An Abstract Separation Logic for Interlinked Extensible Records

Martin Bodin, Thomas Jensen, Alan Schmitt

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.* [**?**]. Second, JavaScript native structures tend to not separate nicely. Gardner *et al.* propose to remedy this through a *partial* separation operator ⋈ ("seppish"). The formula $P \bowtie Q$ 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 [**?**] semantics for JavaScript, a pretty-big step operational semantics [**?**] 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 [**?**] 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 [**?**] providing the building blocks for constructing an abstract interpretation from a pretty-big step operational semantics, following initial ideas of Schmidt [**?**]. 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 [**?**, 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 $\bowtie$ 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 [**?**].

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 **??**; 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 **??**, to deal with the frame rule. Second, we add the summary nodes from shape analyses to the domain in Section **??**. Section **??** presents how we build our program logic for OWhile, leading in Section **??** to the correctness of our abstract semantics. Section **??** examines related work and Section **??** 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 **??**. A detailed version of the concrete semantics can be found in Appendix **??**, but it comes with no surprises for a pretty-big-step semantics [**?**]. 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

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

(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 $\mathit{Loc} = \{l^i \mid i \in \mathbb{N}^\star\}$ 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 [**?**]. 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 $\rho : \mathit{LLoc}^\sharp \rightharpoonup_{inj} \mathit{Loc}$ (where $\rightharpoonup_{inj}$ 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 **??**. 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

$\mathtt{x}$ satisfies the property $v^\sharp$. We follow the tracks of [**?**] and do not consider this formula pure. As we are not interested in concurrency in this paper, we use a simpler version than [**?**] where we either have full permission over $\mathtt{x}$ 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 $\mathtt{f}_1$, …, $\mathtt{f}_n$ are abstracted by respectively $v_1^\sharp$, …, $v_n^\sharp$, 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 $\mathtt{f}$ and its field $\mathtt{g}$ 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 [**?**]. 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 $\mathtt{f\ in}\ e$ 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,

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

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):

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

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 **??**.

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*.

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

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 $dom(M)$ of a membrane M as the set of left-hand sides of its rewritings, and the codomain $codom(M)$ as the union of the set of right-hand sides of its rewritings and the set of newly allocated locations. The interface $interface(\Phi)$ of a formula $\Phi = (M \mid \phi)$ is the domain of its membrane $dom(M)$: 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 $\Phi_i = (l \to k \mid k \mapsto \{\mathtt{f} : k \sqcup nil, \_ : \boxtimes\})$, 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:

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

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 $\circledast$ 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 $\circledast$ 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 $\Phi_i' = (M_i' \,|\, \phi_i')$ such that $\mathit{codom}(M_i') \cap \mathit{codom}(M_o) = \emptyset$. We then make the formula ϕ_o enter the membrane M_i:

$$\Phi_o \circledast \Phi_i = \Phi_o \circledast \Phi_i' = (M_i' \circ M_o \,|\, M_i'(\phi_o) \star \phi_i')$$

We are now ready to state our frame rule.

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

Although our program logic is not introduced until Section **??**, 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: $\mathtt{x} \doteq \boxtimes$, *if*? *skip* ($\mathtt{x} := ?$) $\Downarrow^\sharp \mathtt{x} \doteq \boxtimes \sqcup \top_{\mathbb{Z}}$.

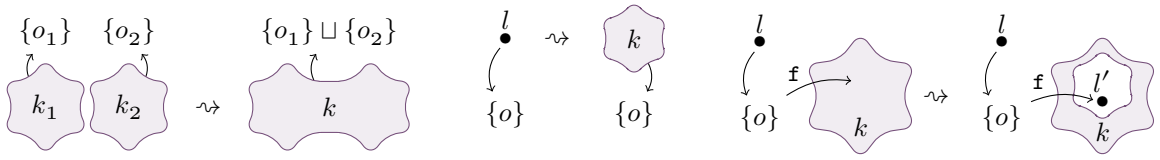
5. Adding Summary Nodes

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 **??** 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 $\mathit{Val}^\sharp = \mathbb{Z}^\sharp \times \left\{\overline{\mathit{nil}}, \mathit{nil}\right\} \times \mathcal{P}_f\left(\mathit{LLoc}^\sharp \uplus \mathit{KLoc}^\sharp\right)^\top \times \{\Box, \boxtimes\}$.

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

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

$$m \in \mathfrak{M} ::= h \to h_1 + ... + h_n \mid \nu h \qquad \Phi ::= (M \mid \phi) \qquad M \in \mathcal{P}_f(\mathfrak{M})$$

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:

$$dom(h \to h_1 + ... + h_n) = \{h\} \qquad codom(h \to h_1 + ... + h_n) = \{h_1, ..., h_n\}$$
$$dom(\nu h) = \emptyset \qquad codom(\nu h) = \{h\}$$
$$dom(M) = \bigcup_{m \in M} dom(m) \qquad codom(M) = \bigcup_{m \in M} codom(m)$$

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 **??** and **??** 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 $h_1 \mapsto \{o_1\} \star ... \star h_n \mapsto \{o_n\} \star \phi'$. The summarization of h_1, …, h_n into k, provided that k does not appear in ϕ, is the following formula.

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

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 **??**. Given an entry point to a summary node k—either on the form of a variable $\mathtt{x} \doteq k$ or a location $l \mapsto \{\mathtt{f} : k, ...\}$—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 **??**. The pre-order $\preccurlyeq$ is defined in Appendix **??**, 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 [**?**] to cover spatial conjunctions and the frame rule. We explain this technique in Section **??**, then present how to abstract rules in Section **??**. Section **??** 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 [**?**]. Pretty-big-step semantics [**?**]

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 [**?**] 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.

$$\textsc{Add2}\ \frac{}{(v_1^\sharp, \mathit{val}^\sharp\, v_2^\sharp), +_2 \Downarrow^\sharp \mathit{add}^\sharp\left(v_1^\sharp, v_2^\sharp\right)} \qquad \textsc{If}(e, s_1, s_2)\ \frac{\Phi, e \Downarrow^\sharp \Phi' \qquad \Phi', \mathit{if}_1\, s_1\, s_2 \Downarrow^\sharp \Phi''}{\Phi, \mathit{if} e\, s_1\, s_2 \Downarrow^\sharp \Phi''}$$

However, as explained in [**?**], 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 ($\mathit{cond}_i^\sharp(\Phi)$ 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 $\mathit{glue}_i^\sharp\left(\Downarrow_0^\sharp\right)$ which computes all triples obtainable from the application of the ith rule:

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

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

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

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 ($\mathit{cond}^\sharp$). 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 **??** in Appendix **??** 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:

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

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 [**?**], 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 [**?**, Section 3.5], as the frame rule can no longer interfere with the resource $\mathtt{x}$.

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

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.

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

6.3. Interfering with the Frame Rule

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

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

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 $\Phi_2 = (l' \to l \,|\, l \mapsto \{\_ : \boxtimes\})$ represent the same concrete states (they have the same concretisation), in the context of $\Phi_c = (k \to l + l' \,|\, l \mapsto \{\_ : \boxtimes\})$,

$\Phi_c \circledast \Phi_1$ 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 [**?**], meta rules are not mixed with abstract rules. We thus force the frame rule into the glue between rules by updating the function $glue^\sharp$, as we did when adding the WEAKEN rule:

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

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 **??**). 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 ASNG1 ($\mathtt{x}$): 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 $\mathtt{x}$ to stand in the input formula:

$$\frac{}{\left(M \,\middle|\, \mathtt{x} \doteq v_0^\sharp, v^\sharp\right), \mathtt{x} :=_1 \Downarrow^\sharp \left(M \,\middle|\, \mathtt{x} \doteq v^\sharp\right)} \text{ASGN1}(\mathtt{x})$$

Given a semantic context $\Phi = \left(M \,\middle|\, \phi, v^\sharp\right)$, there are two cases, whether $\mathtt{x}$ 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 $\mathtt{x}$ into $\Phi = \left(M \,\middle|\, \phi'\right) \circledast \left(M' \,\middle|\, \mathtt{x} \doteq v_0^\sharp, v^\sharp\right)$, where $M' = \{h \to h | h \in codom\left(M\right)\}$ 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 DELETE1 ($\mathtt{f}$). 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 $\Phi = \left(k \to k \,\middle|\, k \mapsto \{\mathtt{f} : nil, \_ : \boxtimes\}, k\right)$.

$$\frac{}{\left(M \,\middle|\, l \mapsto \{o\}, l\right), delete_1\ .\mathtt{f} \Downarrow^\sharp \left(M \,\middle|\, l \mapsto remove^\sharp\left(\mathtt{f}, \{o\}\right)\right)} \text{DELETE1}(\mathtt{f})$$

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

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

We can now apply the abstract rule DELETE1 ($\mathtt{f}$) on the first part to get the following.

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

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: $\Phi'' \preccurlyeq \left(k \to k \,\middle|\, k \mapsto \{\mathtt{f} : nil \sqcup \boxtimes, \_ : \boxtimes\}, k\right)$. 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 **??**; 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:

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

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

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

The approach for correctness is the same than in [**?**]: we require every abstract rule to be locally correct, i.e., their transfer functions (noted for axioms ax and $ax^\sharp$ 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.

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

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 [**?**], 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 [**?**], 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 [**?**] to a fully Coq-verified abstract interpreter.

There have been some work about mixing abstract interpretation and separation logic, such as [**?**] or [**?**], 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 [**?**] 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 [**?**] 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 [**?**], 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 [**?**] 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

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

(a) Statements

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

(b) Expressions

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

(c) Extended statements

$$e_e ::= .\mathtt{f} \mid \mathtt{f}\ \mathtt{in}_1 \mid \bowtie_1\ e_2 \mid \bowtie_2 \mid \neg_1 \mid =_1\ e_2 \mid =_2$$

(d) Extended expressions

Figure 4: Complete syntax of the OWhile language

like bi-abduction may be relevant to build efficient oracles for our certified analysers.

A. Concrete Semantics

Figure **??** 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 **??** categorizes rules into four different kinds of categories; Figure **??** shows 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.

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

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

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

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

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

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

$$\textsc{In1False}(\mathtt{f})\ \frac{}{(S, l^i), \mathtt{f\ in}_1 \Downarrow (S, 0)}\ l^i.\mathtt{f} \notin dom(S)$$

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

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

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

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

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

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

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

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

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

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

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

$$\textsc{Asgn}(\mathtt{x}, e)\ \frac{S, e \Downarrow r \qquad r, \mathtt{x} :=_1 \Downarrow r'}{S, \mathtt{x} := e \Downarrow r'} \qquad \textsc{Asgn1}(\mathtt{x})\ \frac{}{(S, v), \mathtt{x} :=_1 \Downarrow \mathit{write}\,(S, x, v)} \qquad \textsc{PropertyAsgn}(e_1, \mathtt{f}, e_2)\ \frac{S, e_1 \Downarrow r \qquad r, \mathtt{.f} :=_1 e_2 \Downarrow r'}{S, e_1\,\mathtt{.f} := e_2 \Downarrow r'}$$

$$\textsc{PropertyAsgn1}(\mathtt{f}, e_2)\ \frac{S, e_2 \Downarrow r \qquad (l^i, r), \mathtt{.f} :=_2 \Downarrow r'}{(S, l^i), \mathtt{.f} :=_1 e_2 \Downarrow r'} \qquad \textsc{PropertyAsgn1Error}(\mathtt{f}, e_2)\ \frac{}{(S, n), \mathtt{.f} :=_1 e_2 \Downarrow \mathit{Err}} \qquad \textsc{PropertyAsgn1Nil}(\mathtt{f}, e_2)\ \frac{}{(S, l^0), \mathtt{.f} :=_1 e_2 \Downarrow \mathit{Err}}$$

$$\textsc{PropertyAsgn2}(\mathtt{f})\ \frac{}{(l^i, (S, v)), \mathtt{.f} :=_2 \Downarrow \mathit{write}\,(S, l^i\,\mathtt{.f}, v)} \qquad \textsc{Delete}(e, \mathtt{f})\ \frac{S, e \Downarrow r \qquad r, \mathit{delete}_1\,\mathtt{.f} \Downarrow r'}{S, \mathit{delete}\, e\,\mathtt{.f} \Downarrow r'}$$

$$\textsc{Delete1}(\mathtt{f})\ \frac{}{(S, l^i), \mathit{delete}_1\,\mathtt{.f} \Downarrow \mathit{remove}\,(S, l^i\,\mathtt{.f})} \qquad \textsc{Delete1Error}(\mathtt{f})\ \frac{}{(S, n), \mathit{delete}_1\,\mathtt{.f} \Downarrow \mathit{Err}} \qquad \textsc{Delete1Nil}(\mathtt{f})\ \frac{}{(S, l^0), \mathit{delete}_1\,\mathtt{.f} \Downarrow \mathit{Err}}$$

B. Pre-order over Formulae

As said in Section **??**, 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 $\Phi_1 = (M_1 \,|\, \phi_1)$ and $\Phi_2 = (M_2 \,|\, \phi_2)$ 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 $\Phi_1 \preccurlyeq \Phi_2$.

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

Figure 5: Categories of rules

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

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: $\forall \Phi_c.\ \gamma\left(\Phi_c \circledast \Phi_1\right) = \gamma\left(\Phi_c \circledast \Phi_{1,a}\right) \cup \gamma\left(\Phi_c \circledast \Phi_{1,b}\right)$.

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

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 **??** and **??**—which is why summarizations are compatible with this pre-order:

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

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:

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

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 $\Phi'_2 = (k_0 \to l', l_0 \to l' \,\big|\, \mathtt{x} \doteq l' \star l' \mapsto \{\mathtt{f} : l' \sqcup \mathit{nil}, \_ : \boxtimes\})$. 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 $\forall \Phi_c.\ \gamma\left(\Phi_c \circledast \Phi_{1,a}\right) \subseteq \gamma\left(\Phi_c \circledast \Phi_2\right)$.

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 **??** 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 defined abstract values, we can split values of the form $n^\sharp \sqcup \mathit{nil}^? \sqcup l_1 \sqcup ... \sqcup l_n \sqcup d$ 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 $\{\_ : \mathit{nil} \sqcup \boxtimes\}$, 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 **??**. 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 $\psi_l : \mathit{LLoc}^\sharp \rightharpoonup \left(\mathit{LLoc}^\sharp \cup \mathit{KLoc}^\sharp\right)$ and $\psi_k : \mathit{KLoc}^\sharp \rightharpoonup \mathit{KLoc}^\sharp$ 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 $\Phi''_1 = \psi(\Phi'_1) = (\psi \mid \mathit{emp}) \circledast \Phi'_1$ 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 $\Phi'_1 = (k_0 \to k, l_0 \to l \,|\, l \mapsto \{o_1\} \star k \mapsto \{o_2\})$, then $\Phi''_1 = (k_0 \to k', l_0 \to k' \,\big|\, k' \mapsto \{o'_1\} \sqcup \{o'_2\})$ 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

[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.

Data: Two abstract heaps Φ_1 and Φ_2.
Result: *true* iff $\Phi_1 \preccurlyeq \Phi_2$.
if *interface* $(\Phi_1) \neq$ *interface* (Φ_2) **then**
  return *false*; `// Formulae with different interfaces are immediately rejected.`
end
for Φ_1' *be a split/materialization of* Φ_1 **do** `// Universal branching`
  for ψ *well-formed* **do** `// Existential choice`
    $\Phi_1'' \leftarrow \psi(\Phi_1')$;
    Remove incoherent summary nodes from Φ_1'';
    Let Φ_2' be Φ_2 where every summary node not present in Φ_1'' has been removed;
    if Φ_1'' *incoherent* **then**
      `//` The current ψ `is enough to show that` $\Phi_1' \preccurlyeq \Phi_2$`.`
      Continue on the next split Φ_1' of Φ_1;
    end
    if Φ_2' *incoherent* **then**
      Try another ψ;
    end
    for $h \to h_1 + ... + h_n$ *present in* Φ_2' *and* $h \to h_1' + ... + h_m'$ *in* Φ_1'' **do**
      if $\{h_1', ..., h_m'\} \not\subseteq \{h_1, ..., h_n\}$ **then**
        Try another ψ;
      end
    end
    for $\mathtt{x} \doteq v_2^\sharp$ *present in* Φ_2' *and* $\mathtt{x} \doteq v_1^\sharp$ *present in* Φ_1'' **do**
      if $v_1^\sharp \not\sqsubseteq v_2^\sharp$ **then**
        Try another ψ;
      end
    end
    for $h \mapsto \{o_2\}$ *present in* Φ_2' *and* $h \mapsto \{o_1\}$ *present in* Φ_1'' **do**
      if $\{o_1\} \not\sqsubseteq \{o_2\}$ **then**
        Try another ψ;
      end
    end
    `//` The chosen ψ `succeeded in matching` Φ_1' `with` Φ_2'`.`
    Continue on the next split Φ_1' of Φ;
  end
  `//` No ψ `succeeded in matching` Φ_1' `with` Φ_2'`.`
  return *false*;
end
`//` Φ_2 `correctly captures all the behaviours of` Φ_1`.`
return *true*;

Algorithm 1: Algorithm comparing formulae.

efficiently, by only considering the possible ones in an iterative algorithm similar to the one of [**?**]. 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 `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 $(k_0 \to k \,|\, k \mapsto \{\mathtt{f} : \bot, \_ : \top\})$: k can represent an empty set of location, which solves the incoherence. Incoherent summary nodes are removed, which can lead to an incoherence later on. For instance $(k_0 \to k \,|\, k \mapsto \{\mathtt{f} : \bot, \_ : \top\} \star \mathtt{x} \doteq k)$ leads to $(k_0 \to \emptyset \,|\, \mathtt{x} \doteq \bot)$, 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.

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

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 $glue_i^\sharp$ ($\Downarrow^\sharp$) function (see Section **??**). We now claim that this comparison algorithm can be used as a valid relation in this work.

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

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

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.