Pretty-big-step-semantics-based Certified Abstract Interpretation

Martin Bodin, Thomas Jensen, Alan Schmitt

Pretty-big-step-semantics-based Certified Abstract Interpretation

Martin Bodin[1,2], Thomas Jensen[2], & Alan Schmitt[2]

1: ENS Lyon; 2: Inria; `firstname.lastname@inria.fr`

Résumé

We present a technique for deriving semantic program analyses from a natural semantics specification of the programming language. The technique is based on the pretty-big-step semantics approach applied to a language with simple objects called O'While. We specify a series of instrumentations of the semantics that makes explicit the flows of values in a program. This leads to a semantics-based dependency analysis, at the core, *e.g.*, of tainting analysis in software security. The formalization is currently being done with the Coq proof assistant.[1]

1. Introduction

David Schmidt gave an invited talk at the 1995 Static Analysis Symposium [11] in which he argued for using natural semantics as a foundation for designing semantic program analyses within the abstract interpretation framework. With natural (or "big-step" or "evaluation") semantics, we can indeed hope to benefit from the compositional nature of a denotational-style semantics while at the same time being able to capture intentional properties that are best expressed using an operational semantics. Schmidt showed how a control flow analysis of a core higher-order functional language can be expressed elegantly in his framework. Subsequent work by Gouranton and Le Métayer showed how this approach could be used to provide a natural semantics-based foundation for program slicing [13].

In this paper, we will pursue the research agenda set out by Schmidt and investigate further the systematic design of semantics-based program analyses based on big-step semantics. Two important issues here will be those of scalability and mechanization. The approach worked nicely for a language whose semantics could be defined in 8 inference rules. How will it react when applied to full-blown languages where the semantic definition comprises hundreds of rules? Strongly linked to this question is that of how the framework can be mechanized and put to work on larger languages using automated tool support. In the present work, we investigate how the Coq proof assistant can serve as a tool for manipulating the semantic definitions and certifying the correctness of the derived static analyses.

Certified static analysis is concerned with developing static analyzers inside proof assistants with the aim of producing a static analyzer and a machine-verifiable proof of its semantic correctness. One long-term goal of the work reported here is to be able to provide a mechanically verified static analysis for the full JavaScript language based on the Coq formalization developed in the JSCert project [1]. JavaScript, with its rich but sometimes quirky semantics, is indeed a good *raison d'être* for studying certified static analysis, in order to ensure that all of the cases in the semantics are catered for.

In our development, we shall take advantage of some recent developments in the theory of operational semantics. In particular, we will be using a particular format of natural semantics call

[1]This work has been presented at the Festschrift for David Schmidt in September 2013. This work has been partially supported by the French National Research Agency (ANR), project Typex ANR-11-BS02-007, and by the Laboratoire d'excellence CominLabs ANR-10-LABX-07-01.

$s ::=$
| `skip`
| $s_1 ; s_2$
| `if` e `then` s_1 `else` s_2
| `while` e `do` s
| `x =` e
| e_1`.f =` e_2
| `delete` e`.f`

$e ::=$
| c
| `x`
| e_1 `op` e_2
| `{}`
| e`.f`

$v ::=$
| c
| l

$r ::=$
| S
| S, v
| $S,$ `err`

$s_e ::=$
| s
| $r ;_1 s$
| `if1`(r, s_1, s_2)
| `while1`(r, e, s)
| `while2`(r, e, s)
| `x` $=_1 r$
| r`.f` $=_1 e$
| l`.f` $=_2 r$
| `delete1` r`.f`

$e_e ::=$
| e
| r `op`$_1$ e
| v `op`$_2$ r
| r `.`$_1$`f`

Figure 1: O'While Syntax, Values, Results, and Extended Syntax

"pretty-big-step" semantics [3] which is a streamlined form of operational semantics retaining the format of natural semantics while being closer to small-step operational semantics.

Even though it is our ultimate goal, JavaScript is far too big to begin with as a goal for analysis: its pretty-big-step semantics contains more than half a thousand rules! We will thus start by studying a much simpler language, called O'While, which is basically a While language with simple objects in the form of extensible records. This language is quite far from JavaScript, but is big enough to catch some issues of the analyses of JavaScript objects. We present the language and its pretty-big-step semantics in Section 2. To test the applicability of the approach to defining static analyses, we have chosen to formalize a data flow dependency analysis as used, *e.g.*, in tainting [12] or "direct information-flow" analyses of JavaScript [15, 5]. The property we ensure is defined in Section 3 and the analysis itself is defined in Sections 4. As stated above, the scalability of the approach relies on the mechanization that will enable the developer of the analyses to prove the correctness of analyses with respect to the semantics, and to extract an executable analyzer. We show how the Coq proof assistant is currently being used to formally achieve these objectives as we go along.

2. O'While and its Pretty Big Step Semantics

As big-step semantics, pretty-big-step semantics directly relates terms to their results. However, pretty-big-step semantics avoids the duplication associated with big-step semantics when features such as exceptions and divergence are added. Since duplication in the definitions often leads to duplication in the formalization and in the proofs, an approach based on a pretty-big-step semantics allows to deal with programming languages with many complex constructs. (We refer the reader to Charguéraud's work on pretty-big-step semantics [3] for detailed information about this duplication.) Even though the language considered here is not complex, we have been using pretty-big-step semantics exclusively for our JavaScript developments, thus we will pursue this approach in the present study.

The syntax of O'While is presented in Figure 1. Two new constructions have been added to the syntax of expressions for the usual While language: `{}` creates a new object, and e`.f` accesses a field of an object. Regarding statements, we allow the addition or the modification of a field to an object using e_1`.f =` e_2, and the deletion of the field of an object using `delete` e`.f`. In the following we write t for terms, *i.e.*, both expressions and statements.

Objects are passed by reference. Values v are either locations l or primitive values c. In this work,

we only consider boolean primitive values. The *state* of a program contains both an *environment* E, which is a mapping from variables to values, and a *heap* H, which is a mapping from locations to objects, that are themselves mappings from fields to values. In the following, we write S for E, H when there is no need to access the environment nor the heap. Results r are either a state S, a pair of a state and value S, v, or a pair of an error and a state $S, \texttt{err}$.

Figure 1 also introduces *extended statements* and *extended expressions* that are used in O'While's pretty-big-step semantics, presented in Figure 2. Extended terms t_e comprise extended statements and expressions. Reduction rules have the form $S, t_e \to r$. The result r can be an error $S', \texttt{err}$. Otherwise, if t_e is an extended statement, then r is a state S', and if t_e is an extended expression, then r is a pair of a state S' and returned value v. We write $\texttt{st}(r)$ for the state S in a result r.

Most rules are the usual While ones, with the exception that they are given in pretty-big-step style. We now detail the new rules for expressions and statements. Rule Obj associates an empty object to a fresh location in the heap. Rule Fld for the expression $e.\texttt{f}$ first evaluates e to some result r, then calls the rule for the extended expression $r\,._1\texttt{f}$. The rule for this extended expression is only defined if r is of the form E', H', l where l is a location in H' that points to an object o containing a field $\texttt{f}$. The rules for field assignment and field deletion are similar: we first evaluate the expression that defines the object to be modified, and in the case it actually is a location, we modify this object using an extended statement.

Finally, our semantics is parameterized by a partial function $\texttt{abort}(\cdot)$ from extended terms to results, that indicates when an error is to be raised or propagated. More precisely, the function $\texttt{abort}(t_e)$ is defined at least if t_e is an extended term containing a subterm equal to $S, \texttt{err}$ for some S. In this case $\texttt{abort}(t_e) = S, \texttt{err}$. We can then extend this function to define erroneous cases. For instance, we could say that $\texttt{abort}((E, H, v).\texttt{f} =_1 e) = E, H, \texttt{err}$ if v is not a location, or if $v = l$ but l is not in the domain of H, or if $\texttt{f}$ is not in the domain of $H[l]$. This function is used in the Abort rule, that defines when an error is raised or propagated. This illustrates the benefit of a pretty-big-step semantics: a single rule covers every possible error propagation case.

The derivation in Figure 14 in Appendix B is an example of a derivation of the semantics.

3. Annotated Semantics

3.1. Execution traces

We want to track how data created at one point flows into locations (variables or object fields) at later points in the execution of the program. To this end, we need a mechanism for talking about "points of time" in a program execution. This information is implicit in the semantic derivation tree corresponding to the execution. To make it explicit, we instrument the semantics to produce a (linear) trace of the inference rules used in the derivation, and use it to refer to particular points in the execution. As every other instrumentation in this paper, it adds no information to the derivation but allows global information to be discussed locally.

More precisely, we add partial traces, $\tau \in \mathit{Trace}$, to both sides of the reduction rules. These traces are lists of names used in the derivation. The two crucial properties from traces is that they uniquely identify a point in the derivation (i.e., a rule in the tree and a side of this rule), and that one may derive from the trace the syntactic program point that is being executed at that point.

Since traces uniquely identify places in a derivation, we use them from now on to refer to states or further instrumentation in the derivation. More precisely, if τ is a trace in a given derivation, we write E_τ and H_τ for the environment and heap at that point.

$$\frac{}{S, \texttt{skip} \to S}\ \textsc{Skip} \qquad \frac{S, s_1 \to r \qquad S, r\ ;_1\ s_2 \to r'}{S, s_1; s_2 \to r'}\ \textsc{Seq} \qquad \frac{S', s \to r}{S, S'\ ;_1\ s \to r}\ \textsc{Seq1}$$

$$\frac{S, e \to r \qquad S, \texttt{if1}(r, s_1, s_2) \to r'}{S, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \to r'}\ \textsc{If} \qquad \frac{S', s_1 \to r}{S, \texttt{if1}((S', \texttt{true}), s_1, s_2) \to r}\ \textsc{IfTrue}$$

$$\frac{S', s_2 \to r}{S, \texttt{if1}((S', \texttt{false}), s_1, s_2) \to r}\ \textsc{IfFalse} \qquad \frac{S, e \to r \qquad S, \texttt{while1}(r, e, s) \to r'}{S, \texttt{while}\ e\ \texttt{do}\ s \to r'}\ \textsc{While}$$

$$\frac{S', s \to r \qquad S', \texttt{while2}(r, e, s) \to r'}{S, \texttt{while1}((S', \texttt{true}), e, s) \to r'}\ \textsc{WhileTrue1} \qquad \frac{S', \texttt{while}\ e\ \texttt{do}\ s \to r}{S, \texttt{while2}(S', e, s) \to r}\ \textsc{WhileTrue2}$$

$$\frac{}{S, \texttt{while1}((S', \texttt{false}), e, s) \to S'}\ \textsc{WhileFalse} \qquad \frac{E, H, e \to r \qquad E, H, \texttt{x} =_1 r \to r'}{E, H, \texttt{x = } e \to r'}\ \textsc{Asg}$$

$$\frac{E' = E[\texttt{x} \mapsto v]}{S, \texttt{x} =_1 (E, H, v) \to E', H}\ \textsc{Asg1} \qquad \frac{S, e_1 \to r \qquad S, r.\texttt{f} =_1 e_2 \to r'}{S, e_1.\texttt{f = } e_2 \to r'}\ \textsc{FldAsg}$$

$$\frac{S', e \to r \qquad S', l.\texttt{f} =_2 r \to r'}{S, (S', l).\texttt{f} =_1 e \to r'}\ \textsc{FldAsg1} \qquad \frac{H[l] = o \qquad o' = o\,[\texttt{f} \mapsto v] \qquad H' = H\,[l \mapsto o']}{S, l.\texttt{f} =_2 (E, H, v) \to E, H'}\ \textsc{FldAsg2}$$

$$\frac{S, e \to r \qquad S, \texttt{delete1}\ r.\texttt{f} \to r'}{S, \texttt{delete}\ e.\texttt{f} \to r'}\ \textsc{Del}$$

$$\frac{H[l] = o \qquad o[\texttt{f}] \neq \bot \qquad o' = o\,[\texttt{f} \mapsto \bot] \qquad H' = H\,[l \mapsto o']}{S, \texttt{delete1}\ (E, H, l).\texttt{f} \to E, H'}\ \textsc{Del1} \qquad \frac{}{S, c \to S, c}\ \textsc{Cst}$$

$$\frac{E[\texttt{x}] = v}{E, H, \texttt{x} \to E, H, v}\ \textsc{Var} \qquad \frac{S, e_1 \to r \qquad S, r\ \texttt{op}_1\ e_2 \to r'}{S, e_1\ \texttt{op}\ e_2 \to r'}\ \textsc{Bin} \qquad \frac{S', e_2 \to r \qquad S', v_1\ \texttt{op}_2\ r \to r'}{S, (S', v_1)\ \texttt{op}_1\ e_2 \to r'}\ \textsc{Bin1}$$

$$\frac{v = v_1\ \texttt{op}\ v_2}{S, v_1\ \texttt{op}_2\ (S, v_2) \to S, v}\ \textsc{Bin2} \qquad \frac{H[l] = \bot \qquad H' = H[l \mapsto \{\}]}{E, H, \{\} \to E, H', l}\ \textsc{Obj} \qquad \frac{S, e \to r \qquad S, r\ ._1\texttt{f} \to r'}{S, e.\texttt{f} \to r'}\ \textsc{Fld}$$

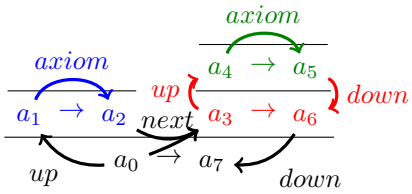
$$\frac{H'[l] = o \qquad o[\texttt{f}] = v}{E, H, (E', H', l)\ ._1\texttt{f} \to E', H', v}\ \textsc{Fld1} \qquad \frac{\texttt{abort}(t_e) = r}{S, t_e \to r}\ \textsc{Abort}$$

Figure 2: O'While's Semantics

3.2. A General Scheme to Define Annotations

In principle, the annotation process takes as argument a full derivation tree and returns an annotated tree. However, every annotation process we define in the following, as well as the one deriving traces, can be described by an iterative process that takes as arguments previous annotations and the parameters of the rule applied, and returns an annotated rule.

More precisely, our iterative process is based on steps of four kinds: *axiom* steps (for axioms), that transform the annotations on the left of axiom rules into annotations on the right of the rule, *up* steps (for rules with inductive premises), that propagate an annotation on the left of a rule to its first premise, *down* steps (for rules with inductive premises), that propagate an annotation on the right of the last premise to the right of the rule, and *next* steps (for rules with two inductive premises), that propagate the annotations from the left of the current rule and from the right of the first premise into the left of the second premise. As we are using a pretty-big-step semantics, there are at most two inductive premises above each rule, thus these steps are sufficient.



This generic approach allows to compose complex annotations, building upon previously defined ones. This general scheme is summed up on the right, where each a_i represents an annotation. The colors show which steps are associated to which rules: a_0 is the initial annotation. It is changed to a_1 and control is passed to the left axiom rule (black *up*). The blue *axiom* step creates a_2, and control returns to the bottom rule, where the black *next* step combines a_2 and a_0 to pass it to the red rule. Annotations are propagated in the right premise, and ultimately control comes back to the black rule which pulls the a_6 annotation from the red rule and creates its a_7 annotation. Note that the types of the annotations on the left and the right of the rules do not have to be the same, as long as every left-hand side annotation has the same type, and the same for right-hand side annotations.

As an example, we define the *axiom*, *up*, *down*, and *next* steps corresponding to the addition of partial traces for rules Var and Asg (see Figure 3b and 4b). Rule Var illustrates the *axiom* step, that adds a Var token at the end of the trace. Rule Asg illustrates the other steps: adding a AsgE as *up* step, adding a $\overline{\text{AsgE}}$ as *next* step, and adding Asg as *down* step. We fully describe in Appendix A.2 how the traces are added following this approach.

3.3. Dependency Relation

We are interested in deriving the dependency analysis underlying tainting analyses for checking that secret values do not flow into other values that are rendered public. To this end, we consider *direct flows* from *sources* to *stores*. We need a mechanism for describing when data was created and when a flow happened, so we annotate locations in the heap with the time when they were allocated. By "time" we here mean the point of time in an execution, represented by a trace τ of the derivation. We write $ALoc = Loc \times Trace$ for the set of *annotated locations*. Similarly, we annotate variables and fields with the point in time that they were last assigned to. When describing a flow, we talk about *sources* and *stores*. Sources are of three kinds: an annotated location, a variable annotated with its last modification time, or a pair of annotated location and field further annotated with their last modification time. Stores are either a variable or a pair of an annotated location and a field, further annotated with their last modification time. Formally, we define the following dependency relation

$$\leftsquigarrow \in Dep = \mathcal{P}\left(Source \times Store\right)$$

where $Store = (Var \times Trace) + (ALoc \times Field \times Trace)$ and $Source = ALoc + Store$.

For instance, we write $\mathtt{y}^{\tau_1} \leftsquigarrow \mathtt{x}^{\tau_2}$ to indicate that the content that was put in the variable $\mathtt{y}$ at time τ_1 has been used to compute the value stored in the variable $\mathtt{x}$ at time τ_2. Similarly, we write

$$\textsc{Var}\ \frac{E[\underline{\mathtt{x}}] = v}{E, H, \underline{\mathtt{x}} \to E, H, v}$$

(a) Basic Rule

$$\frac{E[\underline{\mathtt{x}}] = v \qquad \tau' = \tau +\!\!+ [\textsc{Var}]}{E, H, \tau, \underline{\mathtt{x}} \to E, H, \tau', v}\ \textsc{Var}$$

(b) Adding Partial Traces

$$\textsc{Var}\ \frac{E[\underline{\mathtt{x}}] = v \qquad \tau' = \tau +\!\!+ [\textsc{Var}]}{E, H, \tau, M, \underline{\mathtt{x}} \to E, H, \tau', M, v}$$

(c) Adding Last-Modified Place

$$\frac{E[\underline{\mathtt{x}}] = v \qquad \tau' = \tau +\!\!+ [\textsc{Var}] \qquad M[\mathtt{x}] = \tau_0}{E, H, \tau, M, \underline{\mathtt{x}} \to E, H, \tau', M, \{\mathtt{x}^{\tau_0}\}, v}\ \textsc{Var}$$

(d) Adding Dependencies

Figure 3: Instrumentation Steps for Var

$l^{\tau_2} \leftsquigarrow l'^{\tau_1}.\mathtt{f}^{\tau_3}$ to indicate that the object allocated at location l at time τ_2 flows at time τ_3 into field $\mathtt{f}$ of location l' that was allocated at time τ_1.

3.4. Direct Flows

We now detail how to compose additional annotations to define our direct flow property $\leftsquigarrow$. As flows are a global property of the derivation, we use a series of annotations to propagate local information until we can locally define direct flows.

We first collect in the derivation the traces where locations are created and where variables or object fields are assigned. To this end, we define a new annotation M of type $(\mathit{Loc} + \mathit{Var} + \mathit{ALoc} \times \mathit{Field}) \to \mathit{Trace}$. After this instrumentation step, reductions are of the form $\tau, M_\tau, S_\tau, t \to \tau', M_{\tau'}, r$. Note that the trace in information in $\mathit{ALoc} \times \mathit{Field}$ is redundant in our setting, as locations may not be reused. It is however useful when showing the correspondence with the analyses as the trace information lets us derive the program point at which the location was allocated.

The three rules that modify M are Obj, Asg1, and FldAsg2. We describe them in Figure 5. The other rules simply propagate M. For the purpose of our analysis, we do not consider the deletion of a field as its modification. More precise analyses, in particular ones that also track indirect flows, would need to record such events.

The added instrumentation uses traces to track the moments when locations are created, and when fields and variables are assigned. For field assignment, the rule FldAsg2 relies on the fact that the location of the object assigned has already been created to obtain the annotated location: we have the invariant that if $H[l]$ is defined, then $M[l]$ is defined.

We can now continue our instrumentation by adding *dependencies* $d \in \mathcal{P}(\mathit{Var})$. The instrumented reduction is now $\tau, M_\tau, d_\tau, S_\tau, t \to \tau', M_{\tau'}, d_{\tau'}, r$. Its rules are described in Figure 6. The rules not given only propagate the dependencies. The intuition behind these rules is that expressions generate potential dependencies that are thrown away when they don't result in direct flow (for instance when computing the condition of a If statement). The important rules are Var, where the result depends on the last time the variable was modified, Obj, which records the dependency on the creation of the object, and Fld1, whose result depends on the last time the field was assigned. The Asg and FldAsg1 rules make sure these dependencies are transmitted to the inductive call to the rule that will proceed with the assignment for the next series of annotations.

Finally, we build upon this last instrumentation to define flows. The final instrumented derivation is of the form: $\tau, M_\tau, d_\tau, \Delta_\tau, S_\tau, s \to \tau', M_{\tau'}, d_{\tau'}, \Delta_{\tau'}, r_{\tau'}$, where $\{\Delta_\tau, \Delta_{\tau'}\} \subseteq \mathit{Dep}$ are sets of flows defining the $\leftsquigarrow$ relation (see Section 3.3). The two important rules are Asg1 and FldAsg2, which modify respectively a variable and a field, and for which the flow needs to be added. All the other

$$\frac{S, e \to r \qquad S, \underline{\mathtt{x} =_1 r} \to r'}{S, \underline{\mathtt{x} = e} \to r'}\ \textsc{Asg}$$

(a) Basic Rule

$$\tau_1 = \tau_0 +\!\!+ [\textsc{AsgE}] \qquad \tau_3 = \tau_2 +\!\!+ [\overline{\textsc{AsgE}}] \qquad \tau_5 = \tau_4 +\!\!+ [\textsc{Asg}]$$

$$\frac{\tau_1, S, e \to \tau_2, r \qquad \tau_3, S, \underline{\mathtt{x} =_1 r} \to \tau_4, r'}{\tau_0, \underline{\mathtt{x} = e}, \to \tau_5, r'}\ \textsc{Asg}$$

(b) Adding Partial Traces

$$\frac{\tau_1, M, S, e \to \tau_2, M', r \qquad \tau_3, M', S, \underline{\mathtt{x} =_1 r} \to \tau_4, M'', r'}{\tau_0, M, \underline{\mathtt{x} = e}, \to \tau_5, M'', r'}\ \textsc{Asg}$$

(c) Adding Last-Modified Place

$$\frac{\tau_1, M, \emptyset, \Delta, S, e \to \tau_2, M', d, \Delta, r \qquad \tau_3, M', d, \Delta, S, \underline{\mathtt{x} =_1 r} \to \tau_4, M'', \emptyset, \Delta', r'}{\tau_0, M, \emptyset, \Delta, S, \underline{\mathtt{x} = e} \to \tau_5, M'', \emptyset, \Delta', r'}\ \textsc{Asg}$$

(d) Adding Dependencies

Figure 4: Instrumentation Steps for Asg

rules just propagate those new constructions. The two modified rules are given in Figure 7.

3.5. Correctness Properties of the Annotations

The instrumentation of the semantics does not add information to the reduction but only makes existing information explicit. The correctness of the instrumentation can therefore be expressed as a series of consistency properties between the different instrumented semantics.

We first state correctness properties about the instrumentation of the heap. We start by a property concerning the last-modified-place annotations. This property states that the annotation of a location's creation point never changes, and that the value of a field has not changed since the point of modification indicated by the instrumentation component M.

Property 1 *For every instrumented tree, and for every rule in this tree $\tau, M_\tau, E_\tau, H_\tau, t \to \tau', M_{\tau'}, r$ where $\mathtt{st}(r) = E_{\tau'}, H_{\tau'}$ and $M_{\tau'}[l^{\tau_0}.\mathtt{f}] = \tau_1$; we have $M_{\tau'}[l] = \tau_0$ and $H_{\tau'}[l][\mathtt{f}] = H_{\tau_1}[l][\mathtt{f}]$.*

The following property links the last-change-place annotation (M) with the dependencies annotation (Δ). Intuitively, it states that if Δ says that the value assigned to x at time τ_1 later flew into a variable a time τ_2 then x has not changed between τ_1 and τ_2.

Property 2 *For every instrumented tree, and for every rule in this tree $s, \tau, M_\tau, d_\tau, \Delta_\tau, S_\tau, t \to \tau', M_{\tau'}, d_{\tau'}, \Delta_{\tau'}, r$ if $\mathtt{x}^{\tau_1} \leftrightsquigarrow \mathtt{y}^{\tau_2} \in \Delta_{\tau'}$, then at time τ_2, the last write to $\mathtt{x}$ was at time τ_1,* i.e., *$M_{\tau_2}[\mathtt{x}] = \tau_1$.*

We now state the most important property: if at some point during the execution of a program the field of an object contains another object, then there is a chain of direct flows attesting it in the

$$\frac{H[l] = \bot \qquad H' = H[l \mapsto \underline{\{\}}] \qquad M' = M[l \mapsto \tau']}{\tau, M, E, H, \underline{\{\}} \to \tau', M', E, H', l}\ \textsc{Obj}$$

$$\frac{H[l] = o \qquad o' = o\,[\mathtt{f} \mapsto v] \qquad H' = H'\,[l \mapsto o'] \qquad M' = M[(l, M[l], f) \mapsto \tau']}{\tau, M, S, \underline{l.\mathtt{f} =_2 (E, H, v)} \to \tau', M', E, H'}\ \textsc{FldAsg2}$$

$$\frac{E' = E[\mathtt{x} \mapsto v] \qquad M' = M[\mathtt{x} \mapsto \tau']}{\tau, M, S, \underline{\mathtt{x} =_1 (E, H, v)} \to \tau', M', E', H}\ \textsc{Asg1}$$

Figure 5: Adding Modified and Created Information

annotation. More precisely, we write $l_0 \leadsto^*_\Delta l_n.\mathtt{f}$ if there are stores $s_0 \ldots s_n$ such that: $s_0 = l_0^{\tau_0}$ for some τ_0, $s_n = l_n^{\tau_n}.\mathtt{f}^{\tau'_n}$ for some τ_n and τ'_n, and for every i in $[1..n]$ we have either $s_i = l_i^{\tau_i}.\mathtt{f}_i{}^{\tau'_i}$ for some l_i, $\mathtt{f}_i$, τ_i, and τ'_i or $s_i = \mathtt{x}_i{}^{\tau'_i}$ for some $\mathtt{x}_i$ and τ'_i; and for every i, $s_i \leadsto s_{i+1} \in \Delta$.

Property 3 *For every instrumented tree, and for every rule in this tree* $\tau, M_\tau, d_\tau, \Delta_\tau, E_\tau, H_\tau, t \to \tau', M_{\tau'}, d_{\tau'}, \Delta_{\tau'}, r$ *where* $\mathtt{st}(r) = E_{\tau'}, H_{\tau'}$*, we have: for every locations* l, l' *and field* `f` *such that* $H_\tau[l'][\mathtt{f}] = l$*, then* $l \leadsto^*_{\Delta_\tau} l'.\mathtt{f}$*; for every locations* l, l' *and field* `f` *such that* $H_{\tau'}[l'][\mathtt{f}] = l$*, then* $l \leadsto^*_{\Delta_{\tau'}} l'.\mathtt{f}$*.*

3.6. Annotated Semantics in Coq

In the Coq development, we distinguish expressions from statements, and we define the reduction $\to$ as two Coq predicates: `red_expr` and `red_stat`. The first predicate has type `environment → heap → ext_expr → out_expr → Type` (and similarly for the statement reduction). The construction `ext_expr` refers to the extended syntax for expressions e_e. The inductive type `out_expr` is defined as being either the result of a terminating evaluation, containing a new environment, heap, and returned value, or an aborted evaluation, containing a new environment and heap.

```
1 Inductive out_expr :=
2   | out_expr_ter : environment → heap_o → value → out_expr
3   | out_expr_error : environment → heap_o → out_expr.
```

To ease the instrumentation, we directly add the annotations in the semantics: each rule of the semantics takes two additional arguments: the left-hand side annotation and the right-hand side annotation. However, there is no restriction on these annotations, we rely on the correctness properties of Section 3.5 to ensure they define the property of interest.

The semantics is thus parameterized by four types, corresponding to the left and right annotations for expressions and statements. These types are wrapped in a Coq record and used through projections such as `annot_e_l` (for left-hand-side annotations in expressions).

Figure 8 shows the rule for variables from this annotated semantics, where `ext_expr_expr` corresponds to the injection of expressions into extended expressions. The additional annotation arguments of type `annot_e_l` and `annot_e_r` are carried by every rule. As every rule contains such annotations, it is easy to write a function `extract_anot` taking such a derivation tree and returning the corresponding annotations. Every part of the Coq development that uses the reduction $\to$ but not the annotations (such as the interpreter) uses trivial annotations of unit type.

The annotations are then incrementally computed using Coq functions. Each of the new annotating passes takes the result of the previous pass as an argument to add its new annotations. The initial annotation is the trivial one, where every annotating types are unit. The definition of

$$\frac{E[\mathtt{x}] = v \qquad M[\mathtt{x}] = \tau_0}{\tau, M, d, E, H, \underline{\mathtt{x}} \to \tau', M, d \cup \{\mathtt{x}^{\tau_0}\}, E, H, v}\ \textsc{Var} \qquad \frac{H[l] = \bot \qquad H' = H[l \mapsto \underline{\{\}}] \qquad M' = M[l \mapsto \tau']}{\tau, M, d, E, H, \underline{\{\}} \to \tau', M, d \cup \left\{l^{\tau'}\right\}, E, H', l}\ \textsc{Obj}$$

$$\frac{H'[l] = o \qquad o[\mathtt{f}] = v \qquad M[(l, M[l], f)] = \tau_0}{\tau, M, d, E, H, \underline{(E', H', l)\ ._1 \mathtt{f}} \to \tau', M, d \cup \{(l, M[l], f)^{\tau_0}\}, E', H', v}\ \textsc{Fld1}$$

$$\frac{\tau_1, M, \emptyset, S, \underline{e} \to \tau_2, M', d, r \qquad \tau_3, M', \emptyset, S, \underline{\mathtt{if1}(r, s_1, s_2)} \to \tau_4, M'', \emptyset, r'}{\tau_0, M, \emptyset, S, \underline{\mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2} \to \tau_5, M'', \emptyset, r'}\ \textsc{If}$$

$$\frac{\tau_1, M, \emptyset, S, e \to \tau_2, M', d, r \qquad \tau_3, M', \emptyset, S, \underline{\mathtt{while1}(r, x, s)} \to \tau_4, M'', \emptyset, r'}{\tau_0, M, \emptyset, S, \underline{\mathtt{while}\ e\ \mathtt{do}\ s} \to \tau_5, M'', \emptyset, r'}\ \textsc{While}$$

$$\frac{\tau_1, M, \emptyset, S, e \to \tau_2, M', d, r \qquad \tau_3, M', d, S, \underline{\mathtt{x} =_1 r} \to \tau_4, M'', \emptyset, r'}{\tau_0, M, \emptyset, S, \underline{\mathtt{x} = e} \to \tau_5, M'', \emptyset, r'}\ \textsc{Asg}$$

$$\frac{E' = E[\mathtt{x} \mapsto v] \qquad M' = M[\mathtt{x} \mapsto \tau']}{\tau, M, d, S, \underline{\mathtt{x} =_1 (E, H, v)} \to \tau', M', \emptyset, E', H}\ \textsc{Asg1}$$

$$\frac{\tau_1, M, \emptyset, S, e_1 \to \tau_2, M', d, r \qquad \tau_3, M', \emptyset, S, \underline{r.\mathtt{f} =_1 e_2} \to \tau_4, M'', \emptyset, r'}{\tau_0, M, \emptyset, S, \underline{e_1.\mathtt{f} = e_2} \to \tau_5, M'', \emptyset, r'}\ \textsc{FldAsg}$$

$$\frac{\tau_1, M, \emptyset, S', e \to \tau_2, M', d, r \qquad \tau_3, M', d, S', \underline{l.\mathtt{f} =_2 r} \to \tau_4, M'', \emptyset, r'}{\tau_0, M, \emptyset, S, \underline{(S', x).\mathtt{f} =_1 e} \to \tau_5, M'', \emptyset, r'}\ \textsc{FldAsg1}$$

$$\frac{H[l] = o \qquad o' = o\,[\mathtt{f} \mapsto v] \qquad H' = H'\,[l \mapsto o'] \qquad M' = M[(l, M[l], f) \mapsto \tau']}{\tau, M, d, S, \underline{l.\mathtt{f} =_2 (E, H, v)} \to \tau', M', \emptyset, E, H'}\ \textsc{FldAsg2}$$

$$\frac{\tau_1, M, \emptyset, S, e \to \tau_2, M', d, r \qquad \tau_3, M', \emptyset, S, \underline{\mathtt{delete1}\ r.\mathtt{f}} \to \tau_4, M'', \emptyset, r'}{\tau_0, M, \emptyset, S, \underline{\mathtt{delete}\ e.\mathtt{f}} \to \tau_5, M'', \emptyset, r'}\ \textsc{Delete}$$

Figure 6: Rules for Dependencies Annotations

annotations in our Coq development exactly follows the scheme presented in Section 3.2. This allows to only specify the parts of the analysis that effectively change their annotations, using a pattern matching construction ending with a Coq's wild card _ to deal with all the cases that just propagate the annotations. It has been written in a modular way, which is robust to changes. For example, a previous version of the annotations only modified partial traces on one side of the rules. As all the following annotating passes treat the traces as an abstract object whose type is parameterized, it was straightforward to update the Coq development to change traces on both sides of each rule.

Figure 9 shows the introduction of the last-modified annotation (see Figure 3c and 4c). This annotation is parameterized by another (traces for instance) here called `Locations`. In Coq, the heap M of Section 3.4 is represented by the record `LastChangeHeaps` defined on Line 4. Line 9 then states it is the left and right annotation types of this annotation. Next is the pattern matching defining the *axiom* rule for statement, and in particular the case of the assignment Line 17 which, as in Figure 4c, stores the current location τ in the annotation. Line 31 sums up the rules, stating that every rule

$$\frac{E' = E[\mathtt{x} \mapsto v] \qquad M' = M[\mathtt{x} \mapsto \tau']}{\tau, M, d, \Delta, S, \underline{\mathtt{x} =_1 (E, H, v)} \to \tau', M', \emptyset, \left\{\delta \leftsquigarrow \mathtt{x}^{\tau'} | \delta \in d\right\} \cup \Delta, E', H} \textsc{Asg1}$$

$$\frac{H[l] = o \qquad o' = o[\mathtt{f} \mapsto v] \qquad H' = H'[l \mapsto o'] \qquad M' = M[(l, M[l], f) \mapsto \tau']}{\tau, M, d, \Delta, S, \underline{l.\mathtt{f} =_2 (E, H, v)} \to \tau', M', \emptyset, \left\{\delta \leftsquigarrow (l, M[l], f)^{\tau'} | \delta \in d\right\} \cup \Delta, E, H'} \textsc{FldAsg2}$$

Figure 7: Rules for Annotating Dependencies of Statements

```
Inductive red_expr : environment → heap_o → ext_expr → out_expr → Type :=
  (* ... *)
  | red_expr_expr_var : annot_e_l Annots → annot_e_r Annots →
    ∀E H x v, getvalue E x v → red_expr E H (ext_expr_expr (expr_var x)) (out_expr E H v)
```

Figure 8: A Semantic Rule as Written in Coq

of this annotation just propagates their arguments, except the *axiom* rule for statements. As can be seen, the corresponding code is fairly short.

We have also defined an interpreter `run_expr : nat → environment → heap_o → expr → option out` taking as arguments an integer, an environment, a heap, and an expression and returning an output. The presence of a `while` in O'While allows the existence of non-terminating executions, whereas every Coq function must be terminating. To bypass this mismatch, the interpreters `run_expr` and `run_stat` (respectively running over expressions and statements) take an integer (the first argument of type `nat` above), called *fuel*. At each recursive call, this fuel is decremented, the interpreter giving up and returning `None` once it reaches 0. We have proven the interpreter is correct related to the semantics, and we have extracted it as an OCaml program using the Coq extraction mechanism.

4. Dependency Analysis

The annotating process makes the property we want to track appear explicitly in derivation trees. Our next step is to define an abstraction of the semantics for computing safe approximations of these properties, and to prove its correctness with respect to the instrumented semantics.

4.1. Abstract Domains

The analysis is expressed as a reduction relation operating over abstractions of the concrete semantic domains. The notion of program point will play a central role, as program points are used both in the abstraction of points of allocation and points of modification. This analysis thus uses the set PP of program points, so we assume that the input program is a result of Function Π defined in Appendix A.1. Property 4, defined in Appendix A.3, ensures that the added program points are correct with respect to the associated traces, which are used to name objects, and thus that this abstraction is sound. To avoid burdening notations, program points are only shown when needed.

Values are defined to be either basic values or locations. Regarding locations, we use the standard abstraction in which object locations are abstracted by the program points corresponding to the instruction that allocated the object. We abstract basic values, which are booleans in our setting, using a lattice $Bool^\sharp$. Thus $l^\sharp \in Loc^\sharp = \mathcal{P}(PP)$ and $v^\sharp \in Val^\sharp = Loc^\sharp + Bool^\sharp$. We define $v^\sharp_l$ as being

```
Variable Locations : Annotations.

Definition ModifiedAnnots := annot_s_r Locations.
Record LastModifiedHeaps : Type :=
  makeLastModifiedHeaps {
    LCEnvironment : heap var ModifiedAnnots;
    LCHeap : heap loc (heap prop_name ModifiedAnnots)}.

Definition LastModified := ConstantAnnotations LastModifiedHeaps.

Definition LastModifiedAxiom_s (r : LastModifiedHeaps)
  E H t o (R : red_stat Locations E H t o) :=
  let LCE := LCEnvironment r in
  let LCH := LCHeap r in
  let (_, tau) := extract_annot_s R in
  match R with
  | red_stat_ext_stat_assign_1 _ _ _ _ _ _ _ x _ _ ⇒
    let LCE' := write LCE x tau
    in makeLastModifiedHeaps LCE' LCH
  | red_stat_stat_delete _ _ _ _ _ l _ f _ _ _ _ _ _ ⇒
    let aob := read LCH l
    in let LCH' := write LCH l (write aob f tau)
    in makeLastModifiedHeaps LCE LCH'
  | red_stat_ext_stat_set_2 _ _ _ _ _ _ _ _ l _ f _ _ _ _ _ _ ⇒
    let aob := read LCH l
    in let LCH' := write LCH l (write aob f tau)
    in makeLastModifiedHeaps LCE LCH'
  | _ ⇒makeLastModifiedHeaps LCE LCH
  end.

Definition annotLastModified :=
  makeIterativeAnnotations LastModified
    (init_e Transmit) (axiom_e Transmit) (up_e Transmit) (down_e Transmit) (next_e Transmit)
    (up_s_e Transmit) (next_e_s Transmit)
    (init_s Transmit) LastModifiedAxiom_s (up_s Transmit) (down_s Transmit) (next_s Transmit).
```

Figure 9: Coq Definitions of the Last-Modified Annotation

either $v^\sharp$ if $v^\sharp \in Loc^\sharp$ or as $\emptyset$ otherwise.

For objects stored at heap locations, we keep trace of the values that the fields may reference. As with annotations, we record the last place each variable and field has been modified. Environments and heaps are thus abstracted as follows: $E^\sharp \in Env^\sharp = Var \to (\mathcal{P}(PP) \times Val^\sharp)$ maps variables to abstract values $v^\sharp$; $H^\sharp \in Heap^\sharp = Loc^\sharp \to Field \to (\mathcal{P}(PP) \times Val^\sharp)$ maps abstract locations to object abstractions (that map fields to abstract values), also storing their last place(s) of modification. Note that we shall freely use curryfied version of $H^\sharp \in Heap^\sharp$.

The two abstract domains inherit a lattice structure in the canonical way as monotone maps, ordered pointwise. The abstract heaps $H^\sharp$ map abstract locations $Loc^\sharp$ (sets of program points) to abstract object, but as locations are abstracted by sets, each write of a value $v^\sharp$ into an abstract heap at abstract location $l^\sharp$ implicitly yields a join between $v^\sharp$ and every value associated to an $l'^\sharp \sqsubseteq l^\sharp$. In practice, such an abstract heap $H^\sharp$ is implemented by a map from program points to abstract values and those writes yield joins with every $p \in l^\sharp$.

We abstract $l^\tau \in ALoc$ by the program point that allocated l^τ, and the traces by program points

(using $\prec$). We can thus abstract the relation $\Delta_\tau \in \mathit{Dep}$ (and the relation $\leadsto$) by making the natural abstraction $\Delta^\sharp$ of those definitions: $\Delta^\sharp \in \mathit{Dep}^\sharp = \mathcal{P}\left(\mathit{Source}^\sharp \times \mathit{Store}^\sharp\right)$; $\mathit{ALoc}^\sharp = \mathit{PP}$; $\mathit{Store}^\sharp = (\mathit{Var} \times \mathit{PP}) + (\mathit{PP} \times \mathit{Field} \times \mathit{PP})$; and $d^\sharp \in \mathit{Source}^\sharp = \mathit{PP} + \mathit{Store}^\sharp$. Abstract flows are written using the symbol $\leadsto^\sharp$. To avoid confusion, program points $p \in \mathit{PP}$ interpreted as elements of $\mathit{Source}^\sharp$ (thus representing locations) are written o^p.

Abstract flows are thus usual flows in which all traces have been replaced by program points. We've seen in Section 3.1 that there exists an abstraction relation $\prec$ between traces and program points such that $\tau \prec p$ if and only if p corresponds to the trace τ. This relation can be directly extended to Dep and $\mathit{Dep}^\sharp$: for instance for each $\mathtt{x}^\tau \in \mathit{Var} \times \mathit{Trace} \subset \mathit{Store}$ such that $\tau \prec p$, we have $\mathtt{x}^\tau \prec \mathtt{x}^p \in \mathit{Store}^\sharp$. Similarly, this relation $\prec$ can also be defined over Val and $\mathit{Val}^\sharp$, Env and $\mathit{Env}^\sharp$, and Heap and $\mathit{Heap}^\sharp$.

4.2. Abstract Reduction Relation

We formalize the analysis as an abstract reduction relation $\rightarrow^\sharp$ for expressions and statements: $E^\sharp, H^\sharp, \underline{s} \rightarrow^\sharp E'^\sharp, H'^\sharp, \Delta^\sharp$ and $E^\sharp, H^\sharp, \underline{e} \rightarrow^\sharp v^\sharp, d^\sharp$. On statements, the analysis returns an abstract environment, an abstract heap, and a partial dependency relation. On expressions, it returns the set of all its possible locations and the set of its dependencies. The analysis is correct if, for all statements, the result of the abstract reduction relation is a correct abstraction of the instrumented reduction. More precisely, the analysis is correct if for each statement $\underline{\mathtt{s}}$ such that

$$\tau, M_\tau, d_\tau, \Delta_\tau, E_\tau, H_\tau, \underline{s} \rightarrow \tau', M_{\tau'}, d_{\tau'}, \Delta_{\tau'}, E_{\tau'}, H_{\tau'} \quad \text{and} \quad E^\sharp, H^\sharp, \underline{s} \rightarrow^\sharp E'^\sharp, H'^\sharp, \Delta^\sharp$$

where M_τ is chosen accordingly to E_τ and H_τ, $E_\tau \prec E^\sharp$, and $H_\tau \prec H^\sharp$, we have $\Delta_{\tau'} \prec \Delta^\sharp$. In other words, the analysis captures at least all the real flows, defined by the annotations.

Figure 10 shows the rules of this analysis. To avoid burdening notations, we denote by $d^\sharp \leadsto f$ the abstract dependency relation $\left\{(f_d, f) \middle| f_d \in d^\sharp\right\}$. Following the same scheme, we freely use the notation $l^\sharp.\mathtt{f}^p$ to denote the set $\left\{p_0.\mathtt{f}^p \middle| p_0 \in l^\sharp\right\}$. As an example, here is the rule for assignments:

$$\frac{E^\sharp, H^\sharp, \underline{e} \rightarrow^\sharp v^\sharp, d^\sharp}{E^\sharp, H^\sharp, \underline{\mathtt{x}^p = e} \rightarrow^\sharp E^\sharp\left[\mathtt{x} \mapsto \left(\{p\}, v^\sharp\right)\right], H^\sharp, \left(v_l^\sharp \cup d^\sharp\right) \leadsto^\sharp \mathtt{x}^p} \textsc{Asg}$$

This rule expresses that when encountering an assignment, an over-approximation of all the possible locations in the form of an abstract value $v^\sharp$ and of the dependencies $d^\sharp$ of the assigned expression $\underline{e}$ is computed. The abstract environment is then updated by setting the variable $\mathtt{x}$ to this new abstract value. Every possible flow from a potential dependency $\mathtt{y} \in d^\sharp$ or possible location value $v_l^\sharp$ of the expression $\underline{e}$ is marked as flowing into $\mathtt{x}$. The position of $\mathtt{x}$ is taken into account in the resulting flows.

The Bin rule makes use of an abstract operation $\mathtt{op}^\sharp$, which depends on the operators added in the language. Figure 16 in Appendix B shows an example of analysis on the code we have seen on the previous sections, namely `x = {}; x.f = {}; if false then y = x.f else y = {}`.

There are several possible variations and extensions of this analysis. For one notable example it could be refined with strong updates on locations. For the moment, we leave for further work how exactly to annotate the semantics and to abstract locations in order to state whether or not an abstract location represents a unique concrete location in the heap.

4.3. Analysis in Coq

The abstract domains are essentially the same as the ones described in Section 4.1. They are straightforward to formalize as soon as basic constructions for lattices are available: the abstract domains are just specific instances of standard lattices from abstract interpretation (flat lattices,

$$\frac{}{E^\sharp, H^\sharp, \underline{\texttt{skip}} \to^\sharp E^\sharp, H^\sharp, \emptyset}\ \textsc{Skip} \qquad \frac{E^\sharp, H^\sharp, \underline{s_1} \to^\sharp E_1^\sharp, H_1^\sharp, \Delta_1^\sharp \qquad E_1^\sharp, H_1^\sharp, \underline{s_2} \to^\sharp E_2^\sharp, H_2^\sharp, \Delta_2^\sharp}{E^\sharp, H^\sharp, \underline{s_1; s_2} \to^\sharp E_2^\sharp, H_2^\sharp, \Delta_1^\sharp \cup \Delta_2^\sharp}\ \textsc{Seq}$$

$$\frac{E^\sharp, H^\sharp, \underline{s_1} \to^\sharp E_1^\sharp, H_1^\sharp, \Delta_1^\sharp \qquad E^\sharp, H^\sharp, \underline{s_2} \to^\sharp E_2^\sharp, H_2^\sharp, \Delta_2^\sharp}{E^\sharp, H^\sharp, \underline{\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2} \to^\sharp E_1^\sharp \sqcup E_2^\sharp, H_1^\sharp \sqcup H_2^\sharp, \Delta_1^\sharp \cup \Delta_2^\sharp}\ \textsc{If}$$

$$\frac{E^\sharp \sqsubseteq E_0^\sharp \qquad H^\sharp \sqsubseteq H_0^\sharp \qquad E_0^\sharp, H_0^\sharp, \underline{s} \to^\sharp E_1^\sharp, H_1^\sharp, \Delta^\sharp \qquad E_1^\sharp \sqsubseteq E_0^\sharp \qquad H_1^\sharp \sqsubseteq H_0^\sharp}{E^\sharp, H^\sharp, \underline{\texttt{while } e \texttt{ do } s} \to^\sharp E_0^\sharp, H_0^\sharp, \Delta^\sharp}\ \textsc{While}$$

$$\frac{E^\sharp, H^\sharp, \underline{e} \to^\sharp v^\sharp, d^\sharp}{E^\sharp, H^\sharp, \underline{\mathtt{x}^p \texttt{ = } e} \to^\sharp E^\sharp \left[\mathtt{x} \mapsto \left(\{p\}, v^\sharp\right)\right], H^\sharp, \left(v_l^\sharp \cup d^\sharp\right) \leftsquigarrow^\sharp \mathtt{x}^p}\ \textsc{Asg}$$

$$\frac{E^\sharp, H^\sharp, \underline{e_1} \to^\sharp v_1^\sharp, d_1^\sharp \qquad E^\sharp, H^\sharp, \underline{e_2} \to^\sharp v_2^\sharp, d_2^\sharp}{E^\sharp, H^\sharp, \underline{e_1.\mathtt{f}^p \texttt{ = } e_2} \to^\sharp E^\sharp, H^\sharp \sqcup \left\{\left(v_1{}_l^\sharp, \mathtt{f}\right) \mapsto \left(\{p\}, v_2^\sharp\right)\right\}, \left(v_2{}_l^\sharp \cup d_1^\sharp \cup d_2^\sharp\right) \leftsquigarrow^\sharp v_1{}_l^\sharp.\mathtt{f}^p}\ \textsc{FldAsg}$$

$$\frac{E^\sharp, H^\sharp, \underline{e} \to^\sharp v^\sharp, d^\sharp \qquad H^\sharp[v_l^\sharp] \sqsubseteq o^\sharp \qquad o^\sharp[\mathtt{f}] = \left(p_0, v_f^\sharp\right)}{E^\sharp, H^\sharp, \underline{\texttt{delete } e.\mathtt{f}} \to^\sharp E^\sharp, H^\sharp, d^\sharp \leftsquigarrow^\sharp v_l^\sharp.\mathtt{f}^{p_0}}\ \textsc{Del} \qquad \frac{}{E^\sharp, H^\sharp, \underline{c} \to^\sharp c^\sharp, \emptyset}\ \textsc{Cst}$$

$$\frac{E^\sharp[\mathtt{x}] = \left(p_0, v^\sharp\right)}{E^\sharp, H^\sharp, \underline{\mathtt{x}^p} \to^\sharp v^\sharp, \{\mathtt{x}^{p_0}\}}\ \textsc{Var} \qquad \frac{E^\sharp, H^\sharp, e_1 \to^\sharp v_1^\sharp, d_1^\sharp \qquad E^\sharp, H^\sharp, e_2 \to^\sharp v_2^\sharp, d_2^\sharp}{E^\sharp, H^\sharp, \underline{e_1 \texttt{ op } e_2} \to^\sharp v_1^\sharp \texttt{ op}^\sharp v_2^\sharp, d_1^\sharp \cup d_2^\sharp}\ \textsc{Bin}$$

$$\frac{}{E^\sharp, H^\sharp, \underline{\{\}^p} \to^\sharp \{p\}, \{o^p\}}\ \textsc{Obj} \qquad \frac{E^\sharp, H^\sharp, \underline{e} \to^\sharp v^\sharp, d^\sharp \qquad H^\sharp[v_l^\sharp] \sqsubseteq o^\sharp \qquad o^\sharp[\mathtt{f}] = \left(p_0, v_f^\sharp\right)}{E^\sharp, H^\sharp, \underline{e.\mathtt{f}^p} \to^\sharp v_f^\sharp, \left(v_l^\sharp.\mathtt{f}^{p_0}\right) \cup d^\sharp}\ \textsc{Fld}$$

Figure 10: Rules for the Abstract Reduction Relation

power set lattices...). For the certification of lattices we refer to the Coq developments by David Pichardie [10].

Similarly to Section 3.6, the rules of the analyzer presented in Figure 10 are first defined as an inductive predicate of type

```
t AEnvironment → t AHeap → stat → t AEnvironment → t AHeap → t AFlows → Prop
```

where the two types `t AEnvironment` and `t AHeap` are the types of the abstract lattices for environments and heaps, and `t AFlows` the type of abstract flows, represented as a lattice for convenience. The analyzer is then defined by an extractable function of similar type (excepting the final "→ `Prop`"), the two definitions being proven equivalent. The situation for expressions is similar.

Once the analysis has been defined as well as the instrumentation, it's possible to formally prove the correctness of the abstract reduction rules with respect to the instrumentation. The property to prove is the one shown in Section 4.2: if from an empty heap, a program reduces to a heap E_τ, H_τ and flows Δ_τ, then if from the $\bot$ abstraction, a program reduces to $E^\sharp$, $H^\sharp$ and abstract flows $\Delta^\sharp$; that is, $[], \emptyset, [], \emptyset, \emptyset, \underline{s} \to \tau, M_\tau, \Delta_\tau, E_\tau, H_\tau$ and $\bot, \bot, \underline{s} \to^\sharp E^\sharp, H^\sharp, \Delta^\sharp$, then $E \prec E^\sharp$, $H \prec H^\sharp$ and $\Delta_\tau \prec \Delta^\sharp$. This is shown in [2].

5. Conclusion

Schmidt's natural semantics-based abstract interpretation is a rich framework which can be instantiated in a number of ways. In this paper, we have shown how the framework can be applied to the particular style of natural semantics called pretty big step semantics. We have studied a particular kind of intentional information about the program execution, *viz.*, how information flows from points of creation to points of use. This has lead us to define a particular abstraction of semantic derivation trees for describing points in the execution. This abstraction can then be further combined with other abstractions to obtain an abstract reduction relation that formalizes the static analysis.

Other systematic derivation of static analyses have taken small-step operational semantics as starting point. Cousot [4] has shown how to systematically derive static analyses for an imperative language using the principles of abstract interpretation. Midtgaard and Jensen [8, 9] 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 [14] 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. Our motivation for developing the big-step approach further is that the semantic framework has certain modularity properties that makes it a popular choice for formalizing real-sized programming languages.

Our preliminary experiments show that the semantics and its abstractions lend themselves well to being implemented in the Coq proof assistant. This is an important point, as some form of mechanization is required to evaluate the scalability of the method. Scalability is indeed one of the goals for this work. The present paper establishes the principles with which we hope to achieve the generation of an analysis for full JavaScript based on its Coq formalization. However, this will require some form of machine-assistance in the production of the abstract semantics. The present work provides a first experience of how to proceed. Further work will now have to extract the essence of this process and investigate how to program it in Coq.

One this has been achieved, we will be well armed to attack other analyses. One immediate candidate for further work is full information flow analysis, taking indirect flows due to conditionals into account. It would in particular be interesting to see if the resulting abstract semantics can be used for a rational reconstruction of the semantic foundations underlying the dynamic and hybrid information flow analysis techniques developed by Le Guernic, Banerjee, Schmidt and Jensen [7]. Combined with the extension to full JavaScript, this would provide a certified version of the recent information flow control mechanisms for JavaScript such as the monitor proposed by Hedin and Sabelfeld [6].

Bibliographie

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

[2] M. Bodin, T. Jensen, and A. Schmitt. Pretty-big-step certified abstract interpretation, coq source code. `http://www.irisa.fr/celtique/aschmitt/research/owhileflows/`, 2013.

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

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

[5] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable javascript. In *ISSTA 2011*, pages 177–187. ACM Press, 2011.

[6] D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *Proc. of the 25th Computer Security Foundations Symp. (CSF'12)*, pages 3–18. IEEE, 2012.

[7] G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based Confidentiality Monitoring. In *ASIAN 2006*, pages 75–89. Springer LNCS vol. 4435, 2006.

[8] J. Midtgaard and T. Jensen. A calculational approach to control-flow analysis by abstract interpretation. In *SAS 2008*, volume 5079 of *LNCS*, pages 347–362. Springer Verlag, 2008.

[9] J. Midtgaard and T. Jensen. Control-flow analysis of function calls and returns by abstract interpretation. In *ICFP 2009*, pages 287–298. ACM, 2009.

[10] D. Pichardie. Building certified static analysers by modular construction of well-founded lattices. In *FICS 2008*, volume 212 of *ENTCS*, pages 225–239, 2008.

[11] D. Schmidt. Natural-semantics-based abstract interpretation (preliminary version). In *Proc. 2d Static Analysis Symposium (SAS'95)*, pages 1–18. Springer LNCS vol. 983, 1995.

[12] E. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *S&P*, 2010.

[13] D. L. M. Valérie Gouranton. Dynamic slicing: a generic analysis based on a natural semantics format. *Journal of Logic and Computation*, 9(6), 1999.

[14] D. Van Horn and M. Might. Abstracting abstract machines. In *ICFP*, pages 51–62. ACM, 2010.

[15] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, volume 42, 2007.

A. Traces and Program Points

In this section we define how to add program points to programs, in order to identify syntactic positions in the program, and we show how to relate traces from a semantic derivation with program points.

A.1. Program Points

Program points are defined as chains of atoms. The transformation Π described below takes a program point and a term, and annotates each sub-term with program points before and after the sub-term. The program point before a syntactic construct is a *context*, a chain of atoms indicating where to find the term in the initial program. The program point after is a context followed by the atom identifying the syntactic construct corresponding to the sub-term. For instance, the program point Seq2/Seq2/IfE refers to the point before `false` in the term `x = {}; x.f = {}; if false then y = x.f else y = {};` and Seq2/Seq2/IfE/Cst to the point after. The notion of "before" and "after" a program point is standard in data flow analysis, but is here given a semantics-based definition.

We write $\cdot$ for the empty program point, and we assume program points have a monoid structure, with / as concatenation and $\cdot$ as neutral element.

$$
\begin{aligned}
\Pi(\mathrm{PP}, \texttt{skip}) &= \mathrm{PP}, \texttt{skip}, \mathrm{PP/Skip} \\
\Pi(\mathrm{PP}, s_1; s_2) &= \mathrm{PP}, \Pi(\mathrm{PP/Seq1}, s_1); \Pi(\mathrm{PP/Seq2}, s_2), \mathrm{PP/Seq} \\
\Pi(\mathrm{PP}, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2) &= \mathrm{PP}, \texttt{if } \Pi(\mathrm{PP/IfE}, e) \texttt{ then } \Pi(\mathrm{PP/IfT}, s_1) \texttt{ else } \Pi(\mathrm{PP/IfF}, s_2), \mathrm{PP/If} \\
\Pi(\mathrm{PP}, \texttt{while } e \texttt{ do } s) &= \mathrm{PP}, \texttt{while } \Pi(\mathrm{PP/WhileE}, e) \texttt{ do } \Pi(\mathrm{PP/WhileS}, s), \mathrm{PP/While} \\
\Pi(\mathrm{PP}, \texttt{x = } e) &= \mathrm{PP}, \texttt{x = } \Pi(\mathrm{PP/AsgE}, e), \mathrm{PP/Asg} \\
\Pi(\mathrm{PP}, e_1\texttt{.f = } e_2) &= \mathrm{PP}, \Pi(\mathrm{PP/FldAsgL}, e_1)\texttt{.f = } \Pi(\mathrm{PP/FldAsgV}, e_2), \mathrm{PP/FldAsg} \\
\Pi(\mathrm{PP}, \texttt{delete } e\texttt{.f}) &= \mathrm{PP}, \texttt{delete } \Pi(\mathrm{PP/DelE}, e)\texttt{.f}, \mathrm{PP/Del} \\
\Pi(\mathrm{PP}, c) &= \mathrm{PP}, c, \mathrm{PP/Cst} \\
\Pi(\mathrm{PP}, \texttt{x}) &= \mathrm{PP}, \texttt{x}, \mathrm{PP/Var} \\
\Pi(\mathrm{PP}, e_1 \texttt{ op } e_2) &= \mathrm{PP}, \Pi(\mathrm{PP/BinL}, e_1) \texttt{ op } \Pi(\mathrm{PP/BinR}, e_2), \mathrm{PP/Bin} \\
\Pi(\mathrm{PP}, \texttt{\{\}}) &= \mathrm{PP}, \texttt{\{\}}, \mathrm{PP/Obj} \\
\Pi(\mathrm{PP}, e\texttt{.f}) &= \mathrm{PP}, \Pi(\mathrm{PP/FldE}, e)\texttt{.f}, \mathrm{PP/Fld}
\end{aligned}
$$

To derive program points from traces, we first need to define an operator that deletes atoms in a chain up to a given atom. More precisely, the operator $\mathrm{PP} \downarrow_{\textsc{Name}}$ removes the shortest suffix of PP that starts with Name, included. Formally, it is defined as follows.

$$
\begin{aligned}
\cdot \downarrow_{\textsc{Name}} &= \cdot \\
\mathrm{PP/Name} \downarrow_{\textsc{Name}} &= \mathrm{PP} \\
\mathrm{PP/Name'} \downarrow_{\textsc{Name}} &= \mathrm{PP} \downarrow_{\textsc{Name}} \quad \text{if Name'} \neq \text{Name}
\end{aligned}
$$

A.2. Traces

We give in Figure 11 the names that are added to the trace for each rule, following the general annotation scheme of Section 3.2. In the *next* step, the annotation of the left of the current rule is ignored, only the annotation on the right of the first premise is used.

Rule	*axiom*	*up*	*next*	*down*
Skip	Skip			
Seq		Seq1	$\overline{\text{Seq1}}$	Seq
Seq1		Seq2		$\overline{\text{Seq2}}$
If		IfE	$\overline{\text{IfE}}$	If
IfTrue		IfT		$\overline{\text{IfT}}$
IfFalse		IfF		$\overline{\text{IfF}}$
While		WhE	$\overline{\text{WhE}}$	While
WhileTrue1		WhS	$\overline{\text{WhS}}$	WhT1
WhileTrue2		WhL		WhT2
WhileFalse	WhF			
Asg		AsgE	$\overline{\text{AsgE}}$	Asg
Asg1	Asg1			
FldAsg		FldAsgL	$\overline{\text{FldAsgL}}$	FldAsg
FldAsg1		FldAsgV	$\overline{\text{FldAsgV}}$	FldAsg1
FldAsg2	FldAsg2			
Del		DelE	$\overline{\text{DelE}}$	Del
Del1	Del1			
Cst	Cst			
Var	Var			
Bin		BinL	$\overline{\text{BinL}}$	Bin
Bin1		BinR	$\overline{\text{BinR}}$	Bin1
Bin2	Bin2			
Obj	Obj			
Fld		FldE	$\overline{\text{FldE}}$	Fld
Fld1	Fld1			
Abort	Abort			

Figure 11: Traces Definition

A.3. From Traces to Program Points

We next define a function $\mathcal{T}$ from traces to program points. There are two challenges in doing so. First, traces mention everything that has happened up to the point under consideration. Program points, however, hide everything that is not under the current execution context. We take care of this folding using the deletion operator defined above. The second challenge is to decide what program point to assign for extended statements and expressions, and when to decide that a statement has finished executing. To illustrate this challenge, we consider the case of a while loop, where in the initial environment we have `x` and `y` equal to `true` and z equal to `false`.

The evaluation of the variable `x` only adds Var at the end of the trace. The evaluation of the two assignments appends the following sequence to the trace, which we call τ_s in the following.

$$[\text{Seq1}; \text{AsgE}; \text{Var}; \overline{\text{AsgE}}; \text{Asg1}; \text{Asg}; \overline{\text{Seq1}}; \text{Seq2}; \text{AsgE}; \text{Var}; \overline{\text{AsgE}}; \text{Asg1}; \text{Asg}; \overline{\text{Seq2}}; \text{Seq}]$$

$$\dfrac{\dfrac{S', \texttt{x = y; y = z} \to S'' \qquad \dfrac{\dfrac{S'', \texttt{x} \to S'', \texttt{false} \qquad \textcircled{2}\ S'', \texttt{while1}((S'', \texttt{false}), \texttt{x}, \texttt{x = y; y = z}) \to S''\ \textcircled{1}}{S'', \texttt{while x do x = y; y = z} \to S''\ \textcircled{1}}}{\textcircled{3}\ S', \texttt{while2}(S'', \texttt{x}, \texttt{x = y; y = z}) \to S''\ \textcircled{1}}}{\textcircled{2}\ S', \texttt{while1}((S', \texttt{true}), \texttt{x}, \texttt{x = y; y = z}) \to S''\ \textcircled{1}}$$

$$\dfrac{\dfrac{S', \texttt{x} \to S', \texttt{true} \qquad \vdots}{S', \texttt{while x do x = y; y = z} \to S''\ \textcircled{1}}}{\textcircled{3}\ S, \texttt{while2}(S', \texttt{x}, \texttt{x = y; y = z}) \to S''\ \textcircled{1}}$$

$$\dfrac{S, \texttt{x = y; y = z} \to S' \qquad \vdots}{\textcircled{2}\ S, \texttt{while1}((S, \texttt{true}), \texttt{x}, \texttt{x = y; y = z}) \to S''\ \textcircled{1}}$$

$$\dfrac{S, \texttt{x} \to S, \texttt{true} \qquad \vdots}{S, \texttt{while x do x = y; y = z} \to S''\ \textcircled{1}}$$

Figure 12: Running a While loop

The whole trace at the end of the execution has the following form.

$$[\text{WhE}; \text{Var}; \overline{\text{WhE}}; \text{WhS}] +\!\!+ \tau_s +\!\!+ [\overline{\text{WhS}}; \text{WhL}] +\!\!+ [\text{WhE}; \text{Var}; \overline{\text{WhE}}; \text{WhS}] +\!\!+ \tau_s +\!\!+ [\overline{\text{WhS}}; \text{WhL}] +\!\!+ [\text{WhE}; \text{Var}; \overline{\text{WhE}}; \text{WhF}] +\!\!+ [\text{While}; \text{WhT2}; \text{WhT1}; \text{While}; \text{WhT2}; \text{WhT1}; \text{While}]$$

The program point on the right-hand-side of every $\to S''$ should be While (①), as it corresponds to the end of the program. The program point before every `while1` (②) should be the same on as right after evaluating the condition, namely WhileE/Var. Similarly, the one before `while2` (③) is taken to be the one right after finishing to evaluate the sequence, namelyWhileS/Seq.

Following this intuition, we define in Figure 13 the $\mathcal{T}$ function that takes a trace and an already computed program point, then creates a program point. A quasi invariant is that part of a program point is deleted only if a new atom is added, reflecting the notion that evaluation never goes back in the program syntax. There is a crucial exception, though: when doing a loop for a `while` loop (premise of rule While2), then the program point jumps back to right before the while loop.

We now state that program points can correctly be extracted from traces. To this end, we consider a derivation where the terms contain program points.

Property 4 *Let t be a term and $t' = \Pi(\cdot, t)$. For any occurrence of a rule the form $\tau, S, PP, t, PP' \to \tau', r$ in any annotated derivation tree from t', we have $PP = \mathcal{T}(\tau, \cdot)$, $PP' = \mathcal{T}(\tau', \cdot)$.*

$$
\begin{aligned}
\mathcal{T}([], \mathrm{PP}) &= \mathrm{PP} \\
\mathcal{T}(\textsc{Skip} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}/\textsc{Skip}) \\
\mathcal{T}(\textsc{Seq1} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}/\textsc{Seq1}) & \mathcal{T}(\overline{\textsc{Seq1}} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\tau, \mathrm{PP}/\textsc{Seq}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{Seq2} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{Seq1}} /\textsc{Seq2}) & \mathcal{T}(\overline{\textsc{Seq2}} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{Seq2}} /\textsc{Seq}) \\
\mathcal{T}(\textsc{IfE} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}/\textsc{IfE}) & \mathcal{T}(\overline{\textsc{IfE}} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{If} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{IfT} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{IfE}} /\textsc{IfT}) & \mathcal{T}(\overline{\textsc{IfT}} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{IfT}} /\textsc{If}) \\
\mathcal{T}(\textsc{IfF} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{IfE}} /\textsc{IfF}) & \mathcal{T}(\overline{\textsc{IfF}} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{IfF}} /\textsc{If}) \\
\mathcal{T}(\textsc{WhE} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}/\textsc{WhE}) & \mathcal{T}(\overline{\textsc{WhE}} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{While} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{WhS} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{WhE}} /\textsc{WhS}) & \mathcal{T}(\overline{\textsc{WhS}} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{WhT1} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{WhL} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{WhS}}) \\
\mathcal{T}(\textsc{WhT2} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{WhF} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}/\textsc{While}) \\
\mathcal{T}(\textsc{AsgE} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}/\textsc{Asge}) & \mathcal{T}(\overline{\textsc{AsgE}} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{Asg} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{Asg1} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{AsgE}} /\textsc{Asg}) \\
\mathcal{T}(\textsc{FldAsgL} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}/\textsc{FldAsgL}) & \mathcal{T}(\overline{\textsc{FldAsgL}} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{FldAsg} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{FldAsgV} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{FldAsgL}} /\textsc{FldAsgV}) & \mathcal{T}(\overline{\textsc{FldAsgV}} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{FldAsg1} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{FldAsg2} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{FldAsgV}} /\textsc{FldAsg}) \\
\mathcal{T}(\textsc{DelE} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}/\textsc{DelE}) & \mathcal{T}(\overline{\textsc{DelE}} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{Del} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{Del1} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{DelE}} /\textsc{Del}) \\
\mathcal{T}(\textsc{Cst} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}/\textsc{Cst}) \\
\mathcal{T}(\textsc{Var} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}/\textsc{Var}) \\
\mathcal{T}(\textsc{BinL} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}/\textsc{BinL}) & \mathcal{T}(\overline{\textsc{BinL}} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{Bin} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{BinR} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{BinL}} /\textsc{BinR}) & \mathcal{T}(\overline{\textsc{BinR}} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{Bin1} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{Bin2} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{BinR}} /\textsc{Bin}) \\
\mathcal{T}(\textsc{Obj} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}/\textsc{Obj}) \\
\mathcal{T}(\textsc{FldE} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}/\textsc{FldE}) & \mathcal{T}(\overline{\textsc{FldE}} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{Fld} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{Fld1} :: \tau, \mathrm{PP}) &= \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{FldE}} /\textsc{Fld}) \\
\mathcal{T}(\textsc{Abort} :: \tau, \mathrm{PP}) &= \mathrm{PP}
\end{aligned}
$$

Figure 13: Traces to Program Points

B. Derivation examples

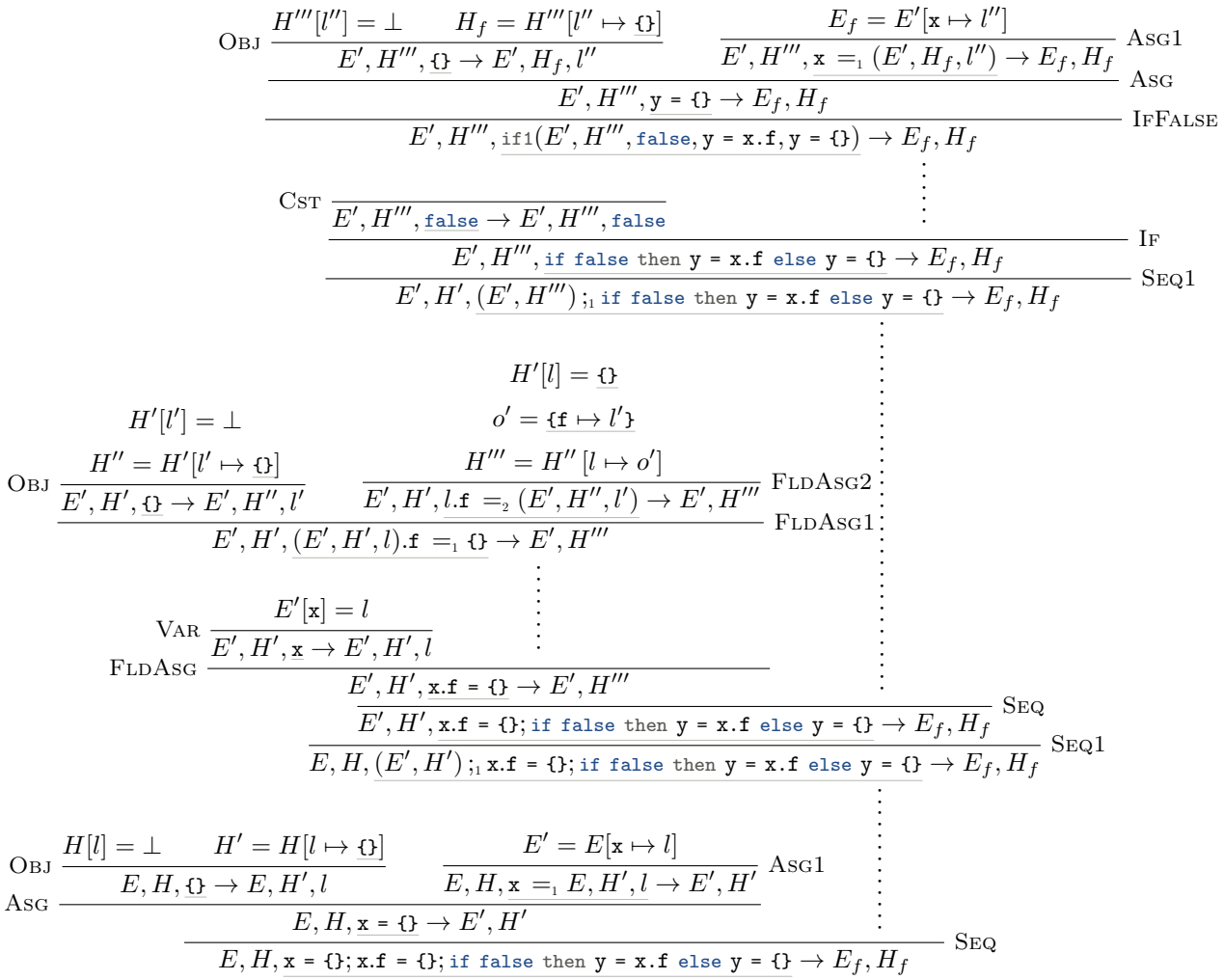


Figure 14: Pretty-big-step derivation

$$\dfrac{\dfrac{\dfrac{}{E, H, \texttt{true} \to E, H, \texttt{true}}\,\textsc{Cst}}{E, H, \texttt{x = true} \to E', H}\,\textsc{Asg} \qquad \dfrac{E' = E[\texttt{x} \mapsto \texttt{true}]}{E, H, \texttt{x} =_1 (E, H, \texttt{true}) \to E', H}\,\textsc{Asg1}}{\dfrac{\vdots}{E, H, \texttt{x = true; y = x} \to E'', H}\,\textsc{Seq}}$$

$$\dfrac{\dfrac{\dfrac{E'[\texttt{x}] = \texttt{true}}{E', H, x \to E', H, \texttt{true}}\,\textsc{Var} \qquad \dfrac{E'' = E'[\texttt{y} \mapsto \texttt{true}]}{E', H, \texttt{x} =_1 E', H, \texttt{true} \to E'', H}\,\textsc{Asg1}}{E', H, \texttt{y = x} \to E'', H}\,\textsc{Asg}}{E, H, (E', H)\,;_1\,\texttt{y = x} \to E'', H}\,\textsc{Seq1}$$

(a) Unannotated Derivation

$$\tau_1 = [] \qquad \tau_2 = [\textsc{Seq1}] \qquad \tau_3 = \tau_2 +\!\!+ [\textsc{AsgE}] \qquad \tau_4 = \tau_3 +\!\!+ [\textsc{Cst}] \qquad \tau_5 = \tau_4 +\!\!+ [\overline{\textsc{AsgE}}]$$

$$\tau_6 = \tau_5 +\!\!+ [\textsc{Asg1}] \qquad \tau_7 = \tau_6 +\!\!+ [\textsc{Asg}] \qquad \tau_8 = \tau_7 +\!\!+ [\overline{\textsc{Seq1}}] \qquad \tau_9 = \tau_8 +\!\!+ [\textsc{Seq2}] \qquad \tau_{10} = \tau_9 +\!\!+ [\textsc{AsgE}]$$

$$\tau_{11} = \tau_{10} +\!\!+ [\textsc{Var}] \qquad \tau_{12} = \tau_{11} +\!\!+ [\overline{\textsc{AsgE}}] \qquad \tau_{13} = \tau_{12} +\!\!+ [\textsc{Asg1}] \qquad \tau_{14} = \tau_{13} +\!\!+ [\textsc{Asg}]$$

$$\tau_{15} = \tau_{14} +\!\!+ [\overline{\textsc{Seq2}}] \qquad \tau_{16} = \tau_{15} +\!\!+ [\textsc{Seq}]$$

$$\dfrac{\dfrac{\dfrac{}{\tau_3, E, H, \texttt{true} \to \tau_4, E, H, \texttt{true}}\,\textsc{Cst}}{\tau_2, E, H, \texttt{x = true} \to \tau_7, E', H}\,\textsc{Asg} \qquad \dfrac{E' = E[\texttt{x} \mapsto \texttt{true}]}{\tau_5, E, H, \texttt{x} =_1 (E, H, \texttt{true}) \to \tau_6, E', H}\,\textsc{Asg1}}{\dfrac{\vdots}{\tau_1, E, H, \texttt{x = true; y = x} \to \tau_{16}, E'', H}\,\textsc{Seq}}$$

$$\dfrac{\dfrac{\dfrac{E'[\texttt{x}] = \texttt{true}}{\tau_{10}, E', H, x \to \tau_{11}, E', H, \texttt{true}}\,\textsc{Var} \qquad \dfrac{E'' = E'[\texttt{y} \mapsto \texttt{true}]}{\tau_{12}, E', H, \texttt{x} =_1 E', H, \texttt{true} \to \tau_{13}, E'', H}\,\textsc{Asg1}}{\tau_9, E', H, \texttt{y = x} \to \tau_{14}, E'', H}\,\textsc{Asg}}{\tau_8, E, H, (E', H)\,;_1\,\texttt{y = x} \to \tau_{15}, E'', H}\,\textsc{Seq1}$$

(b) Derivation Annotated With Traces

Figure 15: Annotating A Simple Derivation

$$E_1^\sharp = \{\mathtt{x} \mapsto (\{p_1\}, \{p_2\})\} \qquad E_2^\sharp = \{\mathtt{x} \mapsto (\{p_1\}, \{p_2\}), \mathtt{y} \mapsto (\{p_6\}, \{p_5\})\}$$

$$E_3^\sharp = \{\mathtt{x} \mapsto (\{p_1\}, \{p_2\}), \mathtt{y} \mapsto (\{p_9\}, \{p_{10}\})\} \qquad E_4^\sharp = \{\mathtt{x} \mapsto (\{p_1\}, \{p_2\}), \mathtt{y} \mapsto (\{p_6, p_9\}, \{p_5, p_{10}\})\}$$

$$H_1^\sharp = \{(p_2, \mathtt{f}) \mapsto (\{p_4\}, \{p_5\})\}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{E_1^\sharp[\mathtt{x}] = (\{p_1\}, \{p_2\})}{E_1^\sharp, H_1^\sharp, \underline{\mathtt{x}^{p_7}} \to^\sharp \{p_2\}, \{\mathtt{x}^{p_1}\}}\textsc{Var} \qquad H_1^\sharp[\{o^{p_2}\}][\mathtt{f}] = (\{p_4\}, \{p_5\})}{E_1^\sharp, H_1^\sharp, \underline{\mathtt{x}^{p_7}.\mathtt{f}^{p_8}} \to^\sharp \{p_5\}, \{o^{p_2}.\mathtt{f}^{p_8}, \mathtt{x}^{p_1}\}}\textsc{Fld} \qquad \vdots}{E_1^\sharp, H_1^\sharp, \underline{\mathtt{y}^{p_6} = \mathtt{x.f}} \to^\sharp E_2^\sharp, H_1^\sharp, \{\{o^{p_5}, o^{p_2}.\mathtt{f}^{p_8}, \mathtt{x}^{p_1}\} \hookleftarrow \mathtt{y}^{p_6}\}}\textsc{Asg} \qquad \dfrac{\dfrac{}{E_1^\sharp, H_1^\sharp, \underline{\{\}^{p_{10}}} \to^\sharp \{p_{10}\}, \{o^{p_{10}}\}}\textsc{Obj}}{E_1^\sharp, H_1^\sharp, \underline{\mathtt{y}^{p_9} = \{\}} \to^\sharp E_3^\sharp, H_1^\sharp, \{o^{p_{10}} \hookleftarrow \mathtt{y}^{p_9}\}}\textsc{Asg}}{E_1^\sharp, H_1^\sharp, \underline{\texttt{if false then y = x.f else y = \{\}}} \to^\sharp E_4^\sharp, H_1^\sharp, \{\{o^{p_5}, o^{p_2}.\mathtt{f}^{p_8}, \mathtt{x}^{p_1}\} \hookleftarrow \mathtt{y}^{p_6}, o^{p_{10}} \hookleftarrow \mathtt{y}^{p_9}\}}\textsc{If}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{E_1^\sharp[\mathtt{x}] = (\{p_1\}, \{p_2\})}{E_1^\sharp, \bot, \underline{\mathtt{x}^{p_3}} \to^\sharp \{p_2\}, \{\mathtt{x}^{p_1}\}}\textsc{Var} \qquad \dfrac{}{E_1^\sharp, \bot, \underline{\{\}^{p_5}} \to^\sharp \{p_5\}, \{o^{p_5}\}}\textsc{Obj}}{E_1^\sharp, \bot, \underline{\mathtt{x}^{p_3}.\mathtt{f}^{p_4} = \{\}^{p_5}} \to^\sharp E_1^\sharp, H_1^\sharp, \{\{\mathtt{x}^{p_1}, o^{p_5}\} \hookleftarrow o^{p_2}.\mathtt{f}^{p_4}\}}\textsc{FldAsg} \qquad \vdots}{E_1^\sharp, \bot, \underline{\texttt{x.f = \{\}; if false then y = x.f else y = \{\}}} \to^\sharp E_4^\sharp, H_1^\sharp, \{\{\mathtt{x}^{p_1}, o^{p_5}\} \hookleftarrow o^{p_2}.\mathtt{f}^{p_4}, \{o^{p_5}, o^{p_2}.\mathtt{f}^{p_8}, \mathtt{x}^{p_1}\} \hookleftarrow \mathtt{y}^{p_6}, o^{p_{10}} \hookleftarrow \mathtt{y}^{p_9}\}}\textsc{Seq}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\bot, \bot, \underline{\{\}^{p_2}} \to^\sharp \{p_2\}, \{o^{p_2}\}}\textsc{Obj}}{\bot, \bot, \underline{\mathtt{x}^{p_1} = \{\}^{p_2}} \to^\sharp E_1^\sharp, \bot, \{o^{p_2} \hookleftarrow \mathtt{x}^{p_1}\}}\textsc{Asg} \qquad \vdots}{\bot, \bot, \underline{\texttt{x = \{\}}^{p_2}\texttt{; x.f = \{\}}^{p_5}\texttt{; if false then y = x.f else y = \{\}}^{p_{10}}} \to^\sharp E_4^\sharp, H_1^\sharp, \left\{\begin{array}{l} o^{p_2} \hookleftarrow \mathtt{x}^{p_1}, \{\mathtt{x}^{p_1}, o^{p_5}\} \hookleftarrow o^{p_2}.\mathtt{f}^{p_4}, \\ \{o^{p_5}, o^{p_2}.\mathtt{f}^{p_8}, \mathtt{x}^{p_1}\} \hookleftarrow \mathtt{y}^{p_6}, o^{p_{10}} \hookleftarrow \mathtt{y}^{p_9} \end{array}\right\}}\textsc{Seq}$$

Figure 16: Analysis Example