Projet Ajacs

Deliverable WP2

Systematic derivation of a static analysis from a formal semantics

June 2016

This deliverable includes the following articles describing work done on WP2 for the first 18 months of the project.

- **Certified Abstract Interpretation with Pretty-Big-Step Semantics** by Martin Bodin, Thomas Jensen, and Alan Schmitt.
- **An Abstract Separation Logic for Interlinked Extensible Records** by Martin Bodin, Thomas Jensen, and Alan Schmitt.

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.
Copyright © 2015 ACM 978-1-4503-3300-9/15/01…$15.00.
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 $\mathit{ret}\, E \in \mathit{out}_s$.

$$\dfrac{\dfrac{\dfrac{\vdots}{e, E \Downarrow val\,k} \qquad \dfrac{\dfrac{\vdots}{s, E \Downarrow ret\,E'} \qquad \dfrac{\vdots}{while_1\, e\, s, ret\,E' \Downarrow ret\,E''}}{while_2\, e\, s, (E, val\,k) \Downarrow ret\,E''}\ \textsc{While2True}}{while_1\, e\, s, ret\,E \Downarrow ret\,E''}\ \textsc{While1}}{while\, e\, s, E \Downarrow ret\,E''}\ \textsc{While}$$

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 $+_1\ e$.

$$\begin{array}{l}\textsc{Add}_1\,(e)\\ \dfrac{e, E \Downarrow o \qquad +_2, (v_1, o) \Downarrow o'}{+_1\ e, (E, val\,v_1) \Downarrow o'}\end{array}$$

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 $val\,v_1$ 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 $\textsc{Abort}\text{E}\,(+_1\ e)$ 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 $\textsc{Abort}\text{E}\,(+_2)$ applies and the error is propagated. Otherwise, the rule $\textsc{Add}_2$ applies.

$$\begin{array}{l}\textsc{Add}_2\\ \dfrac{add\,(v_1, v_2) \rightsquigarrow v}{+_2, (v_1, val\,v_2) \Downarrow val\,v}\end{array}$$

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

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

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 $\sigma \in st\,(t)$. 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 $res\,(t)$.

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).

$$rule : (i \in \mathcal{I}) \to Rule_i$$

They also provide a function $\llcorner$ that maps rule identifiers to the syntactic term to which the rule applies.

$$\llcorner : \mathcal{I} \to term$$

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

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 $\textsc{VarUndef}\,(x)$.

$$\begin{array}{l}\textsc{Var}(x)\\ \dfrac{E[x] \rightsquigarrow v}{x, E \Downarrow val\,v}\quad x \in \mathrm{dom}(E)\end{array} \qquad \begin{array}{l}\textsc{VarUndef}(x)\\ \dfrac{}{x, E \Downarrow Err}\quad x \notin \mathrm{dom}(E)\end{array}$$

The predicate *cond* has the type

$$cond : (i \in \mathcal{I}) \to st\,(\llcorner_i) \to Prop$$

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 *type* : $\mathcal{I} \to \{Ax, R_1, R_2\}$ returns the format (axiom, rule 1, or rule 2) of the rule identified by $i \in \mathcal{I}$, and $\llcorner : \mathcal{I} \to term$ 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 AbortE (e_x) and AbortS (s_x) of Figure 3.

Figure 2: Concrete Semantics Definitions

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

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

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

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

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

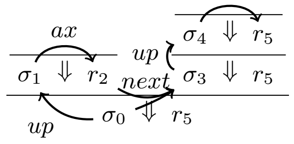
Figure 3: Concrete Semantics

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.



ax
ax
$\sigma_4 \Downarrow r_5$
up
$\sigma_1 \Downarrow r_2$
next
$\sigma_3 \Downarrow r_5$
up
$\sigma_0 \Downarrow r_5$

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 $\mathit{res}\,(\mathfrak{l}_i)$, and are thus described by a function of type

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

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

$$\overset{\mathrm{ID}}{\dfrac{\mathit{ax}\,(\sigma,\pi) \rightsquigarrow r}{\mathfrak{l}_{\mathrm{ID}}, \sigma \Downarrow r}} \quad \mathit{cond}_{\mathrm{ID}}\,(\sigma)$$

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

$$\overset{\mathrm{ID}}{\dfrac{\mathfrak{u}_{1,\mathrm{ID}}, \mathit{up}\,(\sigma) \Downarrow r}{\mathfrak{l}_{\mathrm{ID}}, \sigma \Downarrow r}} \quad \mathit{cond}_{\mathrm{ID}}\,(\sigma)$$

  Such a rule specifies a new term $\mathfrak{u}_{1,\mathrm{ID}}$ as described above, as well as a new semantic context $\mathit{up}\,(\sigma)$ of type $\mathit{st}\,(\mathfrak{u}_{1,\mathrm{ID}})$ and returns the result of evaluating $\mathfrak{u}_{1,\mathrm{ID}}$ in this context as the semantics of $\mathfrak{l}_{\mathrm{ID}}$. For such rules, the format thus implicitly requires that $\mathit{res}\,(\mathfrak{l}_{\mathrm{ID}}) = \mathit{res}\,(\mathfrak{u}_{1,\mathrm{ID}})$. Hence, the essence of a format 1 rule is the function *up* that maps σ to $\mathit{up}\,(\sigma)$. 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

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

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

$$\overset{\mathrm{ID}}{\dfrac{\mathfrak{u}_{2,\mathrm{ID}}, \mathit{up}\,(\sigma) \Downarrow r \qquad \mathfrak{n}_{2,\mathrm{ID}}, \mathit{next}\,(\sigma, r) \Downarrow r'}{\mathfrak{l}_{\mathrm{ID}}, \sigma \Downarrow r'}} \quad \mathit{cond}_{\mathrm{ID}}\,(\sigma)$$

  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 $\mathit{up}\,(\sigma) \in \mathit{st}\,(\mathfrak{u}_{2,\mathrm{ID}})$, and a function $\mathit{next}\,(\sigma,\cdot)$ transforming the result of the first inductive call into a semantic context of type $\mathit{st}\,(\mathfrak{n}_{2,\mathrm{ID}})$.

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

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

  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., $\mathit{res}\,(\mathfrak{l}_i) = \mathit{res}\,(\mathfrak{n}_{2,i})$.

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

$$\begin{array}{c} \mathit{Ax}_i(\mathit{ax} : (\sigma \in \mathit{st}\,(\mathfrak{l}_i)) \to \mathit{cond}_i\,(\sigma) \to \mathit{res}\,(\mathfrak{l}_i)) \\ R_{1,i}(\mathit{up} : (\sigma \in \mathit{st}\,(\mathfrak{l}_i)) \to \mathit{cond}_i\,(\sigma) \to \mathit{st}\,(\mathfrak{u}_{1,i})) \\ R_{2,i}\left(\begin{array}{c} \mathit{up} : (\sigma \in \mathit{st}\,(\mathfrak{l}_i)) \to \mathit{cond}_i\,(\sigma) \to \mathit{st}\,(\mathfrak{u}_{2,i}) \\ \mathit{next} : (\sigma \in \mathit{st}\,(\mathfrak{l}_i)) \to \mathit{cond}_i\,(\sigma) \to \mathit{res}\,(\mathfrak{u}_{2,i}) \to \mathit{st}\,(\mathfrak{n}_{2,i}) \end{array}\right) \end{array}$$

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.
```

$$\overset{\mathrm{ID}}{\dfrac{}{\mathfrak{l}_{\mathrm{ID}}, \sigma \Downarrow \mathit{ax}\,(\sigma)}} \quad \mathit{cond}_{\mathrm{ID}}\,(\sigma) \qquad\qquad \overset{\mathrm{ID}}{\dfrac{\mathfrak{u}_{1,\mathrm{ID}}, \mathit{up}\,(\sigma) \Downarrow r}{\mathfrak{l}_{\mathrm{ID}}, \sigma \Downarrow r}} \quad \mathit{cond}_{\mathrm{ID}}\,(\sigma)$$

$$\overset{\mathrm{ID}}{\dfrac{\mathfrak{u}_{2,\mathrm{ID}}, \mathit{up}\,(\sigma) \Downarrow r \qquad \mathfrak{n}_{2,\mathrm{ID}}, \mathit{next}\,(\sigma, r) \Downarrow r'}{\mathfrak{l}_{\mathrm{ID}}, \sigma \Downarrow r'}} \quad \mathit{cond}_{\mathrm{ID}}\,(\sigma)$$

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 $\mathit{res}\,(\mathfrak{l}_{\mathrm{ID}})$ to $\mathit{res}\,(\mathfrak{u}_{k,\mathrm{ID}})$. 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 $\mathit{ax} : \mathit{st} \to \mathit{option}\ \mathit{res}$ 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 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.

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

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

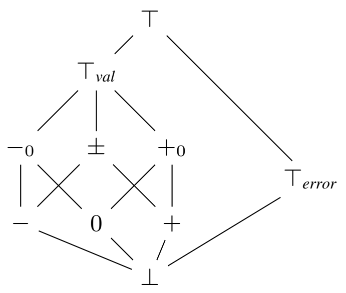


Figure 6: The Hasse diagram of the $valerr^\sharp$ lattice

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,

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

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

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

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

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

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

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

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 $\text{WHILE}\,(e,s)$ and $\text{WHILE1}\,(e,s)$. The former is a format 1 and thus only need an *up* function to be defined. The facts that it applies on $\mathfrak{l}_{\text{WHILE}(e,s)} = while\ e\ s$ and that its intermediate term is $\mathfrak{u}_{1,\text{WHILE}(e,s)} = while_1\ e\ s$ 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 $cond^\sharp_{\text{WHILE}(e,s)}\left(\sigma^\sharp\right)$, 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 $\mathfrak{u}_1 = while_1\ e\ s$. 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 $\text{WHILE1}\,(e,s)$ 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 $while_1\ e\ s$, 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 $ax\,(\sigma) \in \gamma\left(ax^\sharp\left(\sigma^\sharp\right)\right)$.
- 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 $up\,(\sigma) \in \gamma\left(up^\sharp\,(\sigma)\right)$.
- 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 $next\,(\sigma, r) \in \gamma\left(next^\sharp\left(\sigma^\sharp, r^\sharp\right)\right)$.

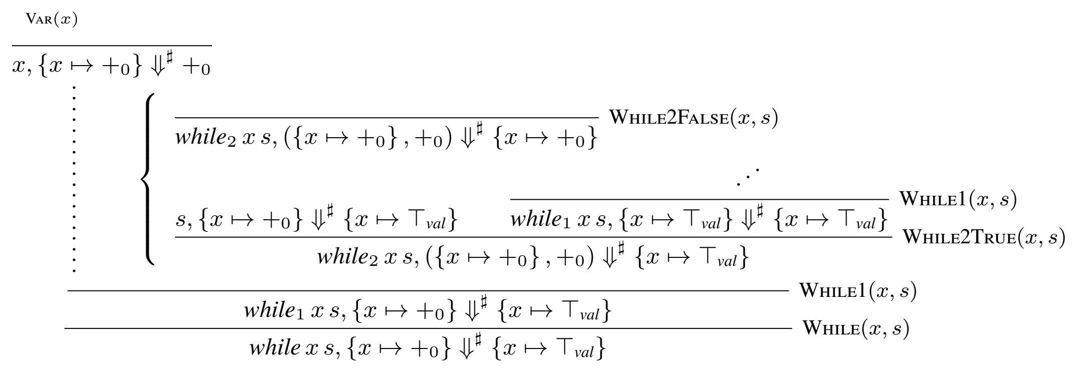
5.3 Inference Trees

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 $apply^\sharp_i$ 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.

$$apply^\sharp_i\left(\Downarrow^\sharp_0\right) = \left\{ (t,\sigma,r) \;\middle|\; \begin{array}{l} \exists \sigma_0, \exists r_0, \\ \quad \sigma \sqsubseteq^\sharp \sigma_0 \wedge r_0 \sqsubseteq^\sharp r \wedge \\ \quad (t,\sigma_0,r_0) \in apply_i\left(\Downarrow^\sharp_0\right) \end{array} \right\}$$

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



$\textsc{Var}(x)$
$x, \{x \mapsto +_0\} \Downarrow^\sharp +_0$
$\textsc{While2False}(x, s)$
$\mathit{while}_2\, x\, s, (\{x \mapsto +_0\}, +_0) \Downarrow^\sharp \{x \mapsto +_0\}$
$\textsc{While1}(x, s)$
$s, \{x \mapsto +_0\} \Downarrow^\sharp \{x \mapsto \top_{val}\}$
$\mathit{while}_1\, x\, s, \{x \mapsto \top_{val}\} \Downarrow^\sharp \{x \mapsto \top_{val}\}$
$\textsc{While2True}(x, s)$
$\mathit{while}_2\, x\, s, (\{x \mapsto +_0\}, +_0) \Downarrow^\sharp \{x \mapsto \top_{val}\}$
$\textsc{While1}(x, s)$
$\mathit{while}_1\, x\, s, \{x \mapsto +_0\} \Downarrow^\sharp \{x \mapsto \top_{val}\}$
$\textsc{While}(x, s)$
$\mathit{while}\, x\, s, \{x \mapsto +_0\} \Downarrow^\sharp \{x \mapsto \top_{val}\}$

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.

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

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 $\top_{val}$, 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 $\top_{val}$ (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 $\top_{val}$. 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 $\top_{val}$, 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 $\{x \mapsto \top_{val}\}$ 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 co-inductive 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 $\mathit{cond}^\sharp$ 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 $r^\sharp \in \mathit{res}^\sharp$.*

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

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 $\mathit{while}_1\ s_1\ s_2$).

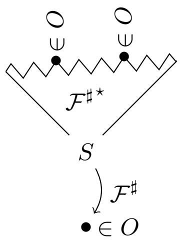


Figure 11: An Illustration of the Action of the Verifier

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.

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

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.

An Abstract Separation Logic for Interlinked Extensible Records

Martin Bodin & Thomas Jensen & Alan Schmitt

Inria, France

Résumé

The memory manipulated by JAVASCRIPT programs can be seen as a heap of extensible records storing values and pointers. We define a separation logic for describing such structures. In order to scale up to full-fledged languages such as JAVASCRIPT, this logic must be integrated with existing abstract domains from abstract interpretation. However, the frame rule—which is a central notion in separation logic—does not easily mix with abstract interpretation. We present a domain of heaps of interlinked extensible records based on both separation logic and abstract interpretation. The domain features spatial conjunction and uses summary nodes from shape analyses. We show how this domain can accommodate an abstract interpretation including a frame rule.

1. Introduction

The memory of a JAVASCRIPT program is a dynamic and complex heap of extensible records storing values and pointers. Fields can be added and removed from records dynamically, and their presence can be tested. Moreover, records are not constrained by a static type structure, which further complicates the analysis of the shapes that these interlinked objects may form. Obtaining a good approximation of the memory structure of a JAVASCRIPT program is a challenge for static analysis, even if we restrict other features of the language such as computed field names and dynamic code generation.

In this paper, we present a solution to this challenge, by mixing elements of separation logic and shape analysis, and integrating them into an abstract interpretation framework. Separation logic in itself is not adequate for describing the inter-connected heaps of JAVASCRIPT. First, separation logic is based on some additional structures, such as lists or trees. For JAVASCRIPT, such structures can be difficult to identify, as illustrated by Gardner *et al.* [10]. Second, JAVASCRIPT native structures tend to not separate nicely. Gardner *et al.* propose to remedy this through a *partial* separation operator $\uplus$ ("seppish"). The formula $P \uplus Q$ describes a heap which can be split in two heaps, one satisfying P and the other Q; but these two heaps do not need to be disjoint. Here, we pursue this idea, but instead of introducing a new operator in separation logic, we inject ideas from shape analyses, and use summary nodes for modelling the portion of memory that may be shared. In this way, we move the approximation into the shape structures while keeping a precise separation operator $\star$.

The work described here is part of a larger project on certified static analyses in which static analysis tools are developed an proved correct based on a mechanised formalisation of the semantics of the underlying language. More precisely, the aim is to build on the JSCERT [2] semantics for JAVASCRIPT, a pretty-big step operational semantics [6] entirely written in Coq. The size of the JAVASCRIPT's semantics imposes that we take an approach that is both principled and mechanisable. We base the development on the theory of abstract interpretation. Abstract interpretation [7] provides a powerful theory for finding and proving loop invariants within a program, assuming minimal structure on the

This research was partially supported by the French ANR-10-LABX-07-01 Laboratoire d'excellence CominLabs and ANR-14-CE28-0008 project Ajacs.

space of abstract domains. For the mechanisation, certification in proof assistants such as Coq is needed. We have previously built a Coq library [3] providing the building blocks for constructing an abstract interpretation from a pretty-big step operational semantics, following initial ideas of Schmidt [14]. In that paper, we showed how to derive abstract rules from concrete ones in such a way that abstract derivations are correct by construction. Here, we show how to extend this approach with abstractions of heaps using techniques from both separation logic and shape analysis, in order to give reasonable results for JavaScript.

The abstract domains arising from separation logic do not have the rich structure of lattices encountered in many abstract interpretations. The theory of abstract interpretation, however, does generalise to the setting where the underlying structure is that of only a subset of a pre-order. We shall hence use this more general framework, which provides the same correctness guarantees but does not explain how to compute a best analysis result.

Separation logic provides useful notions for the analysis of heap-manipulating programs. In separation logic, abstract rules are only given locally: they only state what is changed by a given program, assuming that everything not mentioned is left unchanged. The frame rule then allows to add an unchanged partial heap to the analysed result. This mechanism is very powerful to locally reason about programs. However JavaScript introduces some new issues about the frame rule: Reynolds [13, Section 3.5] stated that the frame rule can not be applied as-is if the language allows constructions similar to JavaScript's *delete* operator. We address this issue using a special value $\boxtimes$.

The main contributions of this paper are as follows.

- A combination of separation logic and shape analysis, which allows to use the separation $\star$ for disjoint domains, and shapes for complex domains with potential sharing.
- An alternative to the $\boxed{\star}$ operator that better fits the frame rule.
- An integration of separation logic into an abstract interpretation framework based on pre-orders and big-step operational semantics, extending our previous work [3].

The paper is organised as follows. We first present our toy language OWhile. Our logic is presented in two steps: first, a logic over abstract domains is built in Section 3; its structure should not be surprising to a reader familiar with separation logic. A crucial step of the approach is the addition of membranes, in Section 4.1, to deal with the frame rule. Second, we add the summary nodes from shape analyses to the domain in Section 5. Section 6 presents how we build our program logic for OWhile, leading in Section 6.4 to the correctness of our abstract semantics. Section 7 examines related work and Section 8 concludes.

2. The OWhile Language

We define our analyses on a small imperative language with interlinked records, called OWhile. This language is inspired from JavaScript's memory model but we shall disregard all aspects related to prototype inheritance or type conversion. We can create new records (which we call *objects*), and read, write, and delete their fields (also called *properties* in JavaScript). Records are *interlinked* because their fields may contain pointers to other objects.

The syntax of our language is presented in Figure 1. A detailed version of the concrete semantics can be found in Appendix A, but it comes with no surprises for a pretty-big-step semantics [6]. There are only numbers in the language, so for the purpose of branching (instructions *if* and *while*), the number 0 behaves as *false*, and any other number as *true*. The operation ? non-deterministically returns a number. Fresh objects are created by the {} expression. We can access the field `f` of an object computed by e through e`.f`. We can check the presence or absence of a given field `f` in an

$$\begin{array}{llll} s ::= \mathit{skip} & \mid s_1;\ s_2 & \mid \mathit{if}\, e\, s_1\, s_2 \\ \quad \mid \mathit{while}\, e\, s & \mid \mathit{throw} & \mid \mathtt{x} := e \\ \quad \mid e_1.\mathtt{f} := e_2 & \mid \mathit{delete}\, e.\mathtt{f} & \end{array}$$

(a) Statements

$$\begin{array}{lllll} e ::= n \in \mathbb{Z} & \mid ? & \mid \mathtt{x} \in \mathit{Var} & \mid \mathit{nil} \\ \quad \mid \{\} & \mid e.\mathtt{f} & \mid \mathtt{f}\,\mathtt{in}\, e & \mid \neg\, e \\ \quad \mid =\, e_1\ e_2 & \mid \bowtie e_1\ e_2 & (\bowtie\, \in \{>, +, -\}) & \end{array}$$

(b) Expressions

Figure 1: The syntax of the OWhile language

object computed by e through $\mathtt{f}\,\mathtt{in}\, e$, which returns 1 if the field is present and 0 otherwise. As in JavaScript, writing to the field $\mathtt{f}$ of an object adds the field if it is not already present, and deleting an object's field succeeds even if the field is absent. There is no explicit declaration of variables: as for fields, writing a variable which is not defined creates it. A program may abort for the following reasons: explicitly running *throw*, reading a variable or a field that is not assigned, or accessing the field of a value that is not an object. The state S of a program is composed of two components.

- An environment (also called *store* in JavaScript parlance) $E : \mathit{Var} \rightharpoonup \mathit{Val}$, where Var is the set of variable names and Val is the set of values. A value $v \in \mathit{Val}$ can either be a location l^i (including the special null location l^0, always out of the domain of S), or a basic value $n \in \mathbb{Z}$.
- A heap $H : \mathit{Loc} \rightharpoonup \mathfrak{F} \rightharpoonup \mathit{Val}$, where $\mathit{Loc} = \left\{ l^i \;\middle|\; i \in \mathbb{N}^\star \right\}$ is the set of non-null locations, and $\mathfrak{F}$ the set of field names. We assume $\mathfrak{F}$ to be infinite.

We define $\mathit{dom}(S)$ to be $\mathit{dom}(E) \cup \mathit{dom}(H)$ where E and H are the respective environment and heap of S. The function *fresh* takes a state S and returns a location fresh in S, i.e., $l^j \notin \mathit{dom}(H)$.

3. Abstract Domains

3.1. Abstract State Formulae

In this section we build a separation logic over an abstract domain of base values. There are various ways of representing separation logic; our logic is based on the work of [10]. Abstract state formulae $\phi \in \mathit{State}^\sharp$ model pairs of concrete heaps and stores. These formulae are defined as follows.

$$\phi ::= \mathit{emp} \mid \phi_1 \star \phi_2 \mid \mathtt{x} \doteq v^\sharp \mid l \mapsto \{o\} \qquad\qquad o ::= \mathtt{f} : v^\sharp,\, o \mid \_ : v^\sharp$$

These formulae make use of abstract locations $l \in \mathit{LLoc}^\sharp$. These locations are identifiers which are meant to represent one concrete (non-null) location. In the concretisation of formulae, abstract locations are related to concrete locations by a valuation $\rho : \mathit{LLoc}^\sharp \rightharpoonup_{\mathit{inj}} \mathit{Loc}$ (where $\rightharpoonup_{\mathit{inj}}$ denotes a partial injection). Concrete locations in Loc are written with an exponent l^i whilst abstract locations use indexes or primes: l, l_i, or l'.

The structure of the abstract domain for values $v^\sharp \in \mathit{Val}^\sharp$ is described in detail in Section 3.2. For now, we just note that this abstract domain contains abstract locations (detailed below) and abstract properties of numeric values (sign, parity, intervals, …). The concretisation function γ_ρ of abstract values relates them to sets of concrete values.

The formula *emp* describes the empty heap and empty environment. The spatial conjunction $\phi_1 \star \phi_2$ describes the set of all heaps and environments which we can separate into two smaller heaps and environments, each respecting one of the two sub-formulae ϕ_1 and ϕ_2. The $\star$ operator is commutative, associative, and has *emp* as neutral element. The formula $\mathtt{x} \doteq v^\sharp$ states that the value of the variable `x` satisfies the property $v^\sharp$. We follow the tracks of [12] and do not consider this formula pure. As we are not interested in concurrency in this paper, we use a simpler version than [12] where we either have full permission over `x` if $\mathtt{x} \doteq v^\sharp$ is present, and no permission otherwise. The construction $l \mapsto \{o\}$ describes the set of heaps whose only defined location $\rho(l)$ points to an object abstracted by $\{o\}$.

Objects are abstracted as a list associating fields to abstract values, with an additional default abstract value for the other fields present in the object.[1] All the specified field names of an object are supposed to be different. An abstract object $\left\{\mathtt{f}_1 : v_1^\sharp, \ldots, \mathtt{f}_n : v_n^\sharp, \_ : v_r^\sharp\right\}$ represents the set of objects whose respective fields $\mathtt{f}_1, \ldots, \mathtt{f}_n$ are abstracted by respectively $v_1^\sharp, \ldots, v_n^\sharp$, and all the other fields are abstracted by $v_r^\sharp$. Abstract values also include the possibility to state that a field is undefined. This is expressed through the special value $\boxtimes$. The abstract object $\left\{\mathtt{f} : \boxtimes, \mathtt{g} : v^\sharp, \_ : \top\right\}$ thus describes the set of objects such that each object has no field `f` and its field `g` can be abstracted by $v^\sharp$. Similarly, $\{\_ : \boxtimes\}$ describes the singleton of the empty object, which is returned by $\{\}$.

An alternative approach to model heaps and objects is to allow the separation of fields themselves, as in $l \overset{\mathtt{f}}{\mapsto} v_1^\sharp \star l \overset{\mathtt{g}}{\mapsto} v_2^\sharp$, in a way similar to [10]. We experimented with this approach and discovered that it results in complex interactions with the frame rule. We thus follow a simpler approach here.

3.2. Abstract Values and Abstract Object

Our separation logic formulae are parameterized over an abstract domain describing the base values which variables and fields can contain. In line with the dynamic typing of JavaScript, we shall consider an abstract domain containing both numerical values and locations. Hence, a variable may contain both types of values depending on the flow of control, and the abstract domain has to be able to join such values together.

In addition, when analyzing JavaScript's heap, we must take into account expressions like `f in` e whose result depends on the absence of a field. We thus have to track whether fields can be undefined.[2] To this mean, we attach a boolean to the abstract values to indicate whether the concrete value can be undefined, as illustrated before with the value $\boxtimes$. We use this boolean at two different places: to indicate that a field is possibly undefined, but also to indicate that a variable is possibly undefined.

Suppose a lattice domain $\mathbb{Z}^\sharp$ carrying abstract properties of numeric, or basic, values (sign, parity, intervals, …). Such a domain must store information about the different instances of values: basic values, locations, the possibility of being the special location l^0, and the possibility of being undefined. We define abstract values to be tuples of the form $\left(n^\sharp, nil^?, L, d\right)$, where $n^\sharp \in \mathbb{Z}^\sharp$ denotes the possible basic values which can be represented by this abstract value; $nil^?$ is a boolean stating whether the value can be l^0 (denoted by nil) or not (denoted by $\overline{nil}$); $L \in \mathcal{P}_f\left(LLoc^\sharp\right)^\top$ denotes the possible location values ($\mathcal{P}_f\left(LLoc^\sharp\right)^\top$ denotes the set of finite subsets of $LLoc^\sharp$ augmented with a $\top$ element); and the boolean $d \in \{\boxtimes, \square\}$ denotes whether the value can be undefined (denoted by $\boxtimes$) or can not (denoted by $\square$). Each part of these tuples carries the information about a kind of value.

For the sake of readability, we will identify the projections of an abstract value $v^\sharp$ with $v^\sharp$ itself if all the other projections are bottoms elements of their respective lattice. For instance we will write $n^\sharp$ to mean $\left(n^\sharp, \overline{nil}, \emptyset, \square\right)$, nil to mean $(\bot, nil, \emptyset, \square)$, and $\boxtimes$ to mean $\left(\bot, \overline{nil}, \emptyset, \boxtimes\right)$. We will also identify l with $\left(\bot, \overline{nil}, \{l\}, \square\right)$. To avoid the cumbersome tuple notation, we will use the natural join operation on this domain and write values such as $l \sqcup \boxtimes$.

The order on the tuple is the usual product order: a tuple is less than another if all its projections are less than the others. Sets of locations are ordered using the usual set lattice. The definition part

[1]This default field is sometimes called a *summary node* in the literature; we do not use this name as summary nodes denote a different concept in this paper.

[2]This abstract value is different from the JavaScript value `undefined`.

d is ordered by $\square \sqsubseteq \boxtimes$ as $\square$ forces the value to be defined while $\boxtimes$ allows (without forcing) it to be undefined. We similarly define $\overline{nil} \sqsubseteq nil$. We use the symbol $\sqsubseteq$ to denote the order within a lattice, that is over abstract values and abstract objects.

We can now define an order on abstract objects as follows: two objects are ordered, $\{o_1\} \sqsubseteq \{o_2\}$ if all the fields of $\{o_1\}$ are associated with a value which is smaller than the value of the same field in $\{o_2\}$. To check the order relation between two objects, we rely on the default value for all fields not explicitly mentioned in the object. With this value, we can rewrite the two objects so that they refer to the same fields, using the following rewriting equality,

$$\left\{\mathtt{f}_1 : v_1^\sharp, \dots, \mathtt{f}_n : v_n^\sharp, \_ : v_r^\sharp\right\} = \left\{\mathtt{f}_1 : v_1^\sharp, \dots, \mathtt{f}_n : v_n^\sharp, \mathtt{g} : v_r^\sharp, \_ : v_r^\sharp\right\}$$

which holds provided that $\mathtt{g}$ is not one of the $\mathtt{f}_1$, …, $\mathtt{f}_n$. This order equips abstract objects with a lattice structure, with $\sqcup$ and $\sqcap$ computing the abstract object whose fields are associated to the results of the corresponding operator $\sqcup$ or $\sqcap$ applied on the corresponding fields of the two operands (completed such that they have the same fields):

$$\begin{aligned}&\left\{\mathtt{f}_1 : v_1^\sharp, \dots, \mathtt{f}_n : v_n^\sharp, \_ : v_r^\sharp\right\} \sqcup \left\{\mathtt{f}_1 : {v'}_1^\sharp, \dots, \mathtt{f}_n : {v'}_n^\sharp, \_ : {v'}_r^\sharp\right\}\\ &\qquad = \left\{\mathtt{f}_1 : v_1^\sharp \sqcup {v'}_1^\sharp, \dots, \mathtt{f}_n : v_n^\sharp \sqcup {v'}_n^\sharp, \_ : v_r^\sharp \sqcup {v'}_r^\sharp\right\}\end{aligned}$$

4. The Frame Rule

Our main contribution deals with the interaction between formulae and the frame rule. In order to introduce it, we first detail how this rule typically works. The frame rule defines how to extend *Hoare triples* using the separation operator $\star$. A *Hoare triple* $\phi_1, t \Downarrow^\sharp \phi_2$ states that the term t changes any heap that satisfies formula ϕ_1 in a heap that satisfies formula ϕ_2. In our setting, a heap satisfies a formula if it belongs to its concretisation. A heap h belongs to the concretisation of a formula $\phi_1 \star \phi_2$ if it can be split in disjoint heaps h_1 and h_2 such that h_1 satisfies ϕ_1 and h_2 satisfies ϕ_2.

$$\textsc{Frame}\quad \frac{\phi_1, t \Downarrow^\sharp \phi_2}{\phi_1 \star \phi_c, t \Downarrow^\sharp \phi_2 \star \phi_c}$$

For the frame rule to be correct, it is crucial that if $\phi_1 \star \phi_c$ is defined (the set of concrete heaps it denotes is not empty), then $\phi_2 \star \phi_c$ is defined. In our setting, this may not be the case when new abstract locations are introduced in ϕ_2. For instance, consider the abstract rule NewObj, which builds the Hoare triple $emp, \{\} \Downarrow^\sharp l \mapsto \{\_ : \boxtimes\}$. The result contains an additional location l which we would like to keep fresh from the initial abstract heap emp. However, the frame rule applied as-is can add a new fact about l and generate the Hoare triple $l \mapsto \{\_ : \boxtimes\}, \{\} \Downarrow^\sharp l \mapsto \{\_ : \boxtimes\} \star l \mapsto \{\_ : \boxtimes\}$, which is wrong as the result formula has an empty concretisation (because l is not separated) whilst a concrete derivation tree can easily be derived. This problem also occurs when renaming abstract locations, as is described in Section 5.

To ensure the soundness of the frame rule, we have to introduce *scopes* for identifiers in a formula. For instance, the scope of a newly created location l should be restricted to the result formula, as in $emp, \{\} \Downarrow^\sharp (\nu l \,|\, l \mapsto \{\_ : \boxtimes\})$: this states that any mention of l outside the formula is actually a different identifier. Since we not only need to restrict the scope of identifiers, but also relate names inside a scope to names outside the scope, we introduce the notion of *membrane*. A membrane M traces the links between these two scopes. Each context added by the frame rule has to be converted when entering a membrane. Membranes behave like substitutions: we can compose them through $\circ$ and apply them to a formula ϕ to update its identifiers. Our version of the frame rule, defined in below, relies on membranes for its soundness.

4.1. Membranes

Membranes M are defined as a set of scope changes m, which can be caused either because a location has been renamed, or because a new location has been allocated. The abstraction Φ of heap and environment is now a couple of a formula ϕ and a rewriting membrane M, written $(M \mid \phi)$. We call the simple formulae ϕ *inner formulae* and the membraned formulae Φ *formulae*.

$$m \in \mathfrak{M} ::= l \to l' \mid \nu l \qquad\qquad \Phi ::= (M \mid \phi) \qquad M \in \mathcal{P}_f(\mathfrak{M})$$

We impose left-hand sides of scope changes to only appear once in a given membrane. We also impose that in a formula $\Phi = (M \mid \phi)$, any location l in ϕ is present on the right-hand side of a scope change or as a new name νl in M. We define the domain $\mathit{dom}(M)$ of a membrane M as the set of left-hand sides of its rewritings, and the codomain $\mathit{codom}(M)$ as the union of the set of right-hand sides of its rewritings and the set of newly allocated locations. The interface $\mathit{interface}(\Phi)$ of a formula $\Phi = (M \mid \phi)$ is the domain of its membrane $\mathit{dom}(M)$: these locations are accessible from the outside of the formula. The substitution $M(\phi)$ applied to inner formulae works as expected: it renames every abstract locations either as values or as memory cells. Trivial rewritings such as $l \to l$ are allowed and sometimes required: an abstract location l may be unchanged by the membrane, but it still has to be in the interface; the domain names of the inner and scope scopes are independent.

Let us consider a simple example: $\Phi = (l_0 \to l \mid \mathtt{x} \doteq l \star l \mapsto \{\mathtt{f} : l, \_ : \boxtimes\})$. The membrane $\{l_0 \to l\}$ renames the outer location identifier l_0 to the inner location l. If the frame rule introduces a context $\phi_c = l_1 \mapsto \{\mathtt{f} : l_0, \_ : \boxtimes\}$ referring to l_0, the integration of ϕ_c in the membrane leads to the renaming of l_0 into l for a final formula $(l_0 \to l \mid \mathtt{x} \doteq l \star l \mapsto \{\mathtt{f} : l, \_ : \boxtimes\} \star l_1 \mapsto \{\mathtt{f} : l, \_ : \boxtimes\})$. On the other hand, if the frame rule introduces a context with a l such as $\phi_c = l_1 \mapsto \{\mathtt{f} : l, \_ : \boxtimes\}$, this l is actually different from the one in Φ. In this case, Φ is α-renamed, for instance to $\Phi = (l_0 \to l' \mid \mathtt{x} \doteq l' \star l' \mapsto \{\mathtt{f} : l', \_ : \boxtimes\})$, to avoid the capture of l when ϕ_c enters the membrane. Note that α-renaming does not change the interface of a formula: only its codomain is modified.

Formulae are used to abstract states (environment and heap), but there are places in our semantics where additional values are carried. In pretty-big-step, we use *intermediate* terms along the execution, which require to carry additional values. For example, the assignment $\mathtt{x} := e$ involves two steps (evaluating the expression and updating the state) so we introduce an intermediate term $\mathtt{x} :=_1$ whose semantic context consists of a state and a value to assign to $\mathtt{x}$ and whose result is the update state. These values can contain locations and must be placed inside membranes: we shall thus sometimes manipulate formulae of the form $(M \mid l \mapsto \{o\}, l \sqcup l')$ where the value $l \sqcup l'$ represents the value of the intermediate semantic context. All operations defined on usual formulae can be extended to extended formulae. We do not show the details here for space reasons.

4.2. Separating Formulae

To express the frame rule in our formalism, the frame has to manipulate membranes; we define the operator $\circledast$ (read "in frame") taking two formulae $\Phi_o = (M_o \mid \phi_o)$ and $\Phi_i = (M_i \mid \phi_i)$—$i$ stands for "inner" and o for "outer"—which intuitively builds $(M_o \mid \phi_o \star (\phi_i \mid M_i))$ (this formula does not fit the grammar of formulae as-is): the inner formula is considered in the context of the outer one. This operation is associative, but not commutative. It performs an α-renaming of the inner identifiers of ϕ_i to prevent conflicts with M_i, then pushes ϕ_o through the membrane M_i. For instance for $\Phi_o = (l_0 \to l, k_0 \to k \mid k \mapsto \{\mathtt{g} : k, \mathtt{h} : l, \_ : \boxtimes\})$ and $\Phi_i = (l \to k \mid k \mapsto \{\mathtt{f} : k \sqcup \mathit{nil}, \_ : \boxtimes\})$, the identifier k is used both in ϕ_i and in ϕ_o, but because of the membranes, it represent a different set of concrete locations in both formulae. We thus α-rename k into k' to avoid name conflict:

$$\begin{aligned}
&(l_0 \to l, k_0 \to k \,|\, k \mapsto \{\mathtt{g} : k,\, \mathtt{h} : l,\, \_ : \boxtimes\}) \boxed{\star} (l \to k \,|\, k \mapsto \{\mathtt{f} : k \sqcup \mathit{nil},\, \_ : \boxtimes\}) \\
&\quad = (l_0 \to l, k_0 \to k \,|\, k \mapsto \{\mathtt{g} : k,\, \mathtt{h} : l,\, \_ : \boxtimes\}) \boxed{\star} \left(l \to k' \,\middle|\, k' \mapsto \{\mathtt{f} : k' \sqcup \mathit{nil},\, \_ : \boxtimes\}\right) \\
&\quad = \left(l_0 \to k', k_0 \to k \,\middle|\, k \mapsto \{\mathtt{g} : k,\, \mathtt{h} : k',\, \_ : \boxtimes\} \star k' \mapsto \{\mathtt{f} : k' \sqcup \mathit{nil},\, \_ : \boxtimes\}\right)
\end{aligned}$$

Because of the composition of membranes, l, which was an identifier introduced by M_o but substituted by M_i, was removed. Membranes are meant to be composed by the $\boxed{\star}$ operator, and the domain of the inner membrane should thus be in the codomain of the outer one: $\mathit{dom}(M_i) \subseteq \mathit{codom}(M_o)$.

We define $\boxed{\star}$ as follows. Given and $\Phi_o = (M_o \,|\, \phi_o)$ $\Phi_i = (M_i \,|\, \phi_i)$ such that $\mathit{dom}(M_i) \subseteq \mathit{codom}(M_o)$, we α-rename Φ_i into $\Phi'_i = \left(M'_i \,\middle|\, \phi'_i\right)$ such that $\mathit{codom}(M'_i) \cap \mathit{codom}(M_o) = \emptyset$. We then make the formula ϕ_o enter the membrane M_i:

$$\Phi_o \boxed{\star} \Phi_i = \Phi_o \boxed{\star} \Phi'_i = \left(M'_i \circ M_o \,\middle|\, M'_i(\phi_o) \star \phi'_i\right)$$

We are now ready to state our frame rule.

$$\textsc{Frame}\quad \frac{\Phi, t \Downarrow^\sharp \Phi'}{\Phi_c \boxed{\star} \Phi, t \Downarrow^\sharp \Phi_c \boxed{\star} \Phi'}$$

Although our program logic is not introduced until Section 6, here is an example of how the frame rule can be used. Consider the program *if*? *skip* ($\mathtt{x} := ?$) where the branch is chosen randomly, one branch does nothing while the other assigns a random value to $\mathtt{x}$. The empty branch of the *if* can be given the Hoare triple $\mathit{emp}, \mathit{skip} \Downarrow^\sharp \mathit{emp}$, and the other branch the Hoare triple $\mathtt{x} \doteq \boxtimes, \mathtt{x} := ? \Downarrow^\sharp \mathtt{x} \doteq \top_{\mathbb{Z}}$. The first branch can then be extended using the frame rule to the triple $\mathtt{x} \doteq \boxtimes, \mathit{skip} \Downarrow^\sharp \mathtt{x} \doteq \boxtimes$. Since both branches now have the same assumption, they may be merged together for the whole conditional: $\mathtt{x} \doteq \boxtimes, \mathit{if}?\ \mathit{skip}\ (\mathtt{x} := ?) \Downarrow^\sharp \mathtt{x} \doteq \boxtimes \sqcup \top_{\mathbb{Z}}$.

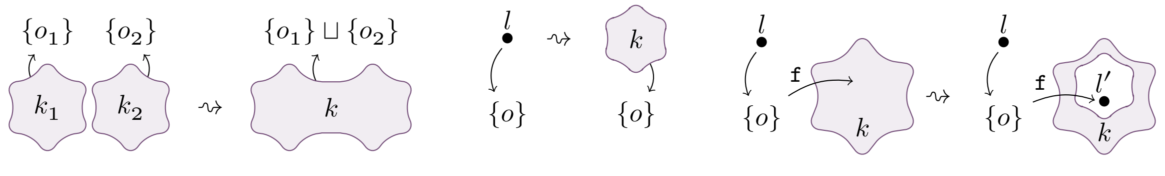
5. Adding Summary Nodes

Up to this point, the formulae which we have defined reflect precisely the structure of the concrete heap. However, this approach is not viable in the presence of loops. We need a way to forget about some information of the structure, in particular its size. To this end, we reuse the idea of summary nodes from shape analysis, by adding a new kind of abstract locations $k \in \mathit{KLoc}^\sharp$ which represent a set (finite and possibly empty) of concrete locations. We call them *summary locations.* As with $\mathit{LLoc}^\sharp$, this new set of abstract locations $\mathit{KLoc}^\sharp$ is supposed to be a new, infinite, set of identifiers. We note abstract locations as $h \in \mathit{KLoc}^\sharp \uplus \mathit{LLoc}^\sharp$.

Abstract values have been defined in Section 3.2 as tuples $\left(n^\sharp, \mathit{nil}^?, L, d\right)$, where $L \in \mathcal{P}_f\left(\mathit{LLoc}^\sharp\right)^\top$. We update them to track summary nodes by changing their projection L to $L \in \mathcal{P}_f\left(\mathit{LLoc}^\sharp \uplus \mathit{KLoc}^\sharp\right)^\top$. Values are thus abstracted by $\mathit{Val}^\sharp = \mathbb{Z}^\sharp \times \left\{\overline{\mathit{nil}}, \mathit{nil}\right\} \times \mathcal{P}_f\left(\mathit{LLoc}^\sharp \uplus \mathit{KLoc}^\sharp\right)^\top \times \{\square, \boxtimes\}$.

In formulae, summary locations may occur on the left-hand side of heaps $k \mapsto \{o\}$, denoting heaps where *every* concrete location in the concretion of k maps to a concretion of o. When k occurs as a value, its concretion is any single location denoted by k. Note the asymmetry in $k \mapsto \{\mathtt{f} : k,\, \_ : \boxtimes\}$, which means that every concrete location represented by k has a field $\mathtt{f}$ pointing to a concrete location in the set represented by k, but there is no relation between these two concrete locations. In particular, they need not be the same.

The set of formulae with summary nodes is defined as follows.



(a) Summarizing two summary nodes (b) Summarizing an abstract location (c) Materialisation

Figure 2: Picturisation of membrane operations

$$\begin{aligned}
\phi &::= \mathit{emp} \mid \phi_1 \star \phi_2 & h &::= l \mid k & m \in \mathfrak{M} &::= h \to h_1 + \dots + h_n \\
&\mid \mathtt{x} \doteq v^\sharp & o &::= \mathtt{f} : v^\sharp,\, o & &\mid \nu h \\
&\mid h \mapsto \{o\} & &\mid \_ : v^\sharp & \Phi &::= (M \mid \phi) \qquad M \in \mathcal{P}_f(\mathfrak{M})
\end{aligned}$$

Abstract values $v^\sharp$ can now contain basic values, abstract locations h (which can be summary nodes k or precise abstract locations l), the special *nil*, and the special abstraction $\boxtimes$. We update the definition of domain and codomain of membranes as expected:

$$\begin{aligned}
\mathit{dom}(h \to h_1 + \dots + h_n) &= \{h\} & \mathit{codom}(h \to h_1 + \dots + h_n) &= \{h_1, \dots, h_n\} \\
\mathit{dom}(\nu h) &= \emptyset & \mathit{codom}(\nu h) &= \{h\} \\
\mathit{dom}(M) &= \bigcup_{m \in M} \mathit{dom}(m) & \mathit{codom}(M) &= \bigcup_{m \in M} \mathit{codom}(m)
\end{aligned}$$

As renamings can now map an abstract location to several abstract locations, substitutions $M(\phi)$ can now duplicate memory cells: $\{k \to k_1 + k_2\}(k \mapsto \{o\}) = k_1 \mapsto \{o[k_1 \sqcup k_2/k]\} \star k_2 \mapsto \{o[k_1 \sqcup k_2/k]\}$.

There are two basic operations on summary locations: summarizations and materializations. These two operations rename abstract locations, thus changing the scope of formulae: membranes are a crucial point for their soundness in accordance to their interaction with the frame rule. Let us first only consider an inner formula ϕ.

The summarization consists in merging abstract locations h_1, ..., h_n into a single new summary node k. Figures 2a and 2b picture two examples of summarizations, respectively of two summary nodes, and of an abstract location. It allows to loose information about the structure of h_1, ..., h_n; typically to get a loop invariant. In order to perform a summarization, we need to have in the considered inner formula ϕ the explicit definition of all these abstract locations: it is not possible to summarize l and l' in the formula $l \mapsto \{\mathtt{f} : l', \_ : \boxtimes\}$ as we do not have access to the resource l'. Let us thus suppose that the formula ϕ is of the form $h_1 \mapsto \{o_1\} \star \dots \star h_n \mapsto \{o_n\} \star \phi'$. The summarization of h_1, ..., h_n into k, provided that k does not appear in ϕ, is the following formula.

$$\left(h_1 \to k, \dots, h_n \to k \,\middle|\, (k \mapsto \{o_1\} \sqcup \dots \sqcup \{o_n\} \star \phi')[k/h_1] \dots [k/h_n]\right)$$

We have merged all the statements about h_1, ..., h_n, replaced in the current context ϕ' and the merged abstract object these abstract locations by k, and left a notice in the form of a membrane for additionnal contexts added by the frame rule about the operation which took place.

The materialization follows the same scheme, pictured in figure 2c. Given an entry point to a summary node k—either on the form of a variable $\mathtt{x} \doteq k$ or a location $l \mapsto \{\mathtt{f} : k, \dots\}$—we can rewrite a summary location into a single location (pointed by the entry point) and another summary node, representing the rest of the concrete locations previously present. Indeed, we know that k cannot represent an empty set of locations, and we would like to split it into the exact location l' accessed by our entry point, and the rest k' of the other locations. This operation allows to perform strong updates on these precise values. The materialization of k into l' and k' through

$l.\mathtt{f}$ (or a variable $\mathtt{x}$) transforms an inner formula of the form $k \mapsto \{o\} \star l \mapsto \{\mathtt{f} : k, ...\} \star \phi'$ into the formula $(k \to l' + k' \,|\, (l' \mapsto \{o\} \star k' \mapsto \{o\} \star l \mapsto \{\mathtt{f} : l', ...\} \star \phi')\,[l' \sqcup k'/k])$: the entry point have been replaced by the precise location l' at the cost of replacing every occurence of k by $l' \sqcup k'$. The membrane is for now only partial as there might be uncaught locations in ϕ' or in $\{o\}$. The materialization can only be performed if the entry point is precise: to perform a materialization over $\mathtt{x} \doteq l \sqcup k$ for instance, we would have to first summarize l and k into the same summary node. Note that materialization can always be reversed using a well-chosen summarization.

These two processes of summarization and materialization have been shown on inner formulae. For formulae, we have to merge the new rewriting to the membrane. For instance, let us consider a summarization of l and k to k' on the following formula:

$$\Phi = (k_0 \to k, l_0 \to l \,|\, \mathtt{x} \doteq l \star l \mapsto \{\mathtt{f} : k, \_ : \boxtimes\} \star k \mapsto \{\mathtt{g} : k, \_ : \boxtimes\})$$

We first forget about the membrane and perform the summarization on its inner formula, getting a new inner formula and the partial membrane $\{k \to k', l \to k'\}$; we then compose this partial membrane with the old membrane to get Φ': $\{k \to k', l \to k'\} \circ \{k_0 \to k, l_0 \to l\} = \{k_0 \to k', l_0 \to k'\}$.

$$\Phi' = (k_0 \to k', l_0 \to k' \,|\, \mathtt{x} \doteq k' \star k' \mapsto \{\mathtt{f} : k' \sqcup \boxtimes, \mathtt{g} : k' \sqcup \boxtimes, \_ : \boxtimes\})$$

For the sake of example, let us continue by materializing k' in Φ' through $\mathtt{x}$. As before, we focus on the inner formula, then compose the generated rewriting $k' \to l'' + k''$ to the membrane to get Φ'': $\{k' \to l'' + k''\} \circ \{k_0 \to k', l_0 \to k'\} = \{k_0 \to l'' + k'', l_0 \to l'' + k''\}$.

$$\begin{aligned}\Phi'' = (k_0 \to l'' + k'', l_0 \to l'' + k'' \,|\, \mathtt{x} \doteq l'' \star l'' \mapsto \{\mathtt{f} : l'' \sqcup k'' \sqcup \boxtimes, \mathtt{g} : l'' \sqcup k'' \sqcup \boxtimes, \_ : \boxtimes\} \\ \star\, k'' \mapsto \{\mathtt{f} : l'' \sqcup k'' \sqcup \boxtimes, \mathtt{g} : l'' \sqcup k'' \sqcup \boxtimes, \_ : \boxtimes\})\end{aligned}$$

These transformations are permitted by a relation $\preccurlyeq$ compatible with them: if Φ becomes Φ' through one of these transformations, then $\Phi \preccurlyeq \Phi'$. Intuitively, $\Phi \preccurlyeq \Phi'$ means that Φ is more precise than Φ'. The soundness of these transformations is then implied by the soundness of $\preccurlyeq$. In contrary to usual abstract interpretation, the relation $\preccurlyeq$ is not required to form a lattice, but only to be sound with respect to the concretisation in any context, as shown in Section 6.3. The pre-order $\preccurlyeq$ is defined in Appendix B, but understanding its heavy definition is not needed to follow the rest of this paper.

Materializations and summarizations are the usual manipulations defined in shape analysis, but our formalism allows to define other similar operations. For instance, we could define a filtering operation which partitions locations depending on the values of their fields: the filtering of k relative to field $\mathtt{f}$ and the values l_1 and l_2 in the formula $(k_0 \to k + l_1 + l_2 \,|\, k \to \{\mathtt{f} : l_1 \sqcup l_2, \_ : \boxtimes\} \star \mathtt{x} \doteq k)$ is $(k_0 \to k_1 + k_2 + l_1 + l_2 \,|\, k_1 \to \{\mathtt{f} : l_1, \_ : \boxtimes\} \star k_2 \to \{\mathtt{f} : l_2, \_ : \boxtimes\} \star \mathtt{x} \doteq k_1 \sqcup k_2)$; we have separated the summary node k into two nodes depending on the value of $\mathtt{f}$. To add this operation into the formalism, the relation $\preccurlyeq$ would have to be updated, as well as its correctness proof.

6. A Program Logic for OWhile

Given the abstract domain of formulae defined in the previous sections, we define a program logic for OWhile to reason about these. We shall derive the program logic in a systematic fashion from the concrete semantics, extending an abstraction technique developed by the authors [3] to cover spatial conjunctions and the frame rule. We explain this technique in Section 6.1, then present how to abstract rules in Section 6.2. Section 6.3 presents the changes to accommodate the frame rule.

6.1. Abstract Interpretation of Pretty-big-step Semantics

Motivated by the JSCert operational semantics for JavaScript, we have defined an abstract interpretation framework for semantics written in *pretty-big-step* style [3]. Pretty-big-step semantics [6] is a particular form of big-step semantics where intermediate evaluation steps are brought out explicitly via intermediate terms that mix syntax and semantics. Following a proposal by Schmidt [14] for abstract interpretation of big-step semantics, we have shown how the inference rules in JSCert can each be interpreted over an abstract domain such that the ensuing derivations are correct analyses of the original program. More precisely, we have exactly one abstract rule for each concrete rule, and the correctness proof is simplified to proving that each concrete and abstract rules are related in a one-to-one manner. For example, the abstract versions of the rules Add2 and If (e, s_1, s_2) follow.

$$\textsc{Add2}\ \frac{}{(v_1^\sharp, val^\sharp\ v_2^\sharp), +_2 \Downarrow^\sharp add^\sharp\left(v_1^\sharp, v_2^\sharp\right)} \qquad \textsc{If}(e, s_1, s_2)\ \frac{\Phi, e \Downarrow^\sharp \Phi' \qquad \Phi', if_1\, s_1\, s_2 \Downarrow^\sharp \Phi''}{\Phi, if\, e\, s_1\, s_2 \Downarrow^\sharp \Phi''}$$

However, as explained in [3], this approach implies that the abstract rules can no longer be interpreted inductively. Each rule describes how to build a new Hoare triple $\Phi, t \Downarrow^\sharp \Phi'$ given a semantic relation $\Downarrow_0^\sharp$ but such a triple is not correct by itself. Instead, we must consider all the applicable rules, i.e., rules i that match the term $(t = \mathfrak{l}_i)$ and may be applied according to the semantic context ($cond_i^\sharp(\Phi)$ holds), and merge their results in order to obtain a valid result. Formally, the abstract evaluation relation $\Downarrow^\sharp$ is defined as in Schmidt as the greatest fixed point of the iterator $\mathcal{F}^\sharp$; the relation $\mathcal{F}^\sharp\left(\Downarrow_0^\sharp\right)$ extends the relation $\Downarrow_0^\sharp$ by adding the triples (Φ_σ, t, Φ_r) valid for *all* applicable rules. It uses the function $glue_i^\sharp\left(\Downarrow_0^\sharp\right)$ which computes all triples obtainable from the application of the ith rule:

$$\mathcal{F}^\sharp\left(\Downarrow_0^\sharp\right) = \left\{(\Phi_\sigma, t, \Phi_r) \,\middle|\, \forall i.\ t = \mathfrak{l}_i \Rightarrow cond_i^\sharp(\Phi_\sigma) \Rightarrow (\Phi_\sigma, t, \Phi_r) \in glue_i^\sharp\left(\Downarrow_0^\sharp\right)\right\}$$

Program logics usually include a rule to weaken a results. In our formalism, it would look like this:

$$\textsc{Weaken}\ \frac{\Phi'_\sigma \preccurlyeq \Phi_\sigma \qquad \Phi_\sigma, t \Downarrow^\sharp \Phi_r \qquad \Phi_r \preccurlyeq \Phi'_r}{\Phi'_\sigma, t \Downarrow^\sharp \Phi'_r}$$

In our previous work [3], this rule is encoded in the above-mentioned $glue^\sharp$ function. It allows the analyser to perform some approximations before and after applying rules. The function $glue_i^\sharp$ was then defined as follows, where $apply_i\left(\Downarrow_0^\sharp\right)$ is the set of all triples that can be directly derived from $\Downarrow_0^\sharp$ by applying rule i once.

$$glue_i^\sharp\left(\Downarrow_0^\sharp\right) = \left\{(\Phi_\sigma, t, \Phi_r) \,\middle|\, \exists \Phi'_\sigma, \Phi'_r.\ \Phi_\sigma \preccurlyeq \Phi'_\sigma \wedge \Phi'_r \preccurlyeq \Phi_r \wedge (\Phi'_\sigma, t, \Phi'_r) \in apply_i\left(\Downarrow_0^\sharp\right)\right\}$$

6.2. Abstract Rules

Rules give semantics to expressions, statements, and intermediate terms of the OWhile language. Each concrete rule is translated into exactly one abstract rule. Due to this one-to-one concrete-abstract rule correspondence, abstract and concrete rules share the same names.

In general, the rules fall into four informal categories, measuring the difficulty to abstract them:

- Administrative rules, which push states around; their abstract translation is straightforward.
- Condition rules, which are similar to administrative rules, but with non-trivial side conditions (*cond*). When translating them into the abstract world, their side condition has to be updated ($cond^\sharp$). Such a translation usually do not give further difficulties.
- Error rules, like condition rules, have a non-trivial side condition. Their result is always an error: they require in practise the same amount of work than condition rules to translate.

- Computational rules, where results are produced. The language operations (summing numbers, writing a variable, creating a field, etc.) take place in these rules and their abstract translations are usually more complex.

Figure 5 in Appendix A classifies the different rules of OWhile. As can be seen, very few rules fall into the computational category, which is the category yielding most of the abstraction effort. These categories are arbitrary and debatable as they just serve as a rough estimate on the amount of work needed to build the abstract semantics; for instance the third category of error rules has been added because a lot of rule falls into it, but they require the same amount of work to abstract than the condition rules. For instance, let us consider the two concrete rules for assignments:

$$\textsc{Asgn}(\mathtt{x}, e)\quad \frac{S, e \Downarrow r \qquad r, \mathtt{x} :=_1 \Downarrow r'}{S, \mathtt{x} := e \Downarrow r'} \qquad\qquad \textsc{Asgn1}(\mathtt{x})\quad \frac{}{(S, v), \mathtt{x} :=_1 \Downarrow \mathit{write}\,(S, x, v)}$$

The first rule $\textsc{Asgn}\,(\mathtt{x}, e)$ is an administrative rule: its definition does not depend on the implementation of states and its abstract version is identical. On the other hand, the rule $\textsc{Asgn1}\,(\mathtt{x})$ uses the concrete *write* operation, which does not straightforwardly translate into the abstract world: we do so by exhibiting its footprint. This leads to the following two abstract rules for assignment. In contrary to [3], we only give a local version of the rules: the abstract rules work with the frame rule (see next section). Note how compact the rule $\textsc{Asng1}\,(\mathtt{x})$ is, its context being implicit. Also note that although the concrete rule $\textsc{Asng1}\,(\mathtt{x})$ applies even if $\mathtt{x}$ is not defined in the state, we require it to be present in the abstract formula—eventually with the value $\boxtimes$. This solves the problem stated by Reynolds [13, Section 3.5], as the frame rule can no longer interfere with the resource $\mathtt{x}$.

$$\textsc{Asgn}(\mathtt{x}, e)\quad \frac{\Phi, e \Downarrow^\sharp \Phi' \qquad \Phi', \mathtt{x} :=_1 \Downarrow^\sharp \Phi''}{\Phi, \mathtt{x} := e \Downarrow^\sharp \Phi''} \qquad\qquad \textsc{Asgn1}(\mathtt{x})\quad \frac{}{\left(M \,\middle|\, \mathtt{x} \doteq v_0^\sharp, v^\sharp\right), \mathtt{x} :=_1 \Downarrow^\sharp \left(M \,\middle|\, \mathtt{x} \doteq v^\sharp\right)}$$

The only abstract rule of OWhile whose footprint updates the membrane is the rule NewObj, whose concrete and abstract rules follow. The concrete rule exhibit a fresh concrete location which has no associated reference $l^{\mathit{fresh}(S)}.\mathtt{f}$ in the current state S. In the abstract rule, we create a new location l and declare it as fresh in the membrane: this ensures it to be different from anything present in the context. We also claim that we have write permission over this new location l by adding a memory cell into the formula, leaving its fields undefined as in the concrete rule.

$$\textsc{NewObj}\quad \frac{}{S, \{\} \Downarrow (S, l^{\mathit{fresh}(S)})} \qquad\qquad \textsc{NewObj}\quad \frac{}{(\emptyset \mid \mathit{emp}), \{\} \Downarrow (\nu l \mid l \mapsto \{\_ : \boxtimes\}, l)}$$

6.3. Interfering with the Frame Rule

The version of the frame rule which we use is recalled below.

$$\textsc{Frame}\quad \frac{\Phi, t \Downarrow^\sharp \Phi'}{\Phi_c \circledast \Phi, t \Downarrow^\sharp \Phi_c \circledast \Phi'}$$

To make sense, the abstract semantics has to keep $\mathit{interface}\,(\Phi)$ constant along the derivation. It would otherwise be possible to exhibit a context Φ_c with a different behaviour in both sides. For instance, although $\Phi_1 = (l \to l \mid l \mapsto \{\_ : \boxtimes\})$ and $\Phi_2 = \left(l' \to l \,\middle|\, l \mapsto \{\_ : \boxtimes\}\right)$ represent the same concrete states (they have the same concretisation), in the context of $\Phi_c = \left(k \to l + l' \,\middle|\, l \mapsto \{\_ : \boxtimes\}\right)$,

$\Phi_c \circledast \Phi_1$ has an empty concretisation but not $\Phi_c \circledast \Phi_2$. Because of the frame rule, we can no longer replace a formula Φ by another Φ' just because they represent the same concrete states.

In contrary to usual abstract interpretation, the subset $\preccurlyeq$ of a pre-order which we consider is requested to be sound in any context Φ_c. The fact that this pre-order does not form a lattice—or that it is only a subset of a pre-order—can be surprising; but building a full-fledged lattice can be difficult. Furthermore, we actually do not need such hypotheses to prove the soundness of our approach: we have quotiented the formulae by the equivalent relation built from $\preccurlyeq$, and we can complete the order $\preccurlyeq$ by taking its transitive closure. The lattice usually requested by abstract interpretation only greatly helps in building analysers, but we are here only interested in building an abstract semantics.

$$\Phi_1 \preccurlyeq \Phi_2 \implies \forall \Phi_c.\ \gamma\left(\Phi_c \circledast \Phi_1\right) \subseteq \gamma\left(\Phi_c \circledast \Phi_2\right)$$

Following Schmidt [14], meta rules are not mixed with abstract rules. We thus force the frame rule into the glue between rules by updating the function $\mathit{glue}^\sharp$, as we did when adding the Weaken rule:

$$\mathit{glue}_i^\sharp\left(\Downarrow_0^\sharp\right) = \left\{\left(\Phi_\sigma, t, \Phi_r\right) \,\middle|\, \exists \Phi'_\sigma, \Phi_c, \Phi'_r.\ \Phi_\sigma \preccurlyeq \Phi_c \circledast \Phi'_\sigma \wedge \Phi_c \circledast \Phi'_r \preccurlyeq \Phi_r \wedge \left(\Phi'_\sigma, t, \Phi'_r\right) \in \mathit{apply}_i\left(\Downarrow_0^\sharp\right)\right\}$$

Given a semantic context Φ_σ, we are allowed to approximate it, then split it into the formula Φ'_σ which matches the rule application and a context Φ_c. We then run the rule on Φ'_σ to get Φ'_r, which we consider in the frame Φ_c. We allow a final approximation to get Φ_r. The abstract states are not always formulae, but can be extended formulae (see Section 4.1). Fortunately the operator $\circledast$ and the pre-order $\preccurlyeq$ can be adapted for extended formulae. We do not show the details here for space reasons.

Let us consider the example of $\textsc{Asng1}\left(\mathtt{x}\right)$: it is the rule taking care of the assignment just after the assigned expression has been computed; It takes an extended semantic context as argument, carrying the computed expression. It requires the variable `x` to stand in the input formula:

$$\frac{\textsc{Asgn1}(\mathtt{x})}{\left(M \,\middle|\, \mathtt{x} \doteq v_0^\sharp, v^\sharp\right), \mathtt{x} :=_1 \Downarrow^\sharp \left(M \,\middle|\, \mathtt{x} \doteq v^\sharp\right)}$$

Given a semantic context $\Phi = \left(M \,\middle|\, \phi, v^\sharp\right)$, there are two cases, whether `x` appears in ϕ. If it does not appear, then the rule does not apply. Otherwise ϕ is on the form $\mathtt{x} \doteq v_0^\sharp \star \phi'$: we isolate `x` into $\Phi = \left(M \,\middle|\, \phi'\right) \circledast \left(M' \,\middle|\, \mathtt{x} \doteq v_0^\sharp, v^\sharp\right)$, where $M' = \{h \to h | h \in \mathit{codom}\left(M\right)\}$ is neutral with M. The application of the rule then returns after reapplying the context $\left(M \,\middle|\, \phi'\right) \circledast \left(M' \,\middle|\, \mathtt{x} \doteq v^\sharp\right) = \phi' \circledast \left(M \,\middle|\, \mathtt{x} \doteq v^\sharp\right)$.

Let us now consider the example of $\textsc{Delete1}\left(\mathtt{f}\right)$. This extended rule receives a location and updates its referenced object. Let us see how it behaves when received a summary node k instead of a precise abstract location l by giving it the semantic context $\Phi = \left(k \to k \,\middle|\, k \mapsto \{\mathtt{f} : \mathit{nil}, \_ : \boxtimes\}, k\right)$.

$$\frac{\textsc{Delete1}(\mathtt{f})}{\left(M \,\middle|\, l \mapsto \{o\}, l\right), \mathit{delete}_1\, .\mathtt{f} \Downarrow^\sharp \left(M \,\middle|\, l \mapsto \mathit{remove}^\sharp\left(\mathtt{f}, \{o\}\right)\right)}$$

The abstract operation $\mathit{remove}^\sharp$ writes $\boxtimes$ in the field `f` of the abstract object $\{o\}$. Using a materialization—the carried value being its entry point—we can build the following formula Φ'.

$$\begin{aligned}\Phi \preccurlyeq \Phi' &= \left(k \to l' + k' \,\middle|\, l' \mapsto \{\mathtt{f} : \mathit{nil}, \_ : \boxtimes\} \star k' \mapsto \{\mathtt{f} : \mathit{nil}, \_ : \boxtimes\}, l'\right)\\ &= \left(k \to l' + k' \,\middle|\, k' \mapsto \{\mathtt{f} : \mathit{nil}, \_ : \boxtimes\}\right) \circledast \left(l' \to l' \,\middle|\, l' \mapsto \{\mathtt{f} : \mathit{nil}, \_ : \boxtimes\}, l'\right)\end{aligned}$$

We can now apply the abstract rule $\textsc{Delete1}\left(\mathtt{f}\right)$ on the first part to get the following.

$$\begin{aligned}\Phi'' &= \left(k \to l' + k' \,\middle|\, k' \mapsto \{\mathtt{f} : \mathit{nil}, \_ : \boxtimes\}\right) \circledast \left(l' \to l' \,\middle|\, l' \mapsto \{\_ : \boxtimes\}, l'\right)\\ &= \left(k \to l' + k' \,\middle|\, l' \mapsto \{\_ : \boxtimes\} \star k' \mapsto \{\mathtt{f} : \mathit{nil}, \_ : \boxtimes\}, l'\right)\end{aligned}$$

We can now either continue with this result, which is a strong update over a local location identifier l'. But we might want to diminish the size of the membrane: let us see what happen if we summarize l' and k' back to k: $\Phi'' \preccurlyeq \left(k \to k \,\middle|\, k \mapsto \{\mathtt{f} : \mathit{nil} \sqcup \boxtimes, \_ : \boxtimes\}, k\right)$. We recognize a weak update over k.

6.4. Correctness

The correctness relies on the concretisation γ of formulae. Concretisation of formula is defined through a predicate $\vDash$ shown in Figure 3; this predicate is parametrized by a valuation $\rho : \mathit{LLoc}^\sharp \rightharpoonup \mathit{Loc} \wedge \mathit{KLoc}^\sharp \rightharpoonup \mathcal{P}(\mathit{Loc})$ of abstract locations to concrete locations. The difficult part stands in the concretisation of objects, where abstract values $v^\sharp$ can represent an undefined concrete value if $\boxtimes \sqsubseteq v^\sharp$. We do not show the definition of the concretisation function of objects, but it comes with no surprise. The set R is used to store the reserved variables: $\mathtt{x} \doteq \boxtimes$ states that the variable $\mathtt{x}$ is not in the environment, but it still reserves the resource $\mathtt{x}$ to be sound with the frame rule; this fact is stored by R. The concretisation $\gamma(\Phi)$ of a formula Φ is then a projection of this predicate:

$$(E, H) \in \gamma((M \mid \phi)) \iff \exists \rho, R.\ (E, R, H) \vDash_\rho \phi$$

$$\begin{aligned}
(E, R, H) \vDash_\rho \mathit{emp} &\iff E = R = H = \emptyset \\
(E, R, H) \vDash_\rho \phi_1 \star \phi_2 &\iff \exists E_1, R_1, H_1, E_2, R_2, H_2.\ E = E_1 \uplus E_2 \wedge R = R_1 \uplus R_2 \\
&\qquad \wedge \mathit{dom}(E_1),\ \mathit{dom}(E_2),\ R_1, \text{ and } R_2 \text{ are disjoint pairwise} \\
&\qquad \wedge H = H_1 \uplus H_2 \wedge \mathit{dom}(H_1) \cap \mathit{dom}(H_2) = \emptyset \\
&\qquad \wedge \forall i.\ (E_i, R_i, H_i) \vDash_\rho \phi_i \\
(E, R, H) \vDash_\rho \mathtt{x} \doteq v^\sharp &\iff H = \emptyset \wedge \big(\boxtimes \sqsubseteq v^\sharp \wedge E = \emptyset \wedge R = \{\mathtt{x}\} \\
&\qquad \vee \exists v \in \gamma_\rho\left(v^\sharp\right) \wedge E = \{(\mathtt{x}, v)\} \wedge R = \emptyset\big) \\
(E, R, H) \vDash_\rho l \mapsto \{o\} &\iff E = R = \emptyset \wedge \exists o_0 \in \gamma_\rho(\{o\}).\ H = \{(\rho(l), \mathtt{f}, o_0)\} \\
(E, R, H) \vDash_\rho k \mapsto \{o\} &\iff E = R = \emptyset \wedge \exists (o_i) \in \left(\gamma_\rho(\{o\})\right)^{\rho(k)}.\ H = \bigcup_{l^i \in \rho(k)} \{(\rho(l), \mathtt{f}, o_i)\}
\end{aligned}$$

Figure 3: Definition of the entailment predicate $\vDash_\rho$.

The approach for correctness is the same than in [3]: we require every abstract rule to be locally correct, i.e., their transfer functions (noted for axioms ax and $\mathit{ax}^\sharp$ in the respective concrete and abstract semantics) and side conditions (noted cond and $\mathit{cond}^\sharp$) follow the corresponding concrete rules, taking into account a potential context. The local correctness conditions over transfer functions and side conditions of axiom rules follow—the correctness of other types of rule is very similar. Because of the context in the side condition, it is possible to have a rule whose side condition holds, but whose semantic context does not match the transfer function: this amounts to say that the construction of a derivation can be blocked if some resources are lacking in the original semantic context. We shall not extend on this technical matter. We can infer from these local properties the global correctness.

$$\begin{aligned}
&\forall S, \Phi, \Phi_c.\ S \in \gamma\left(\Phi_c \circledast \Phi\right) \implies \mathit{ax}(S) \in \gamma\left(\Phi_c \circledast \mathit{ax}^\sharp(\Phi)\right) \\
&\forall i, S, \Phi, \Phi_c.\ S \in \gamma\left(\Phi_c \circledast \Phi\right) \implies \mathit{cond}_i(S) \implies \mathit{cond}_i^\sharp(\Phi)
\end{aligned}$$

Property 1 (Global Correctness) *Let t be a term, S and S' be states, and Φ and Φ' formulae. Given the local correctness, if $S \in \gamma(\Phi)$, $S, t \Downarrow S'$, and $\Phi, t \Downarrow^\sharp \Phi'$ then $S' \in \gamma(\Phi')$.*

In other words, abstract derivations can not miss concrete executions: our abstract semantics is sound.

7. Related Work

This work directly follows from [3], which only focussed on abstract interpretation and how to make a Coq development scale up to big operational semantics such as JavaScript's. This previous work came with a Coq development containing some generic analysers, while the current work only focuses at building an abstract domain compatible with the frame rule. There have been works aiming at providing formally verified analysers on languages other than JavaScript such as [11], but these involve a lot of Coq development and we hope to get to a comparatively lighter development for JavaScript. We aim at diving the work of [10] to a fully Coq-verified abstract interpreter.

There have been some work about mixing abstract interpretation and separation logic, such as [9] or [1], but few provide an abstract semantics compatible with the frame rule able to express the abstractions of shape analysis. The lattices constructed by these works are based on a disjunctive completion of a formula order, which can easily explode in size. Our mechanism provides a protection against these explosions through summarizations, with the cost of potentially big imprecision.

The logic of [8] is very close to this work; their domain is a disjunctive completion of formulae separated at the field level of objects as we did. Locations and fields are both abstracted by either singletons or summary nodes. However, the frame rule is not mentioned, which removes the need of membranes. They carry a set of formulae storing information about the respective inclusion of the concretisations of summary nodes. Their domain is ordered and equipped with a join and a widening operator with an algorithm compatible with the concretisation function.

The same authors previously developped [5] based on separation logic; this work focusses on inductively defined shapes and is able to express and analyse complex structures such as red-black trees. To increase the efficiency of the analyse, they developed a way to change the point of view of these shapes: for instance, a doubly linked list can be defined either by following `next` or `previous` fields. As with this work, the order relation, are defined through an algorithm compatible with the concretisation function, and they similarly defined joins and widenings by an algorithm.

Inductively defined structures have also been examined in [4], which uses abduction to determine the weakest precondition wielding safety or termination of the analysed program. They provide heuristics to infer how to generalize predicates, as well as a running tool. However, their heuristics rely on the syntax of their toy language (comparable to OWhile): scaling such an analyser up to JavaScript might require to look for much complex heuristics, and we think that an approach guided directly by the language semantics can reduce the amount of work to get a certified analyser.

8. Conclusion and Future Works

We have presented a program logic for JavaScript heaps, based on separation logic and integrating ideas from shape analysis. We have expressed this logic within a framework of certified abstract interpretation. The goal is to scale up the logic to the size of JavaScript's semantics, resulting in an abstract semantics for JavaScript of reasonable size, and eventually certified analysers for JavaScript. The particular problem addressed in this paper is that of integrating the frame rule to existing abstract interpretation framework, which is known to be a difficult problem.

Our approach is precise enough to get interesting results on real-world programs, whilst being simple enough to be able to scale it up to a certified abstract semantics of JavaScript. This work is part of a larger project which aims at building certified static analyses based on the JSCert [2] formal semantics of JavaScript.

We have focussed on how to build an abstract semantics, and not on how to build analysers. The abstract domains that arise from our logic are less structured than usual abstract domains, which means that an analyser will be less guided by the abstract interpretation framework. There is thus room for further research into the construction of efficient join and widening operators for abstract domains combining separation and summarization. Furthermore, the frame rule allows to individually analyse functions or recurrent programs; it is thus natural to look for strategies to choose and separately analyse these programs. We have observed that analysing a program without starting from the right resources can lead to big approximations, or to the inability to analyse. Techniques like bi-abduction may be relevant to build efficient oracles for our certified analysers.

Bibliographie

[1] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, page 178–192. Springer, 2007.

[2] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. Jscert: Certified javascript. `http://jscert.org/`, 2012.

[3] M. Bodin, T. Jensen, and A. Schmitt. Certified abstract interpretation with pretty-big-step semantics. In *CPP*, page 29–40. ACM, 2015.

[4] J. Brotherston and N. Gorogiannis. Cyclic abduction of inductively defined safety and termination preconditions. In M. Müller-Olm and H. Seidl, editors, *Static Analysis*, volume 8723 of *Lecture Notes in Computer Science*, page 68–84. Springer International Publishing, 2014.

[5] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *ACM SIGPLAN Notices*, volume 43, page 247–260. ACM, 2008.

[6] A. Charguéraud. Pretty-big-step semantics. In *ESOP*, page 41–60. Springer, 2013.

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

[8] A. Cox, B.-Y. E. Chang, and X. Rival. Automatic analysis of open objects in dynamic language programs. In *Static Analysis*, page 134–150. Springer, 2014.

[9] D. Distefano, P. W. O'hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, page 287–302. Springer, 2006.

[10] P. Gardner, S. Maffeis, and G. Smith. Towards a program logic for javascript. *ACM SIGPLAN Notices*, 47(1):31–44, 2012.

[11] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A formally-verified c static analyzer. In *POPL*, page 247–259. ACM, 2015.

[12] M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in hoare logics. In *LICS*, page 137–146. IEEE, 2006.

[13] J. C. Reynolds. An introduction to separation logic. *Engineering Methods and Tools for Software Safety and Security*, page 285–310, 2008.

[14] D. A. Schmidt. Natural-semantics-based abstract interpretation (preliminary version). In *SAS*, page 1–18. Springer LNCS vol. 983, 1995.

A. Concrete Semantics

Figure 4 presents the complete syntax of OWHILE, which includes extended terms. These extended terms carry intermediary results, and are thus associated with specific kinds of extended semantic contexts, carrying additional values. The concrete rules follow. These rules makes use of functions such as *read* and *write* which perform some semantic manipulation; their semantics is usual and we do not explicit them. Section 6.2 categorizes rules into four different kinds of categories; Figure 5 shows

$$\begin{array}{llll} s ::= \mathit{skip} & \mid s_1;\ s_2 & \mid \mathit{if}\, e\, s_1\, s_2 \\ \quad \mid \mathit{while}\, e\, s & \mid \mathit{throw} & \mid \mathtt{x} := e \\ \quad \mid e_1\,\mathtt{.f} := e_2 & \mid \mathit{delete}\, e\,\mathtt{.f} & \end{array}$$

(a) Statements

$$\begin{array}{lllll} e ::= n \in \mathbb{Z} & \mid ? & \mid \mathtt{x} \in \mathit{Var} & \mid \mathit{nil} \\ \quad \mid \{\} & \mid e\mathtt{.f} & \mid \mathtt{f}\ \mathtt{in}\, e & \mid \neg\ e \\ \quad \mid =\, e_1\ e_2 & \mid \bowtie e_1\ e_2 & (\bowtie \in \{>, +, -\}) & \end{array}$$

(b) Expressions

$$\begin{array}{llll} s_e ::= ;_1\, s_2 & \mid \mathit{if}_1\, s_1\, s_2 & \mid \mathit{while}_1\, e\, s \\ \quad \mid \mathit{while}_2\, e\, s & \mid \mathtt{x} :=_1 & \mid \mathtt{.f} :=_1 e_2 \\ \quad \mid \mathtt{.f} :=_2 & \mid \mathit{delete}_1\ \mathtt{.f} & \end{array}$$

(c) Extended statements

$$\begin{array}{llll} e_e ::= \mathtt{.f} & \mid \mathtt{f}\ \mathtt{in}_1 & \mid \bowtie_1 e_2 \\ \quad \mid \bowtie_2 & \mid \neg_1 & \mid =_1 e_2 \\ \quad \mid =_2 & & \end{array}$$

(d) Extended expressions

Figure 4: Complete syntax of the OWhile language

where all the rules of this semantics fall. As can be seen, the categories are more or less identical in size, with the notable exception of the computational rules, which are the most difficult to abstract.

$$\textsc{AbortExtExpr}(e_e)\quad \frac{}{\sigma, e_e \Downarrow \mathit{Err}}\ \ \sigma = \mathit{Err} \qquad\qquad \textsc{AbortExtStat}(s_e)\quad \frac{}{\sigma, s_e \Downarrow \mathit{Err}}\ \ \sigma = \mathit{Err}$$

$$\textsc{Cst}(n)\quad \frac{}{S, n \Downarrow (S, n)} \qquad \textsc{Random}(n)\quad \frac{}{S, ? \Downarrow (S, n)} \qquad \textsc{Var}(\mathtt{x})\quad \frac{}{S, \mathtt{x} \Downarrow (S, \mathit{read}(S, \mathtt{x}))}\ \ \mathtt{x} \in \mathit{dom}(S) \qquad \textsc{VarUndef}(\mathtt{x})\quad \frac{}{S, \mathtt{x} \Downarrow \mathit{Err}}\ \ \mathtt{x} \notin \mathit{dom}(S)$$

$$\textsc{Nil}\quad \frac{}{S, \mathit{nil} \Downarrow (S, l^0)} \qquad \textsc{NewObj}\quad \frac{}{S, \{\} \Downarrow (S, l^{\mathit{fresh}(S)})} \qquad \textsc{Property}(e, \mathtt{f})\quad \frac{S, e \Downarrow r \qquad r, \mathtt{.f} \Downarrow r'}{S, e\mathtt{.f} \Downarrow r'}$$

$$\textsc{Property1}(\mathtt{f})\quad \frac{}{(S, l^i), \mathtt{.f} \Downarrow (S, \mathit{read}(S, l^i\,\mathtt{.f}))}\ \ l^i\,\mathtt{.f} \in \mathit{dom}(S) \qquad \textsc{Property1NoLoc}(\mathtt{f})\quad \frac{}{(S, n), \mathtt{.f} \Downarrow \mathit{Err}}$$

$$\textsc{Property1Undef}(\mathtt{f})\quad \frac{}{(S, l^i), \mathtt{.f} \Downarrow \mathit{Err}}\ \ l^i\,\mathtt{.f} \notin \mathit{dom}(S) \qquad \textsc{In}(\mathtt{f}, e)\quad \frac{S, e \Downarrow r \qquad r, \mathtt{f}\ \mathtt{in}_1 \Downarrow r'}{S, \mathtt{e}\ \mathtt{in}\, f \Downarrow r'}$$

$$\textsc{In1True}(\mathtt{f})\quad \frac{}{(S, l^i), \mathtt{f}\ \mathtt{in}_1 \Downarrow (S, 1)}\ \ l^i\,\mathtt{.f} \in \mathit{dom}(S) \qquad \textsc{In1NoLoc}(\mathtt{f})\quad \frac{}{(S, n), \mathtt{f}\ \mathtt{in}_1 \Downarrow \mathit{Err}} \qquad \textsc{In1False}(\mathtt{f})\quad \frac{}{(S, l^i), \mathtt{f}\ \mathtt{in}_1 \Downarrow (S, 0)}\ \ l^i\,\mathtt{.f} \notin \mathit{dom}(S)$$

$$\textsc{Op}(\bowtie, e_1, e_2)\;\frac{S, e_1 \Downarrow r \qquad r, \bowtie_1 e_2 \Downarrow r'}{S, \bowtie e_1\ e_2 \Downarrow r'} \qquad \textsc{Op1}(\bowtie, e_2)\;\frac{S, e_2 \Downarrow r \qquad (n_1, r), \bowtie_2 \Downarrow r'}{(S, n_1), \bowtie_1\ e_2 \Downarrow r'} \qquad \textsc{Op1Error}(\bowtie, e_2)\;\frac{}{(S, l^i), \bowtie_1\ e_2 \Downarrow \mathit{Err}}$$

$$\textsc{Op2Error}(\bowtie)\;\frac{}{(n_1, (S, l^j)), \bowtie_2 \Downarrow \mathit{Err}} \qquad \textsc{Add2}\;\frac{}{(n_1, (S, n_2)), +_2 \Downarrow (S, n_1 + n_2)} \qquad \textsc{Sub2}\;\frac{}{(n_1, (S, n_2)), -_2 \Downarrow (S, n_1 - n_2)}$$

$$\textsc{Greater2Greater}\;\frac{}{(n_1, (S, n_2)), >_2 \Downarrow (S, 1)}\; n_1 > n_2 \qquad \textsc{Greater2LesserEq}\;\frac{}{(n_1, (S, n_2)), >_2 \Downarrow (S, 0)}\; n_1 \leq n_2 \qquad \textsc{Not}(e)\;\frac{S, e \Downarrow r \qquad r, \neg_1 \Downarrow r'}{S, \neg\ e \Downarrow r'}$$

$$\textsc{Not1True}\;\frac{}{(S, n), \neg_1 \Downarrow (S, 0)}\; n \neq 0 \qquad \textsc{Not1False}\;\frac{}{(S, n), \neg_1 \Downarrow (S, 1)}\; n = 0 \qquad \textsc{Not1Error}\;\frac{}{(S, l), \neg_1 \Downarrow \mathit{Err}}$$

$$\textsc{Eq}(e_1, e_2)\;\frac{S, e_1 \Downarrow r \qquad r, =_1\ e_2 \Downarrow r'}{S, =\ e_1\ e_2 \Downarrow r'} \qquad \textsc{Eq1}(e_2)\;\frac{S, e_2 \Downarrow r \qquad (v_1, r), =_2 \Downarrow r'}{(S, v_1), =_1\ e_2 \Downarrow r'} \qquad \textsc{Eq2BasicValEq}\;\frac{}{(n_1, (S, n_2)), =_2 \Downarrow (S, 1)}\; n_1 = n_2$$

$$\textsc{Eq2BasicValNeq}\;\frac{}{(n_1, (S, n_2)), =_2 \Downarrow (S, 0)}\; n_1 \neq n_2 \qquad \textsc{Eq2LocEq}\;\frac{}{(l^i, (S, l^j)), =_2 \Downarrow (S, 1)}\; l^i = l^j$$

$$\textsc{Eq2LocNeq}\;\frac{}{(l^i, (S, l^j)), =_2 \Downarrow (S, 0)}\; l^i \neq l^j \qquad \textsc{Eq2MistypeBasicValLoc}\;\frac{}{(n_1, (S, l^j)), =_2 \Downarrow \mathit{Err}} \qquad \textsc{Eq2MistypeLocBasicVal}\;\frac{}{(l^i, (S, n_2)), =_2 \Downarrow \mathit{Err}}$$

$$\textsc{Skip}\;\frac{}{S, \mathit{skip} \Downarrow S} \qquad \textsc{Seq}(s_1, s_2)\;\frac{S, s_1 \Downarrow r \qquad r, ;_1\ s_2 \Downarrow r'}{S, s_1;\ s_2 \Downarrow r'} \qquad \textsc{Seq1}(s_2)\;\frac{S, s_2 \Downarrow r}{S, ;_1\ s_2 \Downarrow r} \qquad \textsc{Throw}\;\frac{}{S, \mathit{throw} \Downarrow \mathit{Err}}$$

$$\textsc{If}(e, s_1, s_2)\;\frac{S, e \Downarrow r \qquad r, \mathit{if}_1\, s_1\, s_2 \Downarrow r'}{S, \mathit{if}\, e\, s_1\, s_2 \Downarrow r'} \qquad \textsc{If1True}(s_1, s_2)\;\frac{S, s_1 \Downarrow r}{(S, n), \mathit{if}_1\, s_1\, s_2 \Downarrow r}\; n \neq 0 \qquad \textsc{If1False}(s_1, s_2)\;\frac{S, s_2 \Downarrow r}{(S, n), \mathit{if}_1\, s_1\, s_2 \Downarrow r}\; n = 0$$

$$\textsc{If1Error}(s_1, s_2)\;\frac{}{(S, l^i), \mathit{if}_1\, s_1\, s_2 \Downarrow \mathit{Err}} \qquad \textsc{While}(e, s)\;\frac{S, \mathit{while}_1\, e\, s \Downarrow r}{S, \mathit{while}\, e\, s \Downarrow r} \qquad \textsc{While1}(e, s)\;\frac{S, e \Downarrow r \qquad r, \mathit{while}_1\, e\, s \Downarrow r'}{S, \mathit{while}_1\, e\, s \Downarrow r'}$$

$$\textsc{While2True}(e, s)\;\frac{S, s \Downarrow r \qquad r, \mathit{while}_1\, e\, s \Downarrow r'}{(S, n), \mathit{while}_2\, e\, s \Downarrow r'}\; n \neq 0 \qquad \textsc{While2False}(e, s)\;\frac{}{(S, n), \mathit{while}_2\, e\, s \Downarrow S}\; n = 0 \qquad \textsc{While2Error}(e, s)\;\frac{}{(S, l^i), \mathit{while}_2\, e\, s \Downarrow \mathit{Err}}$$

Figure 5: Categories of rules

Administrative	Conditions	Error	Computational
$\mathit{Property}\,(e, \mathtt{f})$	$\mathit{AbortExtExpr}\,(e_e)$	$\mathit{VarUndef}\,(\mathtt{x})$	$\mathit{Cst}\,(n)$
$\mathit{In}\,(\mathtt{f}, e)$	$\mathit{AbortExtStat}\,(s_e)$	$\mathit{Property1NoLoc}\,(\mathtt{f})$	$\mathit{Random}\,(n)$
$\mathit{Op}\,(\bowtie, e_1, e_2)$	$\mathit{In1True}\,(\mathtt{f})$	$\mathit{Property1Undef}\,(\mathtt{f})$	$\mathit{Var}\,(\mathtt{x})$
$\mathit{Op1}\,(\bowtie, e_2)$	$\mathit{In1False}\,(\mathtt{f})$	$\mathit{In1NoLoc}\,(\mathtt{f})$	Nil
$\mathit{Not}\,(e)$	$\mathit{Greater2Greater}$	$\mathit{Op1Error}\,(\bowtie, e_2)$	NewObj
$\mathit{Eq}\,(e_1, e_2)$	$\mathit{Greater2LesserEq}$	$\mathit{Op2Error}\,(\bowtie)$	$\mathit{Property1}\,(\mathtt{f})$
$\mathit{Eq1}\,(e_2)$	$\mathit{Not1True}$	$\mathit{Not1Error}$	$\mathit{Add2}$
Skip	$\mathit{Not1False}$	$\mathit{Eq2MistypeBasicValLoc}$	$\mathit{Sub2}$
$\mathit{Seq}\,(s_1, s_2)$	$\mathit{Eq2BasicValEq}$	$\mathit{Eq2MistypeLocBasicVal}$	$\mathit{Asgn1}\,(\mathtt{x})$
$\mathit{Seq1}\,(s_2)$	$\mathit{Eq2BasicValNeq}$	$\mathit{If1Error}\,(s_1, s_2)$	$\mathit{PropertyAsgn2}\,(\mathtt{f})$
$\mathit{If}\,(e, s_1, s_2)$	$\mathit{Eq2LocEq}$	$\mathit{While2Error}\,(e, s)$	$\mathit{Delete1}\,(\mathtt{f})$
$\mathit{While}\,(e, s)$	$\mathit{Eq2LocNeq}$	Throw	
$\mathit{While1}\,(e, s)$	$\mathit{If1True}\,(s_1, s_2)$	$\mathit{PropertyAsgn1Error}\,(\mathtt{f}, e_2)$	
$\mathit{Asgn}\,(\mathtt{x}, e)$	$\mathit{If1False}\,(s_1, s_2)$	$\mathit{PropertyAsgn1Nil}\,(\mathtt{f}, e_2)$	
$\mathit{PropertyAsgn}\,(e_1, \mathtt{f}, e_2)$	$\mathit{While2True}\,(e, s)$	$\mathit{Delete1Error}\,(\mathtt{f})$	
$\mathit{Delete}\,(e, \mathtt{f})$	$\mathit{While2False}\,(e, s)$		
	$\mathit{PropertyAsgn1}\,(\mathtt{f}, e_2)$		
	$\mathit{Delete1Nil}\,(\mathtt{f})$		

$$\textsc{Asgn}(\mathtt{x}, e)\quad \frac{S, e \Downarrow r \qquad r, \mathtt{x} :=_1 \Downarrow r'}{S, \mathtt{x} := e \Downarrow r'}$$

$$\textsc{Asgn1}(\mathtt{x})\quad \frac{}{(S, v), \mathtt{x} :=_1 \Downarrow \mathit{write}\,(S, x, v)}$$

$$\textsc{PropertyAsgn}(e_1, \mathtt{f}, e_2)\quad \frac{S, e_1 \Downarrow r \qquad r, .\mathtt{f} :=_1 e_2 \Downarrow r'}{S, e_1 .\mathtt{f} := e_2 \Downarrow r'}$$

$$\textsc{PropertyAsgn1}(\mathtt{f}, e_2)\quad \frac{S, e_2 \Downarrow r \qquad (l^i, r), .\mathtt{f} :=_2 \Downarrow r'}{(S, l^i), .\mathtt{f} :=_1 e_2 \Downarrow r'}$$

$$\textsc{PropertyAsgn1Error}(\mathtt{f}, e_2)\quad \frac{}{(S, n), .\mathtt{f} :=_1 e_2 \Downarrow \mathit{Err}}$$

$$\textsc{PropertyAsgn1Nil}(\mathtt{f}, e_2)\quad \frac{}{(S, l^0), .\mathtt{f} :=_1 e_2 \Downarrow \mathit{Err}}$$

$$\textsc{PropertyAsgn2}(\mathtt{f})\quad \frac{}{(l^i, (S, v)), .\mathtt{f} :=_2 \Downarrow \mathit{write}\,(S, l^i .\mathtt{f}, v)}$$

$$\textsc{Delete}(e, \mathtt{f})\quad \frac{S, e \Downarrow r \qquad r, \mathit{delete}_1\, .\mathtt{f} \Downarrow r'}{S, \mathit{delete}\, e .\mathtt{f} \Downarrow r'}$$

$$\textsc{Delete1}(\mathtt{f})\quad \frac{}{(S, l^i), \mathit{delete}_1\, .\mathtt{f} \Downarrow \mathit{remove}\,(S, l^i .\mathtt{f})}$$

$$\textsc{Delete1Error}(\mathtt{f})\quad \frac{}{(S, n), \mathit{delete}_1\, .\mathtt{f} \Downarrow \mathit{Err}}$$

$$\textsc{Delete1Nil}(\mathtt{f})\quad \frac{}{(S, l^0), \mathit{delete}_1\, .\mathtt{f} \Downarrow \mathit{Err}}$$

B. Pre-order over Formulae

As said in Section 6.3, we are looking for a relation $\preccurlyeq$ which respects the frame property:

$$\Phi_1 \preccurlyeq \Phi_2 \implies \forall \Phi_c .\ \gamma\,(\Phi_c \circledast \Phi_1) \subseteq \gamma\,(\Phi_c \circledast \Phi_2)$$

The empty relation would be correct; however, it would not be useful as it allows no transformation to the formulae: we would like to account for materializations and summarizations. We define our relation $\preccurlyeq$ by an algorithm taking two abstract heaps $\Phi_1 = (M_1 \mid \phi_1)$ and $\Phi_2 = (M_2 \mid \phi_2)$ and returning *true* if it successfully proved the above property. However, the inclusion of the concretisations does not necessarily yield the pre-order relation. We have not focused on the efficiency of this algorithm.

B.1. Example

To illustrate our comparison algorithm, let us consider the following two formulae to get $\Phi_1 \preccurlyeq \Phi_2$.

$$\begin{aligned}\Phi_1 &= \left(k_0 \to k, l_0 \to l \,\middle|\, \mathtt{x} \doteq l \sqcup k \star k \mapsto \{\mathtt{f} : \bot, \mathtt{g} : l, \_ : \boxtimes\} \star l \mapsto \{\mathtt{f} : l, \_ : \boxtimes\}\right)\\ \Phi_2 &= \left(k_0 \to k' + l', l_0 \to l' + k' \,\middle|\, \mathtt{x} \doteq l' \star k' \mapsto \{\_ : \boxtimes\} \star l' \mapsto \{\mathtt{f} : l' \sqcup k' \sqcup \mathit{nil}, \_ : \boxtimes\}\right)\end{aligned}$$

Our algorithm starts by splitting Φ_1 to remove disjunctions in the values of variables and abstract locations l. In the example $\mathtt{x} \doteq l \sqcup k$ can be split into $\mathtt{x} \doteq l$ and $\mathtt{x} \doteq k$ to get the two formulae $\Phi_{1,a}$ and $\Phi_{1,b}$ below. This is sound as the disjunction $\sqcup$ we chose for values does not loose precision: $\forall \Phi_c.\ \gamma\left(\Phi_c \mathbin{\boxed{\star}} \Phi_1\right) = \gamma\left(\Phi_c \mathbin{\boxed{\star}} \Phi_{1,a}\right) \cup \gamma\left(\Phi_c \mathbin{\boxed{\star}} \Phi_{1,b}\right)$.

$$\begin{aligned}\Phi_{1,a} &= \left(k_0 \to k, l_0 \to l \,\middle|\, \mathtt{x} \doteq l \star k \mapsto \{\mathtt{f} : \bot, \mathtt{g} : l, \_ : \boxtimes\} \star l \mapsto \{\mathtt{f} : l \sqcup k, \_ : \boxtimes\}\right)\\ \Phi_{1,b} &= \left(k_0 \to k, l_0 \to l \,\middle|\, \mathtt{x} \doteq k \star k \mapsto \{\mathtt{f} : \bot, \mathtt{g} : l, \_ : \boxtimes\} \star l \mapsto \{\mathtt{f} : l \sqcup k, \_ : \boxtimes\}\right)\end{aligned}$$

Let us focus on $\Phi_{1,a}$. We look for a function ψ translating identifiers of the formula $\Phi_{1,a}$ to those of Φ_2 in order to match $\Phi_{1,a}$ with Φ_2. Here the function ψ mapping l to l' and k to k' is enough. Applying the rewriting ψ over $\Phi_{1,a}$ follows the same rules as the α-renaming and the summarization presented in Sections 4.2 and 5—which is why summarizations are compatible with this pre-order:

$$\psi\left(\Phi_{1,a}\right) = \left(k_0 \to k', l_0 \to l' \,\middle|\, \mathtt{x} \doteq l' \star k' \mapsto \{\mathtt{f} : \bot, \mathtt{g} : l, \_ : \boxtimes\} \star l' \mapsto \{\mathtt{f} : l' \sqcup k', \_ : \boxtimes\}\right)$$

The summary node k has an incoherence: every concrete location represented by k is supposed to have a field $\mathtt{f}$ represented by $\bot$, which is not possible: the only possible concretisation of k is the empty set. We can thus safely remove k from the formula:

$$\Phi'_{1,a} = \left(k_0 \to \emptyset, l_0 \to l' \,\middle|\, \mathtt{x} \doteq l' \star l' \mapsto \{\mathtt{f} : l', \_ : \boxtimes\}\right)$$

At this point, we compare the inner scope of $\Phi'_{1,a}$ with the one of Φ_2: the latter has an additional summary node k'. We can easily ignore such a summary node as it can represent an empty set of concrete locations; we thus rewrite k' into $\emptyset$ in Φ_2 to get $\Phi'_2 = \left(k_0 \to l', l_0 \to l' \,\middle|\, \mathtt{x} \doteq l' \star l' \mapsto \{\mathtt{f} : l' \sqcup \mathit{nil}, \_ : \boxtimes\}\right)$. We now compare the values in the membrane, the variables, and the objects; which we can check are greater in Φ'_2 for any corresponding in $\Phi'_{1,a}$. We have successfully found a ψ leading to the conclusion $\forall \Phi_c.\ \gamma\left(\Phi_c \mathbin{\boxed{\star}} \Phi_{1,a}\right) \subseteq \gamma\left(\Phi_c \mathbin{\boxed{\star}} \Phi_2\right)$.

In the case of $\Phi_{1,b}$ we can perform a materialization, as the have an entry point to a summary node $\mathtt{x} \doteq k$. But this generates a subformula $l' \mapsto \{\mathtt{f} : \bot, \mathtt{g} : l, \_ : \boxtimes\}$, which has an empty concretisation as the reference $l'.\mathtt{f}$ should be $\bot$. We can conclude that $\Phi_{1,b}$ has an empty concretisation, and thus $\Phi_{1,b} \preccurlyeq \Phi_2$. Overall, as both $\Phi_{1,a}$ and $\Phi_{1,b}$ are below Φ_2 by the pre-order $\preccurlyeq$, we have $\Phi_1 \preccurlyeq \Phi_2$.

B.2. General Procedure

As we want to be able to perform summarizations and materializations, our algorithm has to take these operations into account. Algorithm 1 shows its pseudo-code; it proceeds through two nested loops. The first one splits the first formula Φ_1 into more precise formulae Φ'_1 through materializations. The second tries to match the more precise Φ'_1 with Φ_2 by performing summarizations. We then look for incoherences, then compare the variables and the objects of the two formulae. The two loops could be removed without causing problems for the final theorem; however they are necessary to make materializations and summarizations hold. Formulae with different interfaces are rejected, even if they have the same concretisation in every context: the relation $\preccurlyeq$ does not have to be complete.

The first step consists in splitting the formula Φ_1 to remove the presence of join operators $\sqcup$ in the abstract values; as for the formulae $\Phi_{1,a}$ and $\Phi_{1,b}$ of the example above. Because of the way we

```
Data: Two abstract heaps $\Phi_1$ and $\Phi_2$.
Result: true iff $\Phi_1 \preccurlyeq \Phi_2$.
if $interface(\Phi_1) \neq interface(\Phi_2)$ then
    return false;   // Formulae with different interfaces are immediately rejected.
end
for $\Phi'_1$ be a split/materialization of $\Phi_1$ do                    // Universal branching
    for $\psi$ well-formed do                                                // Existential choice
        $\Phi''_1 \leftarrow \psi(\Phi'_1)$;
        Remove incoherent summary nodes from $\Phi''_1$;
        Let $\Phi'_2$ be $\Phi_2$ where every summary node not present in $\Phi''_1$ has been removed;
        if $\Phi''_1$ incoherent then
            // The current $\psi$ is enough to show that $\Phi'_1 \preccurlyeq \Phi_2$.
            Continue on the next split $\Phi'_1$ of $\Phi_1$;
        end
        if $\Phi'_2$ incoherent then
            Try another $\psi$;
        end
        for $h \rightarrow h_1 + ... + h_n$ present in $\Phi'_2$ and $h \rightarrow h'_1 + ... + h'_m$ in $\Phi''_1$ do
            if $\{h'_1, ..., h'_m\} \not\sqsubseteq \{h_1, ..., h_n\}$ then
                Try another $\psi$;
            end
        end
        for $\mathtt{x} \doteq v^\sharp_2$ present in $\Phi'_2$ and $\mathtt{x} \doteq v^\sharp_1$ present in $\Phi''_1$ do
            if $v^\sharp_1 \not\sqsubseteq v^\sharp_2$ then
                Try another $\psi$;
            end
        end
        for $h \mapsto \{o_2\}$ present in $\Phi'_2$ and $h \mapsto \{o_1\}$ present in $\Phi''_1$ do
            if $\{o_1\} \not\sqsubseteq \{o_2\}$ then
                Try another $\psi$;
            end
        end
        // The chosen $\psi$ succeeded in matching $\Phi'_1$ with $\Phi'_2$.
        Continue on the next split $\Phi'_1$ of $\Phi$;
    end
    // No $\psi$ succeeded in matching $\Phi'_1$ with $\Phi'_2$.
    return false;
end
// $\Phi_2$ correctly captures all the behaviours of $\Phi_1$.
return true;
```

Algorithm 1: Algorithm comparing formulae.

defined abstract values, we can split values of the form $n^\sharp \sqcup nil^? \sqcup l_1 \sqcup ... \sqcup l_n \sqcup d$ to their different components without missing any concrete value. This splitting can only be done if the considered abstract value represents only one concrete value; which is neither the case of the default abstract values of objects[3], as in the object $\{\_ : nil \sqcup \boxtimes\}$, nor of a field value pointed by a summary node k.

The splitting part also takes into account the materializations in Φ_1. This explodes the number of considered formulae Φ_1' as this amounts to look for all possible aliases of Φ_1. These materializations are performed for each entry point to a summary node and performed as described in Section 5. Unfortunately, this part can loop. For instance, consider the inner formula $\mathtt{x} \doteq k \star k \mapsto \{\mathtt{f} : k, \_ : \top\}$: we can materialize $\mathtt{x}$ to get $\mathtt{x} \doteq l_1 \star l_1 \mapsto \{\mathtt{f} : l_1 \sqcup k, \_ : \top\} \star k \mapsto \{\mathtt{f} : l_1 \sqcup k, \_ : \top\}$, which splits into $\mathtt{x} \doteq l_1 \star l_1 \mapsto \{\mathtt{f} : k, \_ : \top\} \star k \mapsto \{\mathtt{f} : l_1 \sqcup k, \_ : \top\}$, which keeps materializing/splitting. But do we need to split it indefinitely? The only goal of this algorithm is to compare the current formula with Φ_2: unfolding up to the size of Φ_2 is enough. Unfoldings can significantly increase the complexity of this algorithm; however, its correction still holds if we limit this unfolding to a given depth—at the cost of the transitivity of $\preccurlyeq$. This is the reason we say that $\preccurlyeq$ is only the subset of a pre-order: it is possible to compute it completely and get a complete pre-order, with the cost of efficiency.

This first step makes us consider every possible shape structure. The next step is run on each of these exploded formulae Φ_1' and succeeds if it succeeded for all instances (universal search). This step accepts a formula Φ_1' if it can prove that $\forall \Phi_c.\ \gamma(\Phi_c \circledast \Phi_1') \subseteq \gamma(\Phi_c \circledast \Phi_2)$. As $\forall \Phi_c.\ \gamma(\Phi_c \circledast \Phi_1) = \bigcup \gamma(\Phi_c \circledast \Phi_1')$, an acceptance yields the requested property $\forall \Phi_c.\ \gamma(\Phi_c \circledast \Phi_1) \subseteq \gamma(\Phi_c \circledast \Phi_2)$.

For each of these exploded formulae Φ_1', we look for two functions $\psi_l : LLoc^\sharp \rightharpoonup \left(LLoc^\sharp \cup KLoc^\sharp\right)$ and $\psi_k : KLoc^\sharp \rightharpoonup KLoc^\sharp$ translating identifiers of the formula Φ_1' to those of Φ_2; which we shall note both ψ. The goal is to find a translation matching Φ_1' with Φ_2. Some restrictions applies to ψ:

- Only identifiers present in Φ_1' and Φ_2 can appear in ψ. This makes this step a finite search.
- For an abstract location l, at most one l_0 reaches l: $\forall l, l_1, l_2.\ \psi(l_1) = \psi(l_2) = l \implies l_1 = l_2$.

We rewrite Φ_1' into the formula $\Phi_1'' = \psi(\Phi_1') = (\psi \mid emp) \circledast \Phi_1'$ where ψ has been identified with its graph. If two or more abstract locations are mapped through ψ to the same summary node k, then all their objects are merged; this step fails if one abstract location lacks a memory property in Φ_1'. For instance, if $\psi(l) = \psi(k) = k'$ and that $\Phi_1' = (k_0 \to k, l_0 \to l \mid l \mapsto \{o_1\} \star k \mapsto \{o_2\})$, then $\Phi_1'' = (k_0 \to k', l_0 \to k' \mid k' \mapsto \{o_1'\} \sqcup \{o_2'\})$ where $\{o_1'\}$ and $\{o_2'\}$ received the renaming process of ψ. However, the same choice of ψ fails if only $l \mapsto \{o\}$ appears in Φ_1'.

We execute the last step for all those ψ and return *true* if at least one of these make the following step returns *true* (existential choice). The research of a working ψ can probably be performed more efficiently, by only considering the possible ones in an iterative algorithm similar to the one of [5]. The last step consists at comparing the exploded and renamed formula Φ_1'' to Φ_2', by checking for incoherences, then comparing the membranes, variables, and objects defined.

We look for incoherences into the formula Φ_1''. If any is found, then the concretisation of Φ_1'' is empty, and we immediately states that $\Phi_1'' \preccurlyeq \Phi_2''$ by returning *true*. There are two cases of incoherences. First, spatial incoherences, when a location l is referred several times in the formula in a left-hand side; such as in $l \mapsto \{o\} \star l \mapsto \{o\}$. Even if the two objects $\{o\}$ are identical, both sides of the operator $\star$ refer to the same region of space, which is forbidden. Second, when a variable or the field $\mathtt{f}$ of an abstract location l has $\bot$ as a value, which has an empty concretisation. Note that $\bot$ found in summary nodes does not trigger incoherences for the whole abstract state. Consider for instance the formula $(k_0 \to k \mid k \mapsto \{\mathtt{f} : \bot, \_ : \top\})$: k can represent an empty set of location, which solves the incoherence. Incoherent summary nodes are removed, which can lead to an incoherence

[3]The default abstract value of the objects can actually be split to make the fields appearing in Φ_2 appear, then perform nevertheless splittings and later on materializations. For readability reasons, we shall not go into details about this as the subset of pre-order we get if we do not make this step is still enough to get a correct abstract semantics.

later on. For instance $(k_0 \rightarrow k \,|\, k \mapsto \{\mathtt{f} : \bot, \_ : \top\} \star \mathtt{x} \doteq k)$ leads to $(k_0 \rightarrow \emptyset \,|\, \mathtt{x} \doteq \bot)$, which has an empty concretisation. We also check for spatial incoherences in ϕ_2: if any, we stop and return *false* (unless ϕ_1 already had an incoherence). At this stage, Φ_2 may have more summary nodes than Φ_1''; this is acceptable as summary nodes can represent empty sets of concrete locations. We thus remove these additional summary nodes in Φ_2 to get Φ_2'.

The last step is a direct comparison between the modified formulae Φ_1'' and Φ_2'. There are three factors to take into account: membranes, environments (expressed through variables), and spatial properties. For membranes, we check that every rewritings in Φ_2' rewrites abstract locations to more abstract locations than in Φ_1'': these renamings represent the inner abstract locations under which these outer abstract locations “hide”. For variables, we check that every variable has a lesser value in Φ_1'' than in Φ_2' in the lattice of abstract values. For spatial properties, we check that each abstract locations h of Φ_1'' is associated a lesser object than in Φ_2'. This step concludes the algorithm.

Note how we perform the splitting and materialization step of the formula Φ_1 before trying to match its locations with these of Φ_2 through ψ. If we had reversed the order of these two operations, the following comparison would not hold. In other words, we can choose in the following example whether l' represents l_1 or l_2 after choosing whether the reference $l.\mathtt{f}$ points to l_1 or l_2.

$$\begin{aligned}&(l_0 \rightarrow l, k \rightarrow l_1 + l_2 \,|\, l \mapsto \{\mathtt{f} : l_1 \sqcup l_2, \_ : \boxtimes\} \star l_1 \mapsto \{\mathtt{a} : \mathit{nil}, \_ : \boxtimes\} \star l_2 \mapsto \{\mathtt{b} : \mathit{nil}, \_ : \boxtimes\})\\ &\preccurlyeq (l_0 \rightarrow l, k \rightarrow l' + l'' \,|\, l \mapsto \{\mathtt{f} : l', \_ : \boxtimes\} \star l' \mapsto \{\mathtt{a} : \mathit{nil} \sqcup \boxtimes, \mathtt{b} : \mathit{nil} \sqcup \boxtimes, \_ : \boxtimes\} \star l'' \mapsto \{\_ : \top\})\end{aligned}$$

This comparison algorithm defines a relation $\preccurlyeq$ with the property of being compatible with the transformations we were looking for: α-conversion of inner locations, materializations, and summarizations. Indeed, if Φ becomes Φ' by one of these transformations, we get $\Phi \preccurlyeq \Phi'$, which allows to rewrite from one to the other when passing through the $\mathit{glue}_i^\sharp$ $(\Downarrow^\sharp)$ function (see Section 6.3). We now claim that this comparison algorithm can be used as a valid relation in this work.

Property 2 *Algorithm 1 defines a suitable relation* $\preccurlyeq$*:*

$$\forall \Phi_1, \Phi_2.\ \Phi_1 \preccurlyeq \Phi_2 \implies \forall \Phi_c.\ \gamma\left(\Phi_c \boxed{\star} \Phi_1\right) \subseteq \gamma\left(\Phi_c \boxed{\star} \Phi_2\right)$$

This property takes as lemmae the similar results for the intermediate parts of the algorithm: it is still valid if we remove the first or the second loop; this will just remove the ability we have to perform materializations and summarizations.