build derivations. The derivations built that way are in principle well formed. However, their validity directly depends on the implementation of the construction tactics. Exactly like in Coq, we do not want our tactics to be inside the trusted code base, since they can have rather complicated implementations, and therefore may contain bugs.

A standard approach to provide a stronger validity guarantee on deep embedded proofs is to define a validity predicate. Such predicate ensures that each node in a derivation correspond to valid rule application. This is what we initially tried, but we quickly noticed that establishing and maintaining a validity predicate across transformations would require a lot of work.

To keep strong guarantees without too much effort, we introduce in this section a reification procedure. This will translate a complete proof derivation expressed in our AST into a standard shallow embedded CFML lemma. The obtained Coq proof is therefore directly using the defined semantics of our programming language, in a foundational way.

During reification, both the proof derivation and the invariants represented inside are translated in their shallow embedded counterparts. For specifying the type of the root of the derivation, we also extract an unannotated version of the program syntax as a CFML deeply embedded program term.

One technical aspect of the reification process is that we need to maintain the mapping between binding names from the AST environment, and corresponding Coq variables that appear in the produced proof term. This can be done by maintaining, during the reification process, a table between names in the deep embedding and identifiers in the shallow version. Since invariants can also refer to a global environment, we also need to provide a Coq implementation for each of these global terms.

When reaching proof leaves, our reification procedure will use the pure lemma stored inside the leaf and call it with reified versions of its deeply embedded arguments. This is enough to force the Coq type checker to verify that the lemma indeed proves the obligation it is applied on.

Concretely, the reification process is implemented as a recursive Ltac function parameterized by the mapping between binding names and Coq variables. On the proof parts, it applies rules expressed as CFML lemmas simply following the derivation. For invariants, it follows the structure of the deeply embedded term to progressively refine the corresponding shallow embedded Coq term.

In order to improve the user experience, when extracting program invariants, we want to reuse the names of the deeply embedded binders for the shallow embedded ones. This is possible only if our Ltac procedure can convert a Coq string into a Coq identifier. Such translation cannot be expressed natively in Ltac but can be implemented using a Coq plugin or a piece of Ltac2 code.

In the end, the Ltac function implementing the reification process is not part of the trusted code base. What goes in the trusted base is the Coq kernel, the language semantics, and the definition of triples regarding the semantics. In addition, for each program, one needs to trust that the shallow embedding of the specification is correct. This shallow specification can be read as the type of the reified verification proof, but it also corresponds to a shallow embedded version of the specification that appears at the root of the annotated AST.

6 Conclusion

In this paper, we aimed at providing a representation for programs annotated with their invariants and proofs, such that we can easily manipulate them.

We achieved that using a deep embedding of the proof derivation with shallow proof leaves and explicit environment manipulations. We provided a method for constructing these annotated programs either interactively or by filling missing proofs.

In the long term, we want to propose an interactive framework for code transformation that preserves proofs and can be used for source-to-source optimization. Using it, one should be able to write a naive version of a program, verify it, and then progressively transform it into a highly optimized code. After the last step we should be able to provide a proof of the final program. By combining such framework with verified compilers such as CakeML [8] or CompCert [9], we could then obtain binaries that are both trustworthy and efficient.

References

[1] Bruno Barras and Benjamin Werner. Coq in coq. *Available on the WWW*, 1997.

[2] Arthur Charguéraud. Separation logic for sequential programs (functional pearl). *Proc. ACM Program. Lang.*, 4(ICFP), August 2020.

[3] Arthur Charguéraud, Begatim Bytyqi, Damien Rouhling, and Yann A Barsamian. OptiTrust: an Interactive Framework for Source-to-Source Transformations. working paper or preprint, September 2022.

[4] Arthur Charguéraud. Characteristic Formulae for the Verification of Imperative Programs. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, page 418–430, New York, NY, USA, 2011. Association for Computing Machinery.

[5] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792, pages 125–128, March 2013.

[6] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle Semantics for Concurrent Separation Logic. In Sophia Drossopoulou, editor, *Programming Languages and Systems*, page 353–367, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[7] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[8] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 179–191, New York, NY, USA, 2014. Association for Computing Machinery.

[9] Xavier Leroy. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52(7):107–115, jul 2009.

[10] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, page 41–62, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[11] O'Hearn, Reynolds, and Yang. Local reasoning about programs that alter data structures. In *CSL: 15th Workshop on Computer Science Logic*. LNCS, Springer-Verlag, 2001.

[12] Pieter Philippaerts, Jan Tobias Mühlberg, Willem Penninckx, Jan Smans, Bart Jacobs, and Frank Piessens. Software verification with VeriFast: Industrial case studies. *Science of Computer Programming*, 82:77–97, 2014. Special Issue on Automated Verification of Critical Systems (AVoCS'11).

[13] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, 64(5):947–999, Jun 2020.