Certified Abstract Interpretation with Pretty-Big-Step Semantics

Martin Bodin, Thomas Jensen, Alan Schmitt

Certified Abstract Interpretation with Pretty-Big-Step Semantics

Martin Bodin Thomas Jensen Alan Schmitt

Inria
Rennes, France
name.surname@inria.fr

Abstract

This paper describes an investigation into developing certified abstract interpreters from big-step semantics using the Coq proof assistant. We base our approach on Schmidt's abstract interpretation principles for natural semantics, and use a pretty-big-step (PBS) semantics, a semantic format proposed by Charguéraud. We propose a systematic representation of the PBS format and implement it in Coq. We then show how the semantic rules can be abstracted in a methodical fashion, independently of the chosen abstract domain, to produce a set of abstract inference rules that specify an abstract interpreter. We prove the correctness of the abstract interpreter in Coq once and for all, under the assumption that abstract operations faithfully respect the concrete ones. We finally show how to define correct-by-construction analyses: their correction amounts to proving they belong to the abstract semantics.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification

Keywords Abstract Interpretation; Big-step semantics; Coq

1. Introduction

Static program analyzers are complex pieces of software that are hard to build correctly. Abstract interpretation [9] is a theory for relating semantics of programming languages which has proven extremely powerful for proving the correctness of static program analyses. Programming the theory of abstract interpretation in a proof assistants such as Coq has led to *certified abstract interpretation*, where static analyzers are developed alongside their correctness proof. This significantly increases the confidence in the analyzers so produced.

In this paper, we study the use of big-step operational semantics as a basis for certified abstract interpretation. Big-step semantics is a semantic framework that can accommodate fine-grained operational features while at the same time keeping some of the compositionality of denotational semantics. Furthermore, it has been shown to be able to handle large-scale definitions of programming languages, as witnessed by the recent JSCert semantics of JavaScript [3]. The latter development is the direct motivation for the work reported here. We present a general Coq framework [4] to build abstract semantics correct by construction out of minimum proof effort.

As JSCert is written in a *pretty-big-step* (PBS) semantics [7], we naturally decided to use it as foundation for this work. PBS semantics are convenient because they reduce duplication in the definition of the language and because they have a constrained format. These constraints allowed us to define our framework in the most general way, without committing to a particular language—see Sections 2 and 3.

1.1 Abstract Interpretation of Natural Semantics

The principles behind abstract interpretation of natural (big-step) semantics were studied by Schmidt [20]. They form the starting point for the mechanization proposed here, although the final result deviates in several ways from Schmidt's proposal (see Section 7).

Intuitively, abstract interpretation of big-step semantics consists of the following steps:

- define *abstract executions* as derivations over abstract domains of program properties;
- show abstract executions are correct by relating them to concrete executions;
- program an *analyzer* that builds an abstract execution among those possible. Such an analyzer is correct by construction, but its precision depends on the abstract execution returned.

The first step in Schmidt's formal development is a precise definition of the notion of semantic tree. These are the derivation trees obtained from applying the inference rules of a big-step semantics to a term. This results in *concrete judgments* of the form $t, E \Downarrow r$.

The abstract interpretation of this big-step semantics starts with a Galois connection (in the form of a correctness relation *rel*) between concrete and abstract domains of base values (see Section 5.1 for an example). This relation extends in the standard way to composite data structures, to environments, and to judgments of the form $t, E \Downarrow r$. An *abstract semantic tree* is then taken to be a semantic tree where the values at the nodes are drawn from the abstract domain. A central step in the development is the extension of the correctness relation to derivation trees. Written rel_U, this relation states that a (concrete) derivation tree is related to an abstract derivation tree if the conclusions are related by *rel*, and that for every concrete sub-derivation there exists an abstract sub-derivation that is rel_U-related to it. This leads to a way of proving correctness of an abstract interpretation, by checking that each rule from the concrete semantics has a corresponding rule in the abstract semantics.

Our approach is similar: concrete and abstract executions are assemblages of *rules*. The rules and the syntax of terms are shared

CPP '15, January 13--14, 2015, Mumbai, India.

http://dx.doi.org/10.1145/2676724.2693174

between the concrete and abstract versions. The difference between the two versions is twofold: on the *semantic* domains (contexts in which the execution occurs, such as state, and results), and the way the rules are assembled. An important feature of our approach is that the soundness of the approach depends neither on the specific abstract domains chosen, nor on the semantics itself, as long as the domains correctly abstract operations on the concrete domain.

Abstract derivation trees may be infinite. Convergence of an analysis is obtained by identifying an invariant in the derivation tree. Whichever invariant the analysis uses, it is correct if the returned derivation belongs to the set of abstract derivations.

To summarize the parametricity of our approach, we describe the steps required to produce a certified analysis. First, our framework is parametric in the language used, which thus must be defined as a PBS semantics based on transfer functions (see Sections 2 and 3). Next, the framework is also parametric in the abstract domains, which must also be defined, along with the abstract transfer functions. Once these functions are shown to correctly abstract the concrete transfer functions, a correct-by-construction abstract semantics is automatically defined. Finally, an analysis must be developed. The fact that the result of the analysis belongs to the abstract semantics is a witness that it is correct.

1.2 Organization of the Paper

The paper is organized as follows. We first review the principles behind PBS operational semantics and show its instantiation on a simple imperative language in Section 2. In Section 3 we make a detailed analysis of PBS rules and propose a dependently-typed formalization of their format. The representation of this formalization in Coq is described in Section 4. Section 5 describes the representation of abstract domains and explains how PBS rules can be abstracted in a systematic fashion which facilitates the proof of correctness. Section 6 demonstrates the use of the abstract interpretation for building additional reasoning principles and program verifiers. Section 7 discusses related work and Section 8 concludes and outlines avenues for further work based on our certified abstract interpretation.

2. Pretty-big-step Semantics

Pretty-big-step semantics (PBS) is a flavor of big-step, or natural, operational semantics which directly relates terms to their results. PBS semantics was proposed by Charguéraud [7] with the purpose of avoiding the duplication associated with big-step semantics when features such as exceptions and divergence are added. In this section, we introduce PBS semantics through a simple While language with an abort mechanism. To simplify the presentation, we restrict the set of values to the integers, and let the value 0 be considered as "false" in the branching statements *if* and *while*.

We give some intuition of how a pretty-big-step semantics works through a simple example: the execution of a while loop. In a big step semantics, the while loop inference rules have one or three premises. In both cases, the first premise is the evaluation of the condition. If it returns 0, there is no further premise. If it returns another number, the other two premises are the evaluation of the statement and the evaluation of the rest of the loop. In the following, the evaluation of expressions returns a value, whereas the evaluation of statements returns a modified state. Writing E for states, such rules would be written as follows.

$$\textsc{WhileFalse}\quad \frac{e, E \Downarrow 0}{\mathit{while}\ e\ s, E \Downarrow E}$$

$$\textsc{WhileTrue}\quad \frac{e, E \Downarrow v \qquad s, E \Downarrow E' \qquad \mathit{while}\ e\ s, E' \Downarrow E''}{\mathit{while}\ e\ s, E \Downarrow E''} \quad v \neq 0$$

In the pretty-big-step approach, only one sub-term is evaluated in each rule, and the result of the evaluation is gathered, along with the state, in a new construct called a *semantic context*. New terms, called *extended terms*, are added to the syntactic constructs. For instance, the first reduction for the *while* loop is as follows.

$$\textsc{While}\quad \frac{\mathit{while}_1\ e\ s, \mathit{ret}\ E \Downarrow o}{\mathit{while}\ e\ s, E \Downarrow o}$$

The *ret* construction signals that there was no error, its role will be detailed below. The extended term while_1 indicates that the loop has been entered. It reduces as follows.

$$\textsc{While1}\quad \frac{e, E \Downarrow o \qquad \mathit{while}_2\ e\ s, (E, o) \Downarrow o'}{\mathit{while}_1\ e\ s, \mathit{ret}\ E \Downarrow o'}$$

This rule says: if the semantic context is a state E that is not an error, then reduce the condition e in the semantic context E, and bundle the result of that evaluation with E as semantic context for the evaluation of the extended syntactic term $\mathit{while}_2\ e\ s$.

The term $\mathit{while}_2\ e\ s$ can in turn be evaluated using one of two rules. If the result that was bundled into the semantic context is the value 0, then return the current state.

$$\textsc{While2False}\quad \frac{}{\mathit{while}_2\ e\ s, (E, \mathit{val}\,0) \Downarrow \mathit{ret}\ E}$$

Otherwise, evaluate s and use its result as semantic context to continue the loop with the term $\mathit{while}_1\ e\ s$.

$$\textsc{While2True}\quad \frac{s, E \Downarrow o \qquad \mathit{while}_1\ e\ s, o \Downarrow o'}{\mathit{while}_2\ e\ s, (E, \mathit{val}\ v) \Downarrow o'} \quad v \neq 0$$

Putting it all together, Figure 1 depicts a full derivation of one run of a loop, where $k \neq 0$.

The set of terms for our language is defined in Figure 2a. Terms t are either expressions e, extended expressions e_x, statements s, or extended statements s_x. (Ordinary) expressions and statements form the standard While language, with an added abort statement **abort**. An example of an extended expression is $+_1\ e_2$ that indicates the left expression of $+\ e_1\ e_2$ has been computed, and it is now the turn of e_2 to be computed. An example of an extended statement is $\mathit{if}_1\ s_1\ s_2$ that indicates the expression forming the condition has been evaluated; and the statement to evaluate depends on that result, present in the semantic context.

Evaluation of terms uses the following semantic domains.

- $\mathit{val} = \mathbb{Z}$;
- $\mathit{error} = \{\mathit{Err}\}$;
- $\mathit{env} = \mathit{var} \to_f \mathit{val}$, the finite maps from *var* to *val*;
- $\mathit{out}_e = \mathit{val} + \mathit{error}$, the expression outputs;
- $\mathit{out}_s = \mathit{env} + \mathit{error}$, the statement outputs.

Thus, the evaluation of an expression or an extended expression will either produce a value $v \in \mathit{val}$ or produce an error. Evaluation of a statement or an extended statement will produce a new environment or an error. To differentiate between a value element of out_e and a value of *val*, the former will be noted $\mathit{val}\ v$ and the latter simply v. We proceed similarly for environments, where $\mathit{ret}\ E \in \mathit{out}_s$.

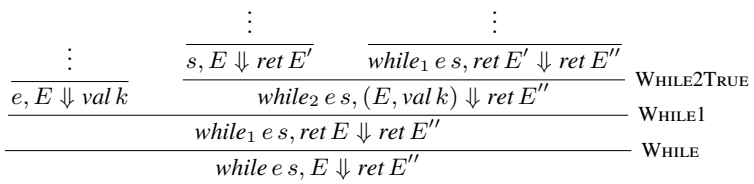


Figure 1: PBS reduction of a while loop

The semantic rules are given in Figure 3. To see how the extended terms work, consider the rule $\textsc{Add}_1\,(e)$ for evaluating the extended expression $+_1\ e$.

$$\frac{\textsc{Add}_1\,(e)}{\dfrac{e, E \Downarrow o \qquad +_2, (v_1, o) \Downarrow o'}{+_1\ e, (E, \mathit{val}\, v_1) \Downarrow o'}}$$

The evaluation of $+_1\ e$ is done with a semantic context comprised of an environment E and the output of evaluating the first operand. This rule pattern-matches the latter, requiring it to be a value $\mathit{val}\, v_1$ and extracting the actual value v_1. If $+_1\ e$ is evaluated with a semantic context of the form (E, Err), then $\textsc{Add}_1\,(e)$ does not apply. In that case, the rule $\textsc{AbortE}\,(+_1\ e)$ applies (see Figure 2d), which propagates the error.

In the case there was no error, the semantics follows rule $\textsc{Add}_1\,(e)$ and evaluates e to obtain an output for the second operand. It then constructs another extended expression $+_2$ and evaluates it with a semantic context that includes the value v_1 the output o.

If this output o is an error, only the rule $\textsc{AbortE}\,(+_2)$ applies and the error is propagated. Otherwise, the rule $\textsc{Add}_2$ applies.

$$\frac{\textsc{Add}_2}{\mathit{add}\,(v_1, v_2) \rightsquigarrow v}{+_2, (v_1, \mathit{val}\, v_2) \Downarrow \mathit{val}\, v}$$

This rule is called an axiom as none of its premises mention a derivation about $\Downarrow$. It only performs a local computation, denoted by $\rightsquigarrow$, and returns the result.

The PBS format only requires a few rules to propagate errors, even though they may appear at any point in the execution.

3. Formalization of Rule Schemes

The mechanization of the abstract interpretation of PBS operational semantics is based on a careful analysis of the rule formats used in these semantics. Traditional operational semantics are defined inductively with rules (or, more precisely, rule schemes) of the form

$$\frac{\textsc{Name}}{\dfrac{t_1, \sigma_1 \Downarrow r_1 \quad t_2, \sigma_2 \Downarrow r_2 \quad \ldots}{t, \sigma \Downarrow r}}\ \textit{side-conditions}$$

explaining how term t evaluates in a state σ to a result r. There are several implicit relations between the elements of such rule schemes that we make explicit, in order to provide a functional representation for them.

First, we describe the types of the components of $t, \sigma \Downarrow r$. The first component t is a *syntactic term* of type *term*. It is the program being evaluated. The second component σ is a *semantic context*. It contains the information required to evaluate the program, such as the current state. Its type depends on the term being evaluated: we have $\sigma \in \mathit{st}\,(t)$. For most terms, the semantic context in our concrete semantics is an environment E (see Figure 3). The exceptions are for extended terms that also need information from the previous computations. For instance, the term $+_1\ e$ needs both an environment E and a result o as semantic context. Finally, the third component r is the result of the evaluation of t in context σ. Its type also depends on t: excluding errors, expressions return values whereas instructions return environments. It is written $\mathit{res}\,(t)$.

Second, rules are identified not only by their name but also by syntactic subterms. For instance, a rule to access the variable x is identified by $\textsc{Var}\,(x)$, whereas the one for variable y is identified by $\textsc{Var}\,(y)$. Similarly, a rule for a "while" loop with condition e and body s may be identified by $\textsc{While}\,(e, s)$. Identifiers are designed such that they uniquely determine the term to which the rule applies.

Formally, a PBS semantics carries a set of rule identifiers $\mathcal{I}$ and a function that maps rule identifiers to actual rules (the type Rule_i is described below).

$$\mathit{rule} : (i \in \mathcal{I}) \to \mathit{Rule}_i$$

They also provide a function $\mathfrak{l}$ that maps rule identifiers to the syntactic term to which the rule applies.

$$\mathfrak{l} : \mathcal{I} \to \mathit{term}$$

For instance, for the rule $\textsc{Var}\,(x)$, we have $\mathfrak{l}_{\textsc{Var}(x)} = x$.

Third, rules have *side-conditions*. We impose a clear separation between these conditions and the hypotheses on the semantics of subterms made above the inference line. The conditions involve the rule identifier i and the semantic context σ and are expressed in a predicate *cond* which states whether rule i applies in the given context σ. For a simple example: two rules can apply to the term x, a variable, depending on whether this variable is defined or not in the given environment E: it is either the look-up rule $\textsc{Var}\,(x)$ or the error rule $\textsc{VarUndef}\,(x)$.

$$\frac{\textsc{Var}(x)}{\dfrac{E[x] \rightsquigarrow v}{x, E \Downarrow \mathit{val}\, v}}\ x \in \mathrm{dom}(E) \qquad \frac{\textsc{VarUndef}(x)}{x, E \Downarrow \mathit{Err}}\ x \notin \mathrm{dom}(E)$$

The predicate *cond* has the type

$$\mathit{cond} : (i \in \mathcal{I}) \to \mathit{st}\,(\mathfrak{l}_i) \to \mathit{Prop}$$

Finally, the general big-step format allows any number of hypotheses above the inference line. The pretty-big-step semantics restricts this to one of three possible formats: axioms (zero hypotheses), rules with one inductive hypothesis, and rules with two inductive hypotheses, respectively written Ax_i, $R_{1,i}$ or $R_{2,i}$ for a rule identified by i.

Syntactic Aspects of Rules To summarize, the function $\mathit{type} : \mathcal{I} \to \{\mathit{Ax}, R_1, R_2\}$ returns the format (axiom, rule 1, or rule 2) of the rule identified by $i \in \mathcal{I}$, and $\mathfrak{l} : \mathcal{I} \to \mathit{term}$ returns the actual syntactic term evaluated by a rule. To evaluate a rule, one needs to specify which terms to inductively consider (syntactic aspects) and how the semantic contexts and results are propagated (semantic aspect). We first describe the former.

In format 1 rules, i.e., rules with one hypothesis, the current computation is redirected to the computation of the semantics of another intermediate term (often a sub-term). We thus define a

$$
\begin{array}{rcl}
t & ::= & e \\
& | & s \\
& | & e_x \\
& | & s_x
\end{array}
\qquad
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & +\, e_1\ e_2 \\
\\
e_x & ::= & +_1\ e_2 \\
& | & +_2
\end{array}
\qquad
\begin{array}{rcl}
s & ::= & \mathit{skip} \\
& | & x := e \\
& | & s_1; s_2 \\
& | & \mathit{if}\, e\, s_1\, s_2 \\
& | & \mathit{while}\, e\, s \\
& | & \mathbf{abort}
\end{array}
\qquad
\begin{array}{rcl}
s_x & ::= & x :=_1 \\
& | & ;_1\ s_2 \\
& | & \mathit{if}_1\ s_1\ s_2 \\
& | & \mathit{while}_1\ e\ s \\
& | & \mathit{while}_2\ e\ s
\end{array}
$$

(a) Terms and Extended Terms

$$
\begin{array}{rcl}
\mathit{st}\,(e) & = & \mathit{env} \\
\mathit{st}\,(s) & = & \mathit{env} \\
\mathit{st}\,(+_1\ e_2) & = & \mathit{env} \times \mathit{out}_e \\
\mathit{st}\,(+_2) & = & \mathit{val} \times \mathit{out}_e
\end{array}
\qquad
\begin{array}{rcl}
\mathit{st}\,(x :=_1) & = & \mathit{env} \times \mathit{out}_e \\
\mathit{st}\,(;_1\ s_2) & = & \mathit{out}_s \\
\mathit{st}\,(\mathit{if}_1\ s_1\ s_2) & = & \mathit{env} \times \mathit{out}_e \\
\mathit{st}\,(\mathit{while}_1\ e\ s) & = & \mathit{out}_s \\
\mathit{st}\,(\mathit{while}_2\ e\ s) & = & \mathit{env} \times \mathit{out}_e
\end{array}
$$

(b) Definition of *st*

The (dependent) type of semantic contexts.

$$
\begin{array}{rcl}
\mathit{res}\,(e_x) & = & \mathit{out}_e \\
\mathit{res}\,(s_x) & = & \mathit{out}_s \\
\mathit{res}\,(e) & = & \mathit{out}_e \\
\mathit{res}\,(s) & = & \mathit{out}_s
\end{array}
$$

(c) Definition of *res*

The type of results.

$$
\begin{array}{rcl}
\mathit{abort}\,(\mathit{Err}) & = & \mathit{True} \\
\mathit{abort}\,(\mathit{ret}\ E) & = & \mathit{False} \\
\mathit{abort}\,(E, \mathit{Err}) & = & \mathit{True} \\
\mathit{abort}\,(E, \mathit{val}\ v) & = & \mathit{False} \\
\mathit{abort}\,(v, \mathit{Err}) & = & \mathit{True} \\
\mathit{abort}\,(v, \mathit{val}\ v) & = & \mathit{False}
\end{array}
$$

(d) Definition of *abort*

The *abort* predicate controls the rules $\textsc{AbortE}\,(e_x)$ and $\textsc{AbortS}\,(s_x)$ of Figure 3.

Figure 2: Concrete Semantics Definitions

$$
\frac{}{e_x, \sigma \Downarrow \mathit{Err}}\ \mathit{abort}\,(\sigma)\ \textsc{AbortE}(e_x)
\qquad
\frac{}{s_x, \sigma \Downarrow \mathit{Err}}\ \mathit{abort}\,(\sigma)\ \textsc{AbortS}(s_x)
\qquad
\frac{}{\mathbf{abort}, E \Downarrow \mathit{Err}}\ \textsc{Abort}
\qquad
\frac{}{c, E \Downarrow \mathit{val}\ c}\ \textsc{Cst}(c)
\qquad
\frac{E[x] \rightsquigarrow v}{x, E \Downarrow \mathit{val}\ v}\ x \in \mathrm{dom}(E)\ \textsc{Var}(x)
$$

$$
\frac{}{x, E \Downarrow \mathit{Err}}\ x \notin \mathrm{dom}(E)\ \textsc{VarUndef}(x)
\qquad
\frac{e_1, E \Downarrow o \qquad +_1\ e_2, (E, o) \Downarrow o'}{+\ e_1\ e_2, E \Downarrow o'}\ \textsc{Add}(e_1, e_2)
\qquad
\frac{e, E \Downarrow o \qquad +_2, (v_1, o) \Downarrow o'}{+_1\ e, (E, \mathit{val}\ v_1) \Downarrow o'}\ \textsc{Add}_1\,(e)
\qquad
\frac{\mathit{add}\,(v_1, v_2) \rightsquigarrow v}{+_2, (v_1, \mathit{val}\ v_2) \Downarrow \mathit{val}\ v}\ \textsc{Add}_2
$$

$$
\frac{}{\mathit{skip}, E \Downarrow \mathit{ret}\ E}\ \textsc{Skip}
\qquad
\frac{e, E \Downarrow o \qquad x :=_1, (E, o) \Downarrow o'}{x := e, E \Downarrow o'}\ \textsc{Asn}(x, e)
\qquad
\frac{E[x \mapsto v] \rightsquigarrow E'}{x :=_1, (E, \mathit{val}\ v) \Downarrow \mathit{ret}\ E'}\ \textsc{Asn}_1\,(x)
\qquad
\frac{s_1, E \Downarrow o \qquad ;_1\ s_2, o \Downarrow o'}{s_1; s_2, E \Downarrow o'}\ \textsc{Seq}(s_1, s_2)
\qquad
\frac{s_2, E \Downarrow o}{;_1\ s_2, \mathit{ret}\ E \Downarrow o}\ \textsc{Seq}_1\,(s_2)
$$

$$
\frac{e, E \Downarrow o \qquad \mathit{if}_1\ s_1\ s_2, (E, o) \Downarrow o'}{\mathit{if}\, e\, s_1\, s_2, E \Downarrow o'}\ \textsc{If}(e, s_1, s_2)
\qquad
\frac{s_1, E \Downarrow o}{\mathit{if}_1\ s_1\ s_2, (E, \mathit{val}\ v) \Downarrow o}\ v \neq 0\ \textsc{If1True}(s_1, s_2)
\qquad
\frac{s_2, E \Downarrow o}{\mathit{if}_1\ s_1\ s_2, (E, \mathit{val}\ v) \Downarrow o}\ v = 0\ \textsc{If1False}(s_1, s_2)
\qquad
\frac{}{\mathit{while}_1\ e\ s, \mathit{ret}\ E \Downarrow o}\ \textsc{While}(e, s)
$$

$$
\frac{e, E \Downarrow o \qquad \mathit{while}_2\ e\ s, (E, o) \Downarrow o'}{\mathit{while}_1\ e\ s, \mathit{ret}\ E \Downarrow o'}\ \textsc{While1}(e, s)
\qquad
\frac{s, E \Downarrow o \qquad \mathit{while}_1\ e\ s, o \Downarrow o'}{\mathit{while}_2\ e\ s, (E, \mathit{val}\ v) \Downarrow o'}\ v \neq 0\ \textsc{While2True}(e, s)
\qquad
\frac{}{\mathit{while}_2\ e\ s, (E, \mathit{val}\ v) \Downarrow \mathit{ret}\ E}\ v = 0\ \textsc{While2False}(e, s)
$$

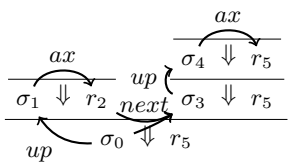
Figure 3: Concrete Semantics

function $\mathfrak{u}_1 : (i \in \mathcal{I}) \to (\mathit{type}\,(i) = R_1) \to \mathit{term}$ returning this term. Note how this function is restricted on format 1 rules.

Similarly, format 2 rules have two inductive hypotheses, hence need to evaluate the semantics of two terms, respectively given by functions $\mathfrak{u}_2 : (i \in \mathcal{I}) \to (\mathit{type}\,(i) = R_2) \to \mathit{term}$ and $\mathfrak{n}_2 : (i \in \mathcal{I}) \to (\mathit{type}\,(i) = R_2) \to \mathit{term}$.[1]

The functions *type*, $\mathfrak{l}$, $\mathfrak{u}_1$, $\mathfrak{u}_2$, and $\mathfrak{n}_2$ describe the *structure* of a rule, but not how it computes with the semantic contexts. This computation is done in the transfer functions that are contained in the constructions of type Rule_i.

Semantic Aspects of Rules We now define how semantic contexts and results are manipulated according to the semantics. To this end, we define transfer functions, which depend on the format of rule we are defining. They can be summed up in the following informal scheme, detailed below.



Depending on the format of a rule i, Rule_i will have different transfer functions. In every case, it will take a semantic context σ of type $\mathit{st}\,(\mathfrak{l}_i)$ and a proof of $\mathit{cond}_i\,(\sigma)$. Depending on the type $\mathit{type}\,(i)$ of the rule, it then proceeds as follows to obtain the semantics of t in context σ.

[1] $\mathfrak{u}_k$ stands for "up" and $\mathfrak{n}_2$ stands for "next".

- Axioms directly return a value of type $res\,(\mathfrak{l}_i)$, and are thus described by a function of type

$$ax : (\sigma \in st\,(\mathfrak{l}_i)) \to cond_i\,(\sigma) \to res\,(\mathfrak{l}_i)$$

  Let $r \in res\,(\mathfrak{l}_{\text{ID}})$ be the result of an axiom ID for input $t \in term$, $\sigma \in st\,(\mathfrak{l}_{\text{ID}})$, and a proof π of $cond_{\text{ID}}\,(\sigma)$. We write such a rule as follows.

$$\text{ID}\ \frac{ax\,(\sigma, \pi) \rightsquigarrow r}{\mathfrak{l}_{\text{ID}}, \sigma \Downarrow r}\ cond_{\text{ID}}\,(\sigma)$$

- Rules with one inductive hypothesis are of the following form.

$$\text{ID}\ \frac{\mathfrak{u}_{1,\text{ID}}, up\,(\sigma) \Downarrow r}{\mathfrak{l}_{\text{ID}}, \sigma \Downarrow r}\ cond_{\text{ID}}\,(\sigma)$$

  Such a rule specifies a new term $\mathfrak{u}_{1,\text{ID}}$ as described above, as well as a new semantic context $up\,(\sigma)$ of type $st\,(\mathfrak{u}_{1,\text{ID}})$ and returns the result of evaluating $\mathfrak{u}_{1,\text{ID}}$ in this context as the semantics of $\mathfrak{l}_{\text{ID}}$. For such rules, the format thus implicitly requires that $res\,(\mathfrak{l}_{\text{ID}}) = res\,(\mathfrak{u}_{1,\text{ID}})$. Hence, the essence of a format 1 rule is the function up that maps σ to $up\,(\sigma)$. Together with the $cond$ predicate and the $\mathfrak{l}$ and $\mathfrak{u}_1$ functions, this function up is the only information needed for completely defining such rules. A format 1 rule identified by i is therefore characterized by a function of type

$$up : (\sigma \in st\,(\mathfrak{l}_i)) \to cond_i\,(\sigma) \to st\,(\mathfrak{u}_{1,i})$$

- Rules with two inductive hypotheses are of the following form.

$$\text{ID}\ \frac{\mathfrak{u}_{2,\text{ID}}, up\,(\sigma) \Downarrow r \qquad \mathfrak{n}_{2,\text{ID}}, next\,(\sigma, r) \Downarrow r'}{\mathfrak{l}_{\text{ID}}, \sigma \Downarrow r'}\ cond_{\text{ID}}\,(\sigma)$$

  Such rules first do an inductive call as in the previous case. The result r of this call is then used to build the semantic context for the second inductive call. As the final result is propagated as-is, the required information is: a first semantic context $up\,(\sigma) \in st\,(\mathfrak{u}_{2,\text{ID}})$, and a function $next\,(\sigma, \cdot)$ transforming the result of the first inductive call into a semantic context of type $st\,(\mathfrak{n}_{2,\text{ID}})$.

  A format 2 rule i thus consists of two transfer functions:

$$\begin{array}{rcl} up & : & (\sigma \in st\,(\mathfrak{l}_i)) \to cond_i\,(\sigma) \to st\,(\mathfrak{u}_{2,i}) \\ next & : & (\sigma \in st\,(\mathfrak{l}_i)) \to cond_i\,(\sigma) \to res\,(\mathfrak{u}_{2,i}) \to st\,(\mathfrak{n}_{2,i}) \end{array}$$

  Analogous to rules of format 1, we impose the result type of $\mathfrak{l}_i$ to be that of $\mathfrak{n}_{2,i}$, i.e., $res\,(\mathfrak{l}_i) = res\,(\mathfrak{n}_{2,i})$.

To sum up, we define the set of rules as the set $Rule_i$ where each element is one is one of the following.

$$\begin{array}{c} Ax_i(ax : (\sigma \in st\,(\mathfrak{l}_i)) \to cond_i\,(\sigma) \to res\,(\mathfrak{l}_i)) \\ R_{1,i}(up : (\sigma \in st\,(\mathfrak{l}_i)) \to cond_i\,(\sigma) \to st\,(\mathfrak{u}_{1,i})) \\ R_{2,i}\begin{pmatrix} up : (\sigma \in st\,(\mathfrak{l}_i)) \to cond_i\,(\sigma) \to st\,(\mathfrak{u}_{2,i}) \\ next : (\sigma \in st\,(\mathfrak{l}_i)) \to cond_i\,(\sigma) \to res\,(\mathfrak{u}_{2,i}) \to st\,(\mathfrak{n}_{2,i}) \end{pmatrix} \end{array}$$

4. Mechanized PBS Semantics

We now describe how we implemented this formalization in Coq. The structural aspects directly follow the approach given in the previous section. Assuming a set of terms, we first define the structural part of rules, corresponding to the $\mathfrak{u}_1$, $\mathfrak{u}_2$, and $\mathfrak{n}_2$ functions. They carry the terms that need to be reduced in inductive hypotheses.

```
Inductive Rule_struct term :=
  | Rule_struct_Ax : Rule_struct term
  | Rule_struct_R1 : term →  Rule_struct term
  | Rule_struct_R2 : term → term → Rule_struct term.
```

$$\text{ID}\ \frac{}{\mathfrak{l}_{\text{ID}}, \sigma \Downarrow ax\,(\sigma)}\ cond_{\text{ID}}\,(\sigma) \qquad \text{ID}\ \frac{\mathfrak{u}_{1,\text{ID}}, up\,(\sigma) \Downarrow r}{\mathfrak{l}_{\text{ID}}, \sigma \Downarrow r}\ cond_{\text{ID}}\,(\sigma)$$

$$\text{ID}\ \frac{\mathfrak{u}_{2,\text{ID}}, up\,(\sigma) \Downarrow r \qquad \mathfrak{n}_{2,\text{ID}}, next\,(\sigma, r) \Downarrow r'}{\mathfrak{l}_{\text{ID}}, \sigma \Downarrow r'}\ cond_{\text{ID}}\,(\sigma)$$

Figure 4: Rule Formats

Rule identifiers (`name` in the Coq files) are associated with the term reduced by the rule (function $\mathfrak{l}$, called `left` in Coq) and to structural terms. They are packaged together as follows.

```
Record structure := {
    term : Type;
    name : Type;

    left : name → term;
    rule_struct : name → Rule_struct term }.
```

A semantics, parameterized by such a structure, is then a type of semantic contexts, a type of results, a predicate to determine whether a rule may be applied, and transfer functions for the rules.

```
Record semantics := make_semantics {
    st : Type;
    res : Type;

    cond : name → st → Prop;
    rule : name → Rule st res }.
```

We now detail the components of this semantics, highlighting the differences with Section 3.

Although a definition based on dependent types is very elegant, its implementation in Coq proved to be quite challenging. The typical difficulty we had appeared in format 1 and 2 rules where results are passed without modification from a premise to the conclusion, but whose types change from $res\,(\mathfrak{l}_{\text{ID}})$ to $res\,(\mathfrak{u}_{k,\text{ID}})$. In such contexts these two types happen to be equal because of the implicit hypotheses we enforced in the previous section. However, as usually with dependent types, a lot of predicates require these terms to have a specific (syntactical) type. Rewriting "equal" terms (i.e., equal under heterogeneous, or "John Major's", equality [14]) becomes really painful when there exist such syntactic constraints.

We thus switched to a simpler approach. First, the type for semantic contexts (respectively results) is no longer specialized by (or dependent on) the term under consideration: it is the union of every possible semantic context (respectively of every result). This can be seen in the `st` and `res` fields above that are simple types.

The rules are then adapted to this setting. They are very similar to the version of Section 3 as can be seen in Figure 4. The *Rule* type uses the following transfer functions.

```
Inductive Rule st res :=
  | Rule_Ax : (st → option res) → Rule st res
  | Rule_R1 : (st → option st) → Rule st res
  | Rule_R2 : (st → option st) →
              (st → res → option st) → Rule st res.
```

The function $ax : st \to option\ res$ for axioms returns `None` if the rule does not apply, either because the semantic context does not have the correct shape, or if the condition to apply the rule is

not satisfied. This is in contrast to the definition of Section 3, where the option was not required: the type $(\sigma \in st\,(\mathfrak{l}_i)) \to cond_i\,(\sigma) \to res\,(\mathfrak{l}_i)$ did guarantee that the semantic context was compatible with the term and that the rule applied.

The transfer function of a format 1 rule is of the form $up : st \to option\ st$, constructing a new semantic context if the context given as argument has the correct shape.

The transfer functions of a format 2 rule are of the form $up : st \to option\ st$ and $next : st \to res \to option\ st$.

It may seem that we compute the same thing twice: $cond_i\,(\sigma)$ states that a given rule i applies to σ, while ax (or the corresponding transfer function) should also return `None` if the rule cannot be applied. We actually relax this second requirement to allow for simpler definition: transfer functions may return a result even if they do not apply. For instance, the transfer function of $\textsc{VarUndef}\,(x)$ always returns *Err*, but it may only be applied if the variable is not in the environment. This separation between side-conditions and transfer functions is a separation between the control flow and the actual computation. In the Coq development, the first one is implemented using predicates, and the second using computable functions.

We now describe how to assemble rules to build a concrete evaluation relation $\Downarrow \in \mathcal{P}\,(term \times st \times res)$. We define the concrete semantics as the least fixed point of a function $\mathcal{F}$ which we now detail.

$$\mathcal{F} : \mathcal{P}\,(term \times st \times res) \to \mathcal{P}\,(term \times st \times res)$$

Given an existing evaluation relation $\Downarrow_0 \in \mathcal{P}\,(term \times st \times res)$, the application function $apply_i\,(\Downarrow_0) \;:\; \mathcal{P}\,(term \times st \times res)$ for $rule\,(i)$ is as follows.

$$\begin{array}{l} apply_i\,(\Downarrow_0) := \\ \left|\begin{array}{l} match\ rule\,(i)\ with \\ \mid\quad Ax\,(ax) \qquad\quad \Rightarrow \{(\mathfrak{l}_i, \sigma, r) \mid ax\,(\sigma) = \texttt{Some}(r)\} \\ \mid\quad R_1\,(up) \qquad\quad \Rightarrow \left\{ (\mathfrak{l}_i, \sigma, r) \;\middle|\; \begin{array}{c} up\,(\sigma) = \texttt{Some}(\sigma') \\ \wedge\, \mathfrak{u}_{1,i}, \sigma' \Downarrow_0 r \end{array} \right\} \\ \mid\quad R_2\,(up, next) \Rightarrow \left\{ (\mathfrak{l}_i, \sigma, r) \;\middle|\; \begin{array}{c} up\,(\sigma) = \texttt{Some}(\sigma') \\ \wedge\, \mathfrak{u}_{2,i}, \sigma' \Downarrow_0 r_1 \\ \wedge\, next\,(\sigma, r_1) = \texttt{Some}(\sigma'') \\ \wedge\, \mathfrak{n}_{2,i}, \sigma'' \Downarrow_0 \texttt{Some}(r) \end{array} \right\} \end{array}\right. \end{array}$$

This relation $apply_i\,(\Downarrow_0)$ accepts a tuple (t, σ, r) if it can be computed by making one semantic step using $rule\,(i)$, calling back $\Downarrow_0$ for every recursive call.

The final evaluation relation is then computed step by step using the function $\mathcal{F}$, computing from an evaluation relation $\Downarrow_0$ the following new relation $\mathcal{F}\,(\Downarrow_0)$:

$$\mathcal{F}\,(\Downarrow_0) = \{(t, \sigma, r) \mid \exists i, cond_i\,(\sigma) \wedge (t, \sigma, r) \in apply_i\,(\Downarrow_0)\}$$

Intuitively, each application of $\mathcal{F}$ extends the relation $\Downarrow_0$ by computing the results of derivations with an extra step.

We can equip the set of evaluation relations $\mathcal{P}\,(term \times st \times res)$ with the usual inclusion lattice structure. In this lattice, the functions $apply_i$ and $\mathcal{F}$ are monotonic. We can thus define the fixed points of $\mathcal{F}$ in this lattice. We consider as our semantics the least fixed point $\Downarrow_{lfp}$, which corresponds to an inductive interpretation of the rules: only finite behaviors are taken into account, and no semantics is given to non-terminating programs. We note it $\Downarrow$.

The implementation in Coq shown in Figure 5 directly builds the fixed point as an inductive definition.

```
Inductive eval : term → st → res → Type :=
  | eval_cons : ∀ t sigma r n,
      t = left n →
      cond n sigma →
      apply n sigma r →
      eval t sigma r
with apply : name → st → res → Type :=
  | apply_Ax : ∀ n ax sigma r,
      rule_struct n = Rule_struct_Ax _ →
      rule n = Rule_Ax ax →
      ax sigma = Some r →
      apply n sigma r
  | apply_R1 : ∀ n t up sigma sigma' r,
      rule_struct n = Rule_struct_R1 t →
      rule n = Rule_R1 _ up →
      up sigma = Some sigma' →
      eval t sigma' r →
      apply n sigma r
  | apply_R2 : ∀ n t1 t2 up next
                 sigma sigma1 sigma2 r r',
      rule_struct n = Rule_struct_R2 t1 t2 →
      rule n = Rule_R2 up next →
      up sigma = Some sigma1 →
      eval t1 sigma1 r →
      next sigma r = Some sigma2 →
      eval t2 sigma2 r' →
      apply n sigma r'.
```

Figure 5: Coq definition of the concrete semantics $\Downarrow$

5. Mechanized PBS Abstract Semantics

The purpose of mechanizing the PBS semantics is to facilitate the correctness proof of static analyzers with respect to a concrete semantics. We thus provide a mechanized way to define an abstract semantics and prove it correct with respect to the concrete one. Its usage to prove static analyzers is described in Section 6.

As stated in the Introduction, the starting point for our development is the abstract interpretation of big-step semantics, laid out by Schmidt [20]. In this section, we describe how an adapted version of Schmidt's framework can be implemented using the Coq proof assistant. There are several steps in such a formalization:

- define the Galois connection that relates concrete and abstract domains of semantic contexts and results;
- based on the Galois connection between concrete and abstract domains, prove the local correctness: the side-conditions and transfer functions of each concrete rule are correctly abstracted by their abstract counterpart;
- given the local correctness, prove the global correctness: the abstract semantics $\Downarrow^\sharp$ is a correct approximation of the concrete semantics $\Downarrow$, i.e., the least fixed point of the $\mathcal{F}$ operator.

The Galois connections relate the concrete and abstract semantic triples (t, σ, r) and $\left(t, \sigma^\sharp, r^\sharp\right)$ by a concretisation function γ. They let us state and prove the following property relating the concrete and the abstract semantics. Let $t \in term$, $\sigma \in st$, $\sigma^\sharp \in st^\sharp$, $r \in res$ and $r^\sharp \in res^\sharp$,

$$\text{if} \left\{ \begin{array}{l} \sigma \in \gamma\left(\sigma^\sharp\right) \\ t, \sigma \Downarrow r \\ t, \sigma^\sharp \Downarrow^\sharp r^\sharp \end{array} \right. \quad \text{then } r \in \gamma\left(r^\sharp\right).$$

We illustrate the development through the implementation of a sign analysis for our simple imperative language. However, we emphasize that the approach is generic: once an abstract domain is

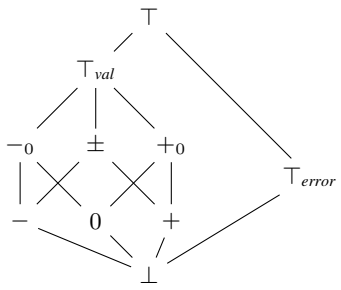


Figure 6: The Hasse diagram of the $valerr^\sharp$ lattice

given, and abstract transfer functions are shown to be correct, then the full abstract semantics is correct by construction.

5.1 Abstract Domains

The starting point for the abstract interpretation of big-step semantics is a collection of abstract domains, related to the concrete semantic domains by a Galois connection, or just by a concretisation function γ. The formalization of Galois connections in proof assistants has been studied in previous work by several authors (e.g., [5, 18]), and we have relied on existing libraries of constructors for building abstract domains.

For our example analysis, we have abstracted the base domain of integers by the abstract domain of signs. The singleton domain of errors is abstracted to a two-point domain where $\bot_{error}$ means absence of errors and $\top_{error}$ means the possible presence of an error. The result of an expression is either a value or an error, modeled by the sum domain out_e. We abstract this by a product domain with elements of the form $\left(v^\sharp, e^\sharp\right)$, where $v^\sharp$ is a property of the result (if any is produced) and $e^\sharp$ indicates the possibility of an error. A result that is known to be an error is thus abstracted by $(\bot_{val}, \top_{error}) \in out_e^\sharp$. To summarize, the analysis uses the following abstract domains:

- $val^\sharp = sign = \{\bot_{val}, -, 0, +, -_0, \pm, +_0, \top_{val}\}$;
- $error^\sharp = \{\bot_{error}, \top_{error}\}$, named `aErr` in the Coq files;
- $valerr^\sharp = \left(val^\sharp \otimes error^\sharp\right)^\top$;
- $env^\sharp = var \to valerr^\sharp$, `aEnv` in Coq;
- $out_e^\sharp = val^\sharp \times error^\sharp$, `aOute` in Coq;
- $out_s^\sharp = env^\sharp \times error^\sharp$, `aOuts` in Coq.

As the absence of variable in a concrete environment leads to a different rule than a defined variable whose value we know nothing about, we have to track the absence of variable in abstract environments. We use the $valerr^\sharp$ lattice to achieve this. Its lattice structure is pictured in Figure 6. Notice that $\bot_{val}$ and $\bot_{error}$ are coalesced in this domain, i.e., we construct $valerr^\sharp$ as the smash product of $val^\sharp$ and $error^\sharp$.

In the Coq formalization, the discrimination between the possible output domains is implemented with a coalescing sum of partial orders that identifies the bottom elements of the two domains

$$\left(out_e^\sharp + out_s^\sharp\right)_\bot^\top$$

where the new top element indicates a type error due to confusion of expressions and statements. The abstract result type is defined as follows in Coq.

```
Inductive ares : Type :=
  | ares_expr : aOute → ares
  | ares_prog : aOuts → ares
  | ares_top : ares
  | ares_bot : ares.
```

5.2 Rule Abstraction

The abstract interpretation of the big-step semantics produces a new set of inference rules where the semantic domains are replaced by their abstract counterparts. Thus, rules no longer operate over values but over properties, represented by abstract values. For instance, the rule for addition Add$_2$, which applies when both sub-expressions of an addition have been evaluated to an integer value,

$$\textsc{Add}_2\ \frac{add\,(v_1, v_2) \rightsquigarrow v}{+_2, (v_1, val\,v_2) \Downarrow val\,v}$$

is replaced by a rule using an abstract operator $add^\sharp$

$$\textsc{Add}_2^\sharp\ \frac{add^\sharp\,(v_1, v_2) \rightsquigarrow v}{+_2, (v_1, val^\sharp\,v_2) \Downarrow^\sharp v}$$

where the concrete addition of integers has been replaced with its abstraction over the abstract domain of signs.

As explained by Schmidt [20, Section 8], the abstract interpretation of a big-step semantics must be built such that all concrete derivations are covered by an abstract counterpart. Here, "covered" is formalized by extending the correctness relation on base domains and environments to derivation trees. A concrete and an abstract derivations Δ and $\Delta^\sharp$ are related if the conclusion statement of Δ is in the correctness relation with the conclusion of $\Delta^\sharp$, and, furthermore, for each sub-derivation of Δ, there exists a corresponding abstract sub-derivation of $\Delta^\sharp$ which covers it.

There are several ways in which coverage can be ensured. One way is to add a number of *ad hoc* rules. For example, it is common for inference-based analyses to include a rule such as

$$\textsc{If-abs}\ \frac{\Gamma \vdash e_1 : \phi_1 \quad \Gamma \vdash e_2 : \phi_2}{\Gamma \vdash \text{if } b \text{ then } e_1 \text{ else } e_2 : \phi_1 \sqcup \phi_2}$$

that covers execution of both branches of an if.

Instead of adding extra rules, we pursue an approach where we obtain coverage in a systematic fashion, by

1. abstracting the conditions and transfer functions of the individual rules according to a common correctness criterion;
2. defining the way that a set of abstract rules are applied when analyzing a given term. This is described in Section 5.3 below.

We use exactly the same framework (as shown in the Coq development) for concrete and abstract rules. The only difference is how we assemble abstract rules to build an abstract semantics $\Downarrow^\sharp$.

Recall that a rule comprises a side-condition that determines if it applies and one or more transfer functions to map the input state to a result. The abstract side-condition $cond^\sharp$ must satisfy the following correctness criterion.

$$\forall \sigma, \sigma^\sharp.\, \sigma \in \gamma(\sigma^\sharp) \Rightarrow cond\,(\sigma) \Rightarrow cond^\sharp\left(\sigma^\sharp\right).$$

Intuitively, this means that whenever there is a state in the concretisation of an abstract state $\sigma^\sharp$ that would trigger a concrete rule, then the corresponding abstract rule is also triggered by $\sigma^\sharp$. Figure 7 is a snippet from the Coq formalization showing the conditions of the various rules for *while*. They correspond in a one-to-one fashion to the rules of the concrete semantics defining the *cond* predicate.

```
Definition acond n asigma : Prop :=
  match n, asigma with
  ...
  | name_while e s, ast_prog aE ⇒
      True
  | name_while_1 e s, ast_while_1 ar ⇒
      ares_prog (⊥) ⊑ ar
  | name_abort_while_1 e s, ast_while_1 ar ⇒
      ares_prog (⊥,⊤) ⊑ ar
  | name_while_2_true e s, ast_while_2 aE o ⇒
      ares_expr (Sign.pos,⊥) ⊑ o ∨
      ares_expr (Sign.neg,⊥) ⊑ o
  | name_while_2_false e s, ast_while_2 aE o ⇒
      ares_expr (Sign.zero,⊥) ⊑ o
  | name_abort_while_2 e s, ast_while_2 aE ar ⇒
      ares_expr (⊥,⊤) ⊑ ar
  ...
```

Figure 7: Snippet of the $cond^\sharp$ predicate

```
Definition arule n : Rule sign_ast sign_ares :=
  match n with
  ...
  | name_while e s ⇒
    let up :=
      if_ast_prog (fun E ⇒
        Some (sign_ast_while_1
          (sign_ares_stat (E, ⊥)))) in
    Rule_R1 _ up
  | name_while_1 e s ⇒
    let up :=
      if_ast_while_1 (fun E err ⇒
        Some (sign_ast_prog E)) in
    let next asigma ar :=
      if_ast_while_1 (fun E err ⇒
        Some (sign_ast_while_2 E ar)) asigma in
    Rule_R2 up next
  ...
```

Figure 8: Snippet of the *rule* function

Similar correctness criteria apply to the transfer function defining the rules. For example, axioms, that are defined by a function ax from input states to results, have an abstraction $ax^\sharp$ that must satisfy

$$\forall \sigma, \sigma^\sharp. \sigma \in \gamma\left(\sigma^\sharp\right) \Rightarrow ax\left(\sigma\right) \in \gamma\left(ax^\sharp\left(\sigma^\sharp\right)\right).$$

These criteria are defined as a relation $\sim$ between rules (called `propagates` in the Coq files), made precise below. We assume it has been shown to hold for every pair of concrete and abstract rules sharing the same identifier.

The Coq snippet of Figure 8 shows the encoding of the abstract rules $\textsc{While}\left(e,s\right)$ and $\textsc{While1}\left(e,s\right)$. The former is a format 1 and thus only need an *up* function to be defined. The facts that it applies on $\mathfrak{t}_{\textsc{While}(e,s)} = \mathit{while}\, e\, s$ and that its intermediate term is $\mathfrak{u}_{1,\textsc{While}(e,s)} = \mathit{while}_1\, e\, s$ are already expressed by the structure part and are not shown here.

This function *up* should be called on a context $\sigma^\sharp$ that satisfies $\mathit{cond}^\sharp_{\textsc{While}(e,s)}\left(\sigma^\sharp\right)$, that is, on an environment. There is however no typing rule that enforces this (as we do not use dependent types in this formalization, as explained in Section 4) and we thus have to check this, returning `None` otherwise. We use the following monad to extract the relevant environment.

```
if_ast_prog :
  (aEnv → option sign_ares)
    → sign_ast → option sign_ares
```

We then compute the semantic context corresponding to $\mathfrak{u}_1 = \mathit{while}_1\, e\, s$. In this case, it is `sign_ast_while_1 (E, ⊥)`, where `E` is the extracted environment, as the corresponding rule does not introduce errors while propagating the environment.

The abstract rule $\textsc{While1}\left(e,s\right)$ is a format 2 rule and thus needs two functions, *up* and *next*, to be similarly defined. As the expected kind of the semantic context is in this case the one of $\mathit{while}_1\, e\, s$, we use a different monad:

```
if_ast_while_1 :
  (aEnv → aErr → option sign_ares)
    → sign_ast → option sign_ares
```

These definitions are so similar to the concrete definitions that they can be built directly from a concrete definition. This similarity simplifies definitions and proofs considerably.

Finally, the relation $\sim$ that relates concrete and abstract rules can be defined as follows.

- A concrete and an abstract axioms $ax : st \to res$ and $ax^\sharp : st^\sharp \to res^\sharp$ are related iff for all σ and $\sigma^\sharp$ on which both functions ax and $ax^\sharp$ are defined, and such that $\sigma \in \gamma\left(\sigma^\sharp\right)$, then $ax\left(\sigma\right) \in \gamma\left(ax^\sharp\left(\sigma^\sharp\right)\right)$.
- A concrete and an abstract format 1 rules $up : st \to st$ and $up^\sharp : st^\sharp \to st^\sharp$ are related iff for all σ and $\sigma^\sharp$ on which both functions up and $up^\sharp$ are defined, and such that $\sigma \in \gamma\left(\sigma^\sharp\right)$, then $up\left(\sigma\right) \in \gamma\left(up^\sharp\left(\sigma\right)\right)$.
- For format 2 rules, we impose the same condition on the up and $up^\sharp$ transfer functions than above, and we add the additional condition over the transfer functions $next : st \to res \to st$ and $next^\sharp : st^\sharp \to res^\sharp \to st^\sharp$: for all σ, $\sigma^\sharp$, r and $r^\sharp$ on which both functions $next$ and $next^\sharp$ are defined, and such that $\sigma \in \gamma\left(\sigma^\sharp\right)$ and $r \in \gamma\left(r^\sharp\right)$, then $next\left(\sigma, r\right) \in \gamma\left(next^\sharp\left(\sigma^\sharp, r^\sharp\right)\right)$.

5.3 Inference Trees

Concrete and abstract semantic rules have been defined to have similar structure. However, the semantics given to a set of abstract rules differs from the concrete semantics defined in Section 4. This difference manifests itself in the way rules are assembled.

First, the function $\mathit{apply}_i^\sharp$ for applying an abstract rule with identifier i extends the apply_i function by allowing to weaken semantic contexts and results. Indeed, the purpose of the abstract semantics is to capture every correct abstract analyses, including the ones that lose precision. It is thus possible to choose a less precise semantic context σ_0 before referring to apply_i, and to then return a less precise result r afterwards.

$$\mathit{apply}_i^\sharp\left(\Downarrow_0^\sharp\right) = \left\{ (t,\sigma,r) \;\middle|\; \begin{array}{l} \exists \sigma_0, \exists r_0, \\ \quad \sigma \sqsubseteq^\sharp \sigma_0 \wedge r_0 \sqsubseteq^\sharp r \wedge \\ \quad (t,\sigma_0,r_0) \in \mathit{apply}_i\left(\Downarrow_0^\sharp\right) \end{array} \right\}$$

Second, we define a function $\mathcal{F}^\sharp$ that infers new derivations from a set of already established derivations, by applying the abstract inference rules. The definition of the function $\mathcal{F}^\sharp$ differs in one important aspect from its concrete counterpart: in order to obtain coverage of concrete rules, $\mathcal{F}^\sharp$ must apply *all* the rules that are

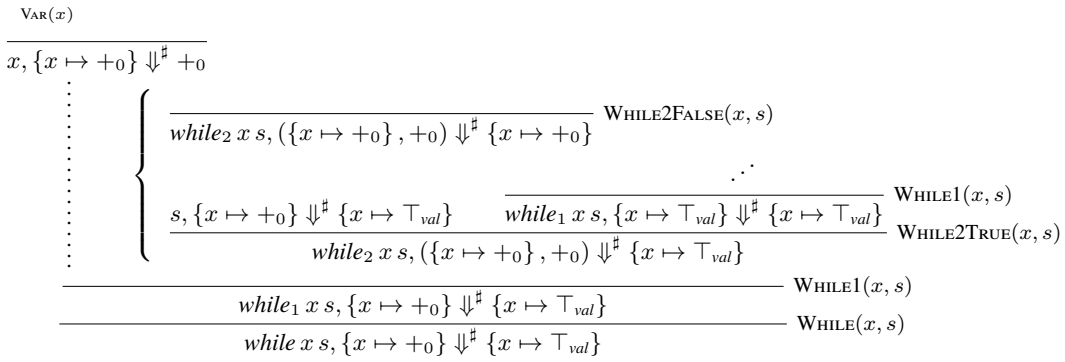


Figure 9: An infinite abstract derivation tree corresponding to a finite concrete derivation tree, where $s \triangleq (x := +\ x\ (-1))$

enabled for a term in the given abstract state.

$$\mathcal{F}^\sharp\left(\Downarrow_0^\sharp\right) = \left\{ (t,\sigma,r) \;\middle|\; \begin{array}{r} \forall i.\, t = \mathfrak{l}_i \Rightarrow \mathit{cond}_i\,(\sigma) \Rightarrow \\ (t,\sigma,r) \in \mathit{apply}_i^\sharp\left(\Downarrow_0^\sharp\right) \end{array} \right\}$$

In other words, the function extends the relation $\Downarrow_0^\sharp$ by adding those triples (t,σ,r) such that the result r is valid for *all* rules. By defining $\mathcal{F}^\sharp$ in this way, we avoid having to add rules such as the IF-ABS rule from above: a correct result is one that includes the computation from both branches.

Let us consider a simple example to give some intuition. The program *if* $x\ (r := 0)\ (r := x)$ always sets r to zero if x is defined. Let us analyze it in an environment $E_1^\sharp \in \mathit{env}^\sharp$ where x is $+$, and in an environment $E_2^\sharp \in \mathit{env}^\sharp$ where x is $\top_{\mathit{val}}$, i.e., x is defined but we know nothing about its value. In either case, it expands after one step to $\mathit{if}_1\ (r := 0)\ (r := x)$, and carries an information about the computed expression x that is either $+$ or $\top_{\mathit{val}}$ (or any weaker result, but we only consider a precise derivation in this example). In the first case we know that this expression is non zero, and only the rule IF1TRUE $(r := 0, r := x)$ applies: we evaluate $r := 0$ and can conclude that r is zero. However in the second case, we don't know which branch will be executed and thus additionally consider the rule IF1FALSE $(r := 0, r := x)$. This branch executes $r := x$ and sets r to $\top_{\mathit{val}}$. This example illustrates a shortcoming of our approach: even though we know the value tested has to be 0 in the "false" branch, there is no information about how that value was computed (evaluating x in this example). The non-local information that allows to deduce that x is bound to 0 in the environment is currently not available to our framework.

The function $\mathcal{F}^\sharp$ is a monotone function on the lattice

$$\mathcal{P}\left(\mathit{term} \times \mathit{st}^\sharp \times \mathit{res}^\sharp\right).$$

The least fixed point of $\mathcal{F}^\sharp$ (with respect to the inclusion $\subseteq$ order) corresponds to all triples that can be inferred using finite derivation trees. These triples are valid properties of the program, but the restriction to finite derivations means that certain properties cannot be inferred.

Consider the program *while* $x\ (x := +\ x\ (-1))$ evaluated on a context where x is positive. Its concrete derivation clearly terminates, but there is no finite derivation in the sign abstraction semantics to witness it. Indeed, initially x is bound to $+_0$. After the first iteration, it is bound to $\top_{\mathit{val}}$, then its value becomes stable. Every subsequent iteration thus has to consider the case where x is not 0 and to compute an additional iteration. Hence, there is no finite abstract derivation: the abstract domain is not precise enough.

Intuitively, since the concrete derivation tree has to be "included" into the abstract derivation tree, and since there is no bound on the number of execution steps in the concrete derivation (which depends on the initial value of x, the loop being unfolded that many times), any abstract derivation has to be infinite.

Figure 9 depicts the abstract derivation tree built by recursively applying $\mathcal{F}^\sharp$, writing s for $(x := +\ x\ (-1))$. Both rules WHILE2TRUE (x,s) and WHILE2FALSE (x,s) are executed and their results $\{x \mapsto +_0\}$ and $\{x \mapsto \top_{\mathit{val}}\}$ are merged (in this case, the second merged element is greater than the first one). This follows the definition of $\mathcal{F}^\sharp$, that applies *every* rule that can be applied.

We thus need to allow infinite abstract derivations. To this end, the abstract evaluation relation, written $\Downarrow^\sharp$, is obtained as the greatest fixed point of $\mathcal{F}^\sharp$. The correctness of this extension, since $\mathit{lfp}\left(\mathcal{F}^\sharp\right) \subseteq \Downarrow^\sharp$, has been proven in Coq. More importantly, a coinductive approach allows analyzers to use more techniques, such as *invariants*, to infer their conclusions. The snippet of Figure 10 shows the definition of $\Downarrow^\sharp$ in Coq. Note the symmetry between this definition and the concrete definition of $\Downarrow$ in Figure 5.

5.4 Correctness of the Abstract Semantics

We have defined the local correctness as the conjunction of the correctness of the side-condition predicates *cond* and *cond*$^\sharp$ and the correctness of the transfer functions $\sim$, whose Coq versions follow:

```
Hypothesis acond_correct : ∀ n asigma sigma,
  gst asigma sigma → cond n sigma → acond n asigma.

Hypothesis Pr : ∀ n, propagates (arule n) (rule n).
```

We proved in Coq that under the local correctness, the concrete and abstract evaluation relations,

$$\begin{array}{rcl} \Downarrow & = & \mathit{lfp}\,(\mathcal{F}) \\ \Downarrow^\sharp & = & \mathit{gfp}\left(\mathcal{F}^\sharp\right) \end{array}$$

are related as follows.

Theorem 1 (Correctness)**.** *Let $t \in \mathit{term}$, $\sigma \in \mathit{st}$, $\sigma^\sharp \in \mathit{st}^\sharp$, $r \in \mathit{res}$ and $r^\sharp \in \mathit{res}^\sharp$.*

$$\mathit{If} \begin{cases} \sigma \in \gamma\left(\sigma^\sharp\right) \\ t, \sigma \Downarrow r \\ t, \sigma^\sharp \Downarrow^\sharp r^\sharp \end{cases} \quad \mathit{then}\ r \in \gamma\left(r^\sharp\right).$$

Here follows the Coq version of this theorem. It has been proven in a completely parameterized way with respect to the concrete and abstract domains, as well as the rules.

```
CoInductive aeval : term → ast → ares → Prop :=
  | aeval_cons : ∀ t sigma r,
      (∀ n,
        t = left n →
        acond n sigma →
        aapply n sigma r) →
      aeval t sigma r
with aapply : name → ast → ares → Prop :=
  | aapply_cons : ∀ n sigma sigma' r r',
      sigma ⊑ sigma' →
      r' ⊑ r →
      aapply_step n sigma' r' →
      aapply n sigma r
with aapply_step : name → ast → ares → Prop :=
  | aapply_step_Ax : ∀ n ax sigma r,
      rule_struct n = Rule_struct_Ax _ →
      arule n = Rule_Ax ax →
      ax sigma = Some r →
      aapply_step n sigma r
  | aapply_step_R1 : ∀ n t up sigma sigma' r,
      rule_struct n = Rule_struct_R1 t →
      arule n = Rule_R1 _ up →
      up sigma = Some sigma' →
      aeval t sigma' r →
      aapply_step n sigma r
  | aapply_step_R2 : ∀ n t1 t2 up next
                       sigma sigma1 sigma2 r r',
      rule_struct n = Rule_struct_R2 t1 t2 →
      arule n = Rule_R2 up next →
      up sigma = Some sigma1 →
      aeval t1 sigma1 r →
      next sigma r = Some sigma2 →
      aeval t2 sigma2 r' →
      aapply_step n sigma r'.
```

Figure 10: Coq definition of the abstract semantics $\Downarrow^\sharp$

```
Theorem correctness : ∀ t asigma ar,
  aeval _ _ _ t asigma ar →
  ∀ sigma r,
    gst asigma sigma → eval _ t sigma r → gres ar r.
```

The predicates `aeval` and `eval` respectively represent $\Downarrow^\sharp$ and $\Downarrow$, while `gst` and `gres` are the concretisation functions for the semantic contexts and the results.

This allows us to easily prove the correctness of an abstract semantics with respect to a concrete semantics. We now show how this abstract semantics can be related to analyzers.

6. Building Certified Analyzers

The abstract semantics $\Downarrow^\sharp$ is the set of all triples provable using the set of abstract inference rules. From a program t and an abstract semantic context $\sigma^\sharp$, the smallest $r^\sharp$ such that $t, \sigma^\sharp \Downarrow^\sharp r^\sharp$ corresponds to the most precise analysis. It is, however, rarely computable. Designing a good certified analysis thus amounts to writing a program that returns a precise result that belongs to the abstract semantics.

To this end, we heavily rely on the co-inductive definition of $\Downarrow^\sharp$ to prove the correctness of static analyzers. In order to prove that a given analyzer $\mathcal{A} : \mathit{term} \to \mathit{st}^\sharp \to \mathit{res}^\sharp$ is correct with respect to $\Downarrow^\sharp$, (and thus with respect to the concrete semantics by Theorem 1), it is sufficient to prove that the set

$$\Downarrow^\sharp_{\mathcal{A}} = \left\{ \left(t, \sigma^\sharp, \mathcal{A}\left(t, \sigma^\sharp\right)\right)\right\}$$

is *coherent*, that is $\Downarrow^\sharp_{\mathcal{A}} \subseteq \mathcal{F}^\sharp\left(\Downarrow^\sharp_{\mathcal{A}}\right)$. Alternatively, on may define for every t and $\sigma^\sharp$ a set $R_{t,\sigma^\sharp} \in \mathcal{P}\left(\mathit{term} \times \mathit{st}^\sharp \times \mathit{res}^\sharp\right)$ such that

$$\left(t, \sigma^\sharp, \mathcal{A}\left(t, \sigma^\sharp\right)\right) \in R_{t,\sigma^\sharp} \text{ and } R_{t,\sigma^\sharp} \subseteq \mathcal{F}^\sharp\left(R_{t,\sigma^\sharp}\right).$$

This is exactly Park's principle [17] applied to $\mathcal{F}^\sharp$.

We instantiate this principle in Coq through the following alternative definition of $\Downarrow^\sharp$. The parameterized predicate `aeval_check` applies one step of the reduction: it exactly corresponds to $\mathcal{F}^\sharp$ and is defined in Coq similarly to `aeval` (Figure 10). More precisely, `aeval` is the co-inductive closure of `aeval_check`; we do not define it directly as such because Coq cannot detect productivity.

```
Inductive aeval_f : term → ast → ares → Prop :=
  | aeval_f_cons : ∀ (R : term → ast → ares → Prop)
                     t sigma r,
    (∀ t sigma r,
       R t sigma r →
       aeval_check R t sigma r) →
    R t sigma r →
    aeval_f t sigma r.
```

We then show the equivalence theorem that allows us to use Park's principle.

```
Theorem aevals_equiv : ∀ t sigma r,
  aeval t sigma r ↔ aeval_f t sigma r.
```

Using this principle, we have built and proved the correctness of several different analyzers, available in the Coq files accompanying this paper [4]. Most of these analyzers are generic and can be reused as-is[2] with any abstract semantics built using our framework. We next describe two such analyzers.

- Admitting a $\top$ rule as a trivial analyzer that always return $\top$ independently of the given t and $\sigma^\sharp$.
- Building a certified program verifier that can check loop invariants from a (non-verified) oracle and use these to make abstract interpretations of programs.

Admitting a $\top$ rule This trivial analyzer shows how to add derived rules to the abstract semantics. There is indeed no axiom rule that directly returns the $\top$ result for any term and context. Admitting this rule (which is often taken for granted) amounts exactly to prove that the corresponding trivial analyzer is correct. We thus define the set $\Downarrow^\sharp_\top = \left\{\left(t, \sigma^\sharp, \top\right)\right\}$ and prove it coherent. We have to prove that every triple $\left(t, \sigma^\sharp, \top\right)$ is also part of $\mathcal{F}^\sharp\left(\Downarrow^\sharp_\top\right)$, that is that for every rule i that applies, i.e., $\mathit{cond}^\sharp_i\left(\sigma^\sharp\right)$, then $\left(t, \sigma^\sharp, \top\right) \in \mathit{apply}^\sharp_i\left(\Downarrow^\sharp_\top\right)$. But as $\top$ is greater than any other result, we just have to prove that there exists at least one result $r^\sharp$ such that $\left(t, \sigma^\sharp, r^\sharp\right) \in \mathit{apply}^\sharp_i\left(\Downarrow^\sharp_\top\right)$. This last property is implied by *semantic fullness*, which we require for every semantics: transfer functions are defined where $\mathit{cond}^\sharp$ holds.

Building a certified program verifier To allow the usage of external heuristics to provide potential program properties, and thus relax proof obligations, we have also proved a verifier: it takes an oracle, i.e., a set of triples $O \in \mathcal{P}\left(\mathit{term} \times \mathit{st}^\sharp \times \mathit{res}^\sharp\right)$, and accepts or rejects it. An acceptance implies the correctness of every triple

[2] A function computing the list of rules which apply to a given t and $\sigma^\sharp$ has to be defined. Some of these generic analyzers also need a function detecting "looping" terms (in this example terms of the form $\mathit{while}_1\ s_1\ s_2$).

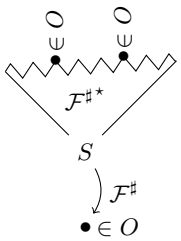


Figure 11: An Illustration of the Action of the Verifier

of O. For every triple $o = \left(t, \sigma^\sharp, r^\sharp\right) \in O$, the verifier checks that it can be deduced from finite derivations starting from axioms and elements of O, i.e., $O \subseteq {\mathcal{F}^\sharp}^+ (O)$. In practice, the verifier computes hypotheses that imply o, a subset S of ${\mathcal{F}^\sharp}^{-1} (o)$ such that $o \in {\mathcal{F}^\sharp}^+ (S)$, and it iterates on S recursively until it reaches only elements of O and axioms, or until it gives up. This is illustrated in Figure 11. We prove the following.

Theorem 2 (Correctness of the verifier)**.** *If the verifier accepts O, then $O \subseteq {\mathcal{F}^\sharp}^+ (O)$ hence $O \subseteq \Downarrow^\sharp$.*

We extracts the verifier into OCaml. Note that it can be given any set, possibly incorrect. In that case it will simply give up. We have tested the verifier on some simple sets of potential program properties. These sets were constructed by following some abstract derivation trees up to a given number of loop unfoldings and ignoring deeper branches.

As an example, consider this program that computes 6×7 using a while loop.

$$a := 6; b := 7; r := 0; n := a; \mathit{while}\, n\ (r := +\ r\ b; n := +\ n\ (-1))$$

Using our analyzer on this program in the environment mapping every variable to $\top_{\mathit{error}}$ returns the following result.

$$\left(\{r \mapsto +, b \mapsto +, a \mapsto +, n \mapsto \top_{\mathit{val}}\}, \bot\right)$$

This means that we successfully proved that the program does not abort (i.e., it does not access an undefined variable), but also that the returned value is strictly positive (i.e., the loop is executed at least once). Note that this is the best result we can get on such an example with this formalism and the sign abstract domains. In particular, remark that the sign domain cannot count how many times the loop needs to be unfolded, hence the abstract derivation is infinite. Nevertheless, the analysis deduces significant information.

7. Related Work

Schmidt's paper on abstract interpretation of big-step operational semantics [20] was seminal but has had few followers. The only reported uses of big-step semantics for designing a static analyzer are those of [10] who built a big-step semantics-based foundation for program slicing by Gouranton and Le Metayer [10] and of Bagnara *et. al.* [1] concerned with building a static analyzer of values and array bounds in C programs.

Other systematic derivations of static analyses have taken small-step operational semantics as starting point. With the aim of analyzing concurrent processes and process algebras, Schmidt [21] discusses the general principles for such an approach and compares small-step and big-step operational semantics as foundations for abstract interpretation. Cousot [8] has shown how to systematically derive static analyses for an imperative language using the principles of abstract interpretation. Midtgaard and Jensen [15, 16] used a similar approach for calculating control-flow analyses for functional languages from operational semantics in the form of abstract machines. Van Horn and Might [22] show how a series of analyses for functional languages can be derived from abstract machines. An advantage of using small-step semantics is that the abstract interpretation theory is conceptually simpler and more developed than its big-step counterpart. In particular, accommodating non-termination is straightforward in small-step semantics. As both Schmidt and later Leroy and Grall [13] show, non-termination can be accommodated in a big-step semantics at the expense of accepting to work with infinite derivation trees defined by co-induction. Interestingly, the development of the formally verified CompCert compiler [12] started with big-step semantics but later switched to a mixture of small-step and big-step semantics. Poulsen and Mosses [19] have used refocusing techniques to automatically compile small-step semantics into PBS semantics.

Machine-checked static analyzers including the Java byte code verifier by [11] and the certified flow analysis of Java byte code by [6] also use a small-step semantics as foundation. Cachera and Pichardie [5] use denotational-style semantics for building certified abstract interpretations. In spite of the difference in style of the underlying semantics, these analyzers rely on the same formalization of abstract domains as lattices. The correctness proof also include similar proof obligations for the basic transfer functions.

In our Coq formalization we have striven to stay as close to Schmidt's original framework as possible, but there are a few deviations.

- Our development is based on a specific kind of big-step operational semantics i.e., the PBS rule format. For the formalization, this has the advantage that the rule format becomes precisely defined while still retaining full generality.
- Schmidt also considers infinite derivations for the concrete semantics. More precisely, the set of derivation trees is taken to be the greatest fixed point $\mathit{gfp}(\Phi)$ of the functional Φ induced by the inference rules. The trees can be ordered so that the set of semantic trees form a cpo, with a distinguished smallest element Ω, denoting the undefined derivation. The semantics of a term t in state E is then defined to be the least derivation tree that ends in a judgment of form $t, E \Downarrow r$. This tree can be obtained as the least fixed point of the functional $\mathcal{E} : \mathit{Term} \to \mathit{env} \to \mathit{gfp}(\Phi)$ induced by the inference rules.
- When constructing the abstract semantics, we only abstract conditions and transfer functions of concrete semantic rules. Schmidt's notion of covering relation between concrete and abstract rules is more flexible in that it allows the abstract semantics to be a completely different set of rules, as long as they can be shown to cover the concrete semantics. Also, we do not include extra meta-rules that can be shown to correspond to sound derivations (such as a fixed point rule for loops and a rule for weakening, for example) in the basic setup. As shown in Section 6, such meta-rules can be shown to be sound within our framework. This deviation guides the definition of the abstract semantics, helping its mechanization.
- Schmidt appeals to an external equation solver over abstract domains to make repetition nodes in a derivation tree. We show how to use an oracle analyzer to provide loop invariants that are then being verified by the abstract interpreter.

8. Conclusions and Future Work

Big-step operational semantics can be used to develop certified abstract interpretations using the Coq proof assistant. In this paper, we have described the foundations of a framework for building such abstract interpreters, and have demonstrated our approach by developing a certified abstract interpreter over a sign domain for a While language extended with an exception mechanism. The

correctness proof of the analyses has been conducted and verified using Coq [4]. The abstraction is performed in a systematic rule-by-rule fashion. While this may complicate the way that certain, more advanced analyses are expressed, this is deliberately done so that the approach can scale to larger semantics and other abstract domains.

The development is based on the PBS style of operational semantics. PBS leads to a well-defined, restricted yet expressive rule format that lends itself well to a formalization in Coq.

We first formalized PBS operational semantics using dependent types (Section 3) in order to obtain a precise model of the semantic foundations. When implementing this style of specification within Coq, it became apparent that Coq is not quite up to reasoning about a formalization in terms of pure, dependent types, and a less stringent model had to be adopted. On the other hand, Coq was fully adequate and very useful for reasoning about the abstraction of the semantics.

The definition of the abstract derivation is co-inductive, but co-induction only plays a well-defined and confined role in the development. In practice, Park's induction principle can be used to prove soundness of related analyses, and of abstract verifiers, as shown in Section 6.

Within our framework, defining a correct abstract interpreter is guided through several basic steps. We first have to define a set of concrete rules, which leads to a concrete semantics. Abstract domains and rules are then to be defined. If the atomic computations of these rules are locally related to the concrete ones, then the framework provides an abstract semantics correct by construction. Analyzers can then be defined, and their correctness amounts to relate them to this abstract semantics.

With the basic principles well established, there are a number of directions for future work. First, we want to apply this framework to develop program analyses for other types of properties. We notably plan to take advantage of the operational semantics to formalize data flow properties such as def-use of variables and its use in dependency analysis. This will be based on preliminary investigations [2] on how to reconstruct traditional execution traces and extract def-use information from derivation trees. Such non-local reasoning is crucial for precise analyses: it allows for instance to use the knowledge about the condition of a while loop to make more precise the abstract semantic context used to evaluate its body. This is the reason why our analyses cannot deduce that variable n is zero in the example at the end of Section 6.

Second, we want to extend our approach to model infinite computations, a standard issue when using big-step operational semantics. As already explained in [20] and recalled in Section 7, infinite computations can be accommodated by using infinite derivation trees for the concrete semantics, and ordering them into a complete partial order on which a least fixed point semantics can be defined. Our correctness theorems should be extended to this more general semantics.

Finally, we plan to test the scalability of the approach on a large semantics. The ultimate goal is to develop certified static analyses for JavaScript based on the JSCert PBS formalization, where the design and correctness proof of the analysis are guided by our framework. Developing such an analysis will furthermore enable us to test another aspect of the approach *viz.* to what extent our approach to certified abstract interpretation helps in maintaining and modifying large-scale analyses.

References

[1] R. Bagnara, P. M. Hill, A. Pescetti, and E. Zaffanella. Verification of C programs via natural semantics and abstract interpretation. In *Proc. of the IFM 2007 C/C++ Verification Workshop*, Technical Report ICIS-R07015, Institute for Computing and Information Sciences (iCIS), pages 75--80. Radboud University Nijmegen, The Netherlands, 2007.

[2] M. Bodin, T. Jensen, and A. Schmitt. Pretty-big-step-semantics-based certified abstract interpretation (preliminary version). In *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday,* Manhattan, Kansas, USA, 19--20th September 2013, volume 129 of *Electronic Proceedings in Theoretical Computer Science*, pages 360--383. Open Publishing Association, 2013.

[3] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A trusted mechanised javascript specification. In *POPL*, pages 87--100. ACM, 2014.

[4] M. Bodin, T. Jensen, and A. Schmitt. Certified abstract interpretation with pretty big-step semantics: Coq development files and analyzers results. `http://ajacs.inria.fr/coq/cpp2015/`, 2015.

[5] D. Cachera and D. Pichardie. A certified denotational abstract interpreter. In *ITP*, pages 9--24, 2010.

[6] D. Cachera, T. P. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. *Theor. Comput. Sci.*, pages 56--78, 2005.

[7] A. Charguéraud. Pretty-big-step semantics. In *ESOP*, pages 41--60. Springer, 2013. doi: 10.1007/978-3-642-37036-6_3.

[8] P. Cousot. The calculational design of a generic abstract interpreter. In *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238--252. ACM, 1977.

[10] V. Gouranton and D. L. Métayer. Dynamic slicing: a generic analysis based on a natural semantics format. *Journal of Logic and Computation*, 9(6), 1999. doi: 10.1093/logcom/9.6.835.

[11] G. Klein and T. Nipkow. Verified bytecode verifiers. *Theor. Comput. Sci.*, pages 583--626, 2003.

[12] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107--115, 2009. URL `http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf`.

[13] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207:284--304, 2009.

[14] C. McBride. Elimination with a motive. In *Types for proofs and programs*, pages 197--216. Springer, 2002.

[15] J. Midtgaard and T. Jensen. A calculational approach to control-flow analysis by abstract interpretation. In *SAS*, volume 5079 of *LNCS*, pages 347--362. Springer Verlag, 2008. doi: 10.1007/978-3-540-69166-2_23.

[16] J. Midtgaard and T. Jensen. Control-flow analysis of function calls and returns by abstract interpretation. In *ICFP*, pages 287--298. ACM, 2009. doi: 10.1145/1596550.1596592.

[17] D. Park. Fixpoint induction and proofs of program properties. In *Machine Intelligence 5*, pages 59--78. Edinburgh University Press, 1969.

[18] D. Pichardie. Building certified static analysers by modular construction of well-founded lattices. In *FICS*, volume 212 of *ENTCS*, pages 225--239, 2008. doi: 10.1016/j.entcs.2008.04.064.

[19] C. B. Poulsen and P. D. Mosses. Deriving pretty-big-step semantics from small-step semantics. In *Programming Languages and Systems*, pages 270--289. Springer, 2014.

[20] D. A. Schmidt. Natural-semantics-based abstract interpretation (preliminary version). In *SAS*, pages 1--18. Springer LNCS vol. 983, 1995. doi: 10.1007/3-540-60360-3_28.

[21] D. A. Schmidt. Abstract interpretation of small-step semantics. In *Proc. 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, Springer LNCS vol. 1192, pages 76--99, 1997.

[22] D. Van Horn and M. Might. Abstracting abstract machines. In *ICFP*, pages 51--62. ACM, 2010. doi: 10.1145/1995376.1995399.