Pretty-big-step-semantics-based Certified Abstract Interpretation (Preliminary version)

Martin Bodin
ENS Lyon and Inria

Thomas Jensen
Inria

Alan Schmitt
Inria

We present a technique for deriving semantic program analyses from a natural semantics specification of the programming language. The technique is based on a particular kind of semantics called pretty-big-step semantics. We present a pretty-big-step semantics of a language with simple objects called O'While and specify a series of instrumentations of the semantics that explicitates 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 has been realized with the Coq proof assistant.

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 intensional 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 also 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 "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. We will give a high-level introduction to the main features of pretty-big-step semantics in Section 2

A. Banerjee, O. Danvy, K.-G. Doh, J. Hatcliff (Eds):
David A. Schmidt's 60th Birthday Festschrift
EPTCS 129, 2013, pp. 360–383, doi:10.4204/EPTCS.129.23

© M. Bodin, T. Jensen, & A. Schmitt
This work is licensed under the
Creative Commons Attribution License.

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 3.

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 4 and the analysis itself is defined in Sections 5.

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 will show how the Coq proof assistant has been used successfully to achieve these objectives as we go along.

2 Pretty-Big-Step Operational 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.

We give an intuition on how pretty-big-step semantics works through a simple example: the execution of a while loop. In a big-step semantics, the while loop has one or three premises. First, the condition is evaluated. Then, in the case it returned `true`, the statement and the rest of the loop is evaluated. The evaluation of terms returns either a pair of a state and a value (when evaluating an expression) or simply a state (when evaluating a statement). Writing S for states, we thus have the following.

$$\frac{S, \underline{e} \to S', \texttt{false}}{S, \underline{\texttt{while}\ e\ \texttt{do}\ s} \to S'} \qquad \frac{S, \underline{e} \to S', \texttt{true} \qquad S', s \to S'' \qquad S'', \underline{\texttt{while}\ e\ \texttt{do}\ s} \to S'''}{S, \underline{\texttt{while}\ e\ \texttt{do}\ s} \to S'''}$$

In the pretty-big-step approach, only one sub-term is evaluated in each rule. The result of the evaluation is gathered, along with the remaining sub-terms, in a new syntactic construct called an *extended term*. For instance, the first reduction for the `while` loop is as follows.

$$\frac{S, \underline{e} \to S', v \qquad S, \underline{\texttt{while1}(S', v, e, s)} \to S''}{S, \underline{\texttt{while}\ e\ \texttt{do}\ s} \to S''}$$

The $\underline{\texttt{while1}(S', v, e, s)}$ term includes the result of evaluating $\underline{e}$ (namely S', v), as well as the information required to evaluate the rest of the loop. The $\underline{\texttt{while1}(\cdot,\cdot,\cdot,\cdot)}$ term is evaluated using one of the two rules below. If v is `false`, then the evaluation immediately returns. Otherwise a second extended term is used.

$$
\begin{array}{ll}
s ::= & e ::= \\
\mid \texttt{skip} & \mid c \\
\mid s_1; s_2 & \mid \texttt{x} \\
\mid \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 & \mid e_1 \texttt{ op } e_2 \\
\mid \texttt{while } e \texttt{ do } s & \mid \texttt{\{\}} \\
\mid \texttt{x = } e & \mid e\texttt{.f} \\
\mid e_1\texttt{.f = } e_2 & \\
\mid \texttt{delete } e\texttt{.f} &
\end{array}
$$

Figure 1: O'WHILE Syntax

Note that the state S in the conclusion of the rules is not used, as the starting state is present in the extended term. We still include it to ensure all the rules have the same shape.

$$
\frac{}{S, \texttt{while1}(S', \texttt{false}, e, s) \to S'}
\qquad
\frac{S', s \to S'' \qquad S', \texttt{while2}(S'', e, s) \to S'''}{S, \texttt{while1}(S', \texttt{true}, e, s) \to S'''}
$$

In this second case, the result of evaluating the statement s, namely a new state S'', is stored in the extended term $\texttt{while2}(S'', e, s)$. Finally, this new state is used to evaluate the next iteration of the loop.

$$
\frac{S'', \texttt{while } e \texttt{ do } s \to S'''}{S', \texttt{while2}(S'', e, s) \to S'''}
$$

Putting it all together, here is a full derivation of one run of a loop.

$$
\frac{\dfrac{\dots}{S, e \to S', \texttt{true}} \qquad \dfrac{\dfrac{\dots}{S', s \to S''} \qquad \dfrac{\dfrac{\dots}{S'', \texttt{while } e \texttt{ do } s \to S'''}}{S', \texttt{while2}(S'', e, s) \to S'''}}{S, \texttt{while1}(S', \texttt{true}, e, s) \to S'''}}{S, \texttt{while } e \texttt{ do } s \to S'''}
$$

3 O'WHILE and its Pretty Big Step Semantics

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.

$v ::=$	$r ::=$	$s_e ::=$	$e_e ::=$
$\mid c$	$\mid S$	$\mid s$	$\mid e$
$\mid l$	$\mid S, v$	$\mid r ;_1 s$	$\mid r\ \mathtt{op}_1\ e$
	$\mid S, \mathtt{err}$	$\mid \mathtt{if1}(r, s_1, s_2)$	$\mid v\ \mathtt{op}_2\ r$
		$\mid \mathtt{while1}(r, e, s)$	$\mid r\ ._1 \mathtt{f}$
		$\mid \mathtt{while2}(r, e, s)$	
		$\mid \mathtt{x} =_1 r$	
		$\mid r.\mathtt{f} =_1 e$	
		$\mid l.\mathtt{f} =_2 r$	
		$\mid \mathtt{delete1}\ r.\mathtt{f}$	

Figure 2: O'WHILE Values and Extended Syntax

Objects are passed by reference, thus values v (Figure 2) 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 $S, \mathtt{err}$.

Figure 2 also introduces *extended statements* and *extended expressions* that are used in O'WHILE's pretty-big-step semantics, presented in Figure 3. 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 $S', \mathtt{err}$. 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 $\mathtt{st}(r)$ 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 $e.\mathtt{f}$ first evaluates e to some result r, then calls the rule for the extended expression $r\ ._1\mathtt{f}$. 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 $\mathtt{f}$. 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 $\mathtt{abort}(\cdot)$ from extended terms to results, that indicates when an error is to be raised or propagated. More precisely, the function $\mathtt{abort}(t_e)$ is defined at least if t_e is an extended term containing a subterm equal to $S, \mathtt{err}$ for some S. In this case $\mathtt{abort}(t_e) = S, \mathtt{err}$. We can then extend this function to define erroneous cases. For instance, we could say that $\mathtt{abort}((E, H, v).\mathtt{f} =_1 e) = E, H, \mathtt{err}$ if v is not a location, or if $v = l$ but l is not in the domain of H, or if $\mathtt{f}$ 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 4 is an example of a derivation of the semantics. It will be the basis for the running example of this paper.

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

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

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

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

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

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

$$\frac{S', e \to r \qquad S', l.\texttt{f} =_2 r \to r'}{S, (S', l).\texttt{f} =_1 e \to r'}\ \textsc{FldAsg1}$$

$$\frac{H[l] = o \qquad o' = o[\texttt{f} \mapsto v] \qquad H' = H\left[l \mapsto o'\right]}{S, l.\texttt{f} =_2 (E, H, v) \to E, H'}\ \textsc{FldAsg2} \qquad \frac{S, e \to r \qquad S, \texttt{delete1}\ r.\texttt{f} \to r'}{S, \texttt{delete}\ e.\texttt{f} \to r'}\ \textsc{Del}$$

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

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

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

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

Figure 3: O'WHILE's Semantics

$$\textsc{Obj}\ \frac{H'''[l''] = \bot \qquad H_f = H'''[l'' \mapsto \texttt{\{\}}]}{E', H''', \texttt{\{\}} \to E', H_f, l''} \qquad \frac{E_f = E'[\texttt{x} \mapsto l'']}{E', H''', \texttt{x} =_1 (E', H_f, l'') \to E_f, H_f}\ \textsc{Asg1}$$

$$\frac{\cdots}{E', H''', \texttt{y = \{\}} \to E_f, H_f}\ \textsc{Asg}$$

$$\frac{\cdots}{E', H''', \texttt{if1}(E', H''', \texttt{false}, \texttt{y = x.f}, \texttt{y = \{\}}) \to E_f, H_f}\ \textsc{IfFalse}$$

$$\vdots$$

$$\textsc{Cst}\ \frac{}{E', H''', \texttt{false} \to E', H''', \texttt{false}}$$

$$\frac{\cdots}{E', H''', \texttt{if false then y = x.f else y = \{\}} \to E_f, H_f}\ \textsc{If}$$

$$\frac{\cdots}{E', H', (E', H''') ;_1 \texttt{if false then y = x.f else y = \{\}} \to E_f, H_f}\ \textsc{Seq1}$$

$$\vdots$$

$$\textsc{Obj}\ \frac{H'[l'] = \bot \qquad H'' = H'[l' \mapsto \texttt{\{\}}]}{E', H', \texttt{\{\}} \to E', H'', l'} \qquad \frac{H'[l] = \texttt{\{\}} \qquad o' = \texttt{\{f} \mapsto l'\texttt{\}} \qquad H''' = H''\left[l \mapsto o'\right]}{E', H', l.\texttt{f} =_2 (E', H'', l') \to E', H'''}\ \textsc{FldAsg2}$$

$$\frac{\cdots}{E', H', (E', H', l).\texttt{f} =_1 \texttt{\{\}} \to E', H'''}\ \textsc{FldAsg1}$$

$$\vdots$$

$$\textsc{Var}\ \frac{E'[\texttt{x}] = l}{E', H', \texttt{x} \to E', H', l}$$

$$\textsc{FldAsg}\ \frac{\cdots}{E', H', \texttt{x.f = \{\}} \to E', H'''}$$

$$\frac{\cdots}{E', H', \texttt{x.f = \{\};if false then y = x.f else y = \{\}} \to E_f, H_f}\ \textsc{Seq}$$

$$\frac{\cdots}{E, H, (E', H') ;_1 \texttt{x.f = \{\};if false then y = x.f else y = \{\}} \to E_f, H_f}\ \textsc{Seq1}$$

$$\vdots$$

$$\textsc{Obj}\ \frac{H[l] = \bot \qquad H' = H[l \mapsto \texttt{\{\}}]}{E, H, \texttt{\{\}} \to E, H', l} \qquad \frac{E' = E[\texttt{x} \mapsto l]}{E, H, \texttt{x} =_1 E, H', l \to E', H'}\ \textsc{Asg1}$$

$$\textsc{Asg}\ \frac{\cdots}{E, H, \texttt{x = \{\}} \to E', H'}$$

$$\frac{\cdots}{E, H, \texttt{x = \{\};x.f = \{\};if false then y = x.f else y = \{\}} \to E_f, H_f}\ \textsc{Seq}$$

Figure 4: Pretty-big-step derivation

4 Annotated Semantics

4.1 Execution traces

We want to track how data created at one point in the execution flows into locations (variables or object fields) at another, later point 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. Notice that this instrumentation adds no information to the instrumented trace. We will use these traces later in this section to define direct flows.

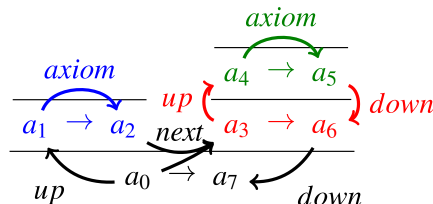
More precisely, we add partial traces, $\tau \in \mathit{Trace}$, to both sides of the reduction rules. These traces are lists of rule names decorated with "*i*" or "*o*", *i* meaning that we entered in the context of the rule and *o* that we finished executing it. Given a rule whose name is R and an initial trace τ, we put $\tau; \mathrm{R}_i$ on the left of the rule. If the rule is an axiom (such as CST), we then put $\tau; \mathrm{R}_i; \mathrm{R}_o$ on the right-hand side of the rule. Otherwise we recursively call this annotation algorithm on the first premise, then thread its result on the second premise if there is one. The final trace on the right is the one returned by the last premise, with R_o appended at the end. As each rule appends its name in the annotation on both sides, they can be uniquely identified. An example of a derivation with explicitly written traces is given in Figure 5.

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 place.

4.2 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 five kinds: *init* steps (for every rule), that create the annotation on the left of the rules based on the annotation passed in argument, *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.



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

(a) Unannotated Derivation

$$\tau_1 = [\textsc{Seq}_i] \qquad \tau_2 = [\textsc{Seq}_i ; \textsc{Asg}_i] \qquad \tau_3 = \tau_2 +\!\!+ [\textsc{Cst}_i] \qquad \tau_4 = \tau_3 +\!\!+ [\textsc{Cst}_o] \qquad \tau_5 = \tau_4 +\!\!+ [\textsc{Asg1}_i]$$

$$\tau_6 = \tau_5 +\!\!+ [\textsc{Asg1}_o] \qquad \tau_7 = \tau_6 +\!\!+ [\textsc{Asg}_o] \qquad \tau_8 = \tau_7 +\!\!+ [\textsc{Seq1}_i] \qquad \tau_9 = \tau_8 +\!\!+ [\textsc{Asg}_i]$$

$$\tau_{10} = \tau_9 +\!\!+ [\textsc{Var}_i] \qquad \tau_{11} = \tau_{10} +\!\!+ [\textsc{Var}_o] \qquad \tau_{12} = \tau_{11} +\!\!+ [\textsc{Asg1}_i] \qquad \tau_{13} = \tau_{12} +\!\!+ [\textsc{Asg1}_o]$$

$$\tau_{14} = \tau_{13} +\!\!+ [\textsc{Asg}_o] \qquad \tau_{15} = \tau_{14} +\!\!+ [\textsc{Seq1}_o] \qquad \tau_{16} = \tau_{15} +\!\!+ [\textsc{Seq}_o]$$

$$
\dfrac{
\dfrac{
\dfrac{}{\tau_3, E, H, \texttt{true} \rightarrow \tau_4, E, H, \texttt{true}}\,\textsc{Cst}
\qquad
\dfrac{E' = E[\texttt{x} \mapsto \texttt{true}]}{\tau_5, E, H, \texttt{x} =_1 (E, H, \texttt{true}) \rightarrow \tau_6, E', H}\,\textsc{Asg1}
}{\tau_2, E, H, \texttt{x = true} \rightarrow \tau_7, E', H}\,\textsc{Asg}
\qquad
\dfrac{
\dfrac{
\dfrac{E'[\texttt{x}] = \texttt{true}}{\tau_{10}, E', H, \texttt{x} \rightarrow \tau_{11}, E', H, \texttt{true}}\,\textsc{Var}
\qquad
\dfrac{E'' = E'[\texttt{y} \mapsto \texttt{true}]}{\tau_{12}, E', H, \texttt{x} =_1 E', H, \texttt{true} \rightarrow \tau_{13}, E'', H}\,\textsc{Asg1}
}{\tau_9, E', H, \texttt{y = x} \rightarrow \tau_{14}, E'', H}\,\textsc{Asg}
}{\tau_8, E, H, (E', H) ;_1 \texttt{y = x} \rightarrow \tau_{15}, E'', H}\,\textsc{Seq1}
}{\tau_1, E, H, \texttt{x = true; y = x} \rightarrow \tau_{16}, E'', H}\,\textsc{Seq}
$$

(b) Derivation Annotated With Traces

Figure 5: Annotating A Simple Derivation

This generic approach allows to compose complex annotations, building upon previously defined ones. This general scheme is summed up in the previous picture, where each a_i represents an annotation. As *init* steps are defined for every rule, they are not depicted. The colors show which steps are associated to which rules: a_0 is created by the *init* step of the bottom rule (black), then it is transformed and control is passed to the left axiom rule (black *up*), which applies its *init* step to create the blue a_1. The blue *axiom* step creates a_2, and control returns to the bottom rule, where the black *next* step combines a_1 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 *init*, *axiom*, *up*, *down*, and *next* steps corresponding to the addition of partial traces (see Figure 6b and 7b).

- The *init* step is defined for every rule and can be seen in both figures. Assuming the rule name is NAME, then this step appends NAME_i to the previous trace τ, written $\tau +\!\!+ [\text{NAME}_i]$.
- The *axiom* step only needs to be defined for rules with no inductive premise, namely SKIP, WHILE-FALSE, ASG1, FLDASG2, DEL1, CST, VAR, BIN2, OBJ, FLD1, and ABORT. It adds the current rule name to its argument τ, decorated with "*o*", as illustrated in Figure 6b: $\tau +\!\!+ [\text{NAME}_o]$.
- The *up* step is only defined on rules that have an inductive premise, and it simply propagates the annotation to the first premise. It is illustrated in Figure 7b.
- The *down* step takes the right trace from the last premise of the rule above and adds the current rule name to the right annotation decorated with "*o*" (Figure 7b).
- The *next* step takes two arguments: the right trace of the first premise τ_1 and the left trace τ_0 of the current rule. It ignores τ_0 and propagates τ_1 (Figure 7b).

4.3 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

$$\leftsquigarrow \in Dep = \mathscr{P}(Source \times Store)$$

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

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

(a) Basic Rule

$$\frac{E[\underline{\mathtt{x}}] = v \qquad \tau' = \tau + [\text{VAR}_i; \text{VAR}_o]}{E, H, \tau + [\text{VAR}_i], \underline{\mathtt{x}} \to E, H, \tau', v}\ \text{VAR}$$

(b) Adding Partial Traces

$$\text{VAR}\ \frac{E[\underline{\mathtt{x}}] = v \qquad \tau' = \tau + [\text{VAR}_i; \text{VAR}_o]}{E, H, \tau + \text{VAR}_i, M, \underline{\mathtt{x}} \to E, H, \tau', M, v}$$

(c) Adding Last-Modified Place

$$\frac{E[\underline{\mathtt{x}}] = v \qquad \tau' = \tau + [\text{VAR}_i; \text{VAR}_o] \qquad M[\mathtt{x}] = \tau_0}{E, H, \tau + \text{VAR}_i, M, \underline{\mathtt{x}} \to E, H, \tau', M, \{\mathtt{x}^{\tau_0}\}, v}\ \text{VAR}$$

(d) Adding Dependencies

Figure 6: Instrumentation Steps for VAR

For instance, we write $\mathtt{y}^{\tau_1} \rightsquigarrow \mathtt{x}^{\tau_2}$ to indicate that the content that was put in the variable y at time τ_1 has been used to compute the value stored in the variable x at time τ_2. Similarly, we write $l^{\tau_2} \rightsquigarrow l'^{\tau_1}.\mathtt{f}^{\tau_3}$ to indicate that the object allocated at location l at time τ_2 flows at time τ_3 into field f of location l' that was allocated at time τ_1.

4.4 Direct Flows

We now detail how to compose additional annotations to define our direct flow property $\rightsquigarrow$. 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 $(Loc + Var + ALoc \times Field) \to Trace$. After this instrumentation step, reductions are of the form $\tau, M_\tau, S_\tau, t \to \tau', M_{\tau'}, r$.

The three rules that modify M are OBJ, ASG1, and FLDASG2. We describe them in Figure 8. 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* $d \in \mathscr{P}(Var)$. 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 9. 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 $\{\Delta_\tau, \Delta_{\tau'}\} \subseteq Dep$ are sets of flows defining

$$\textsc{Rule}\ \frac{\dfrac{\vdots}{S, e \to r} \qquad \dfrac{\vdots}{S, \underline{\mathtt{x} =_1 r} \to r'}\ \textsc{Asg1}}{S, \underline{\mathtt{x = }e} \to r'}\ \textsc{Asg}$$

(a) Basic Rule

$$\tau_0 = \tau +\!\!+ [\textsc{Asg}_i] \qquad \tau_1 = \tau_0 +\!\!+ [\textsc{Rule}_i] \qquad \tau_3 = \tau_2 +\!\!+ [\textsc{Asg1}_i] \qquad \tau_5 = \tau_4 +\!\!+ [\textsc{Asg}_o]$$

$$\textsc{Rule}\ \frac{\dfrac{\vdots}{\tau_1, S, e \to \tau_2, r} \qquad \dfrac{\vdots}{\tau_3, S, \underline{\mathtt{x} =_1 r} \to \tau_4, r'}\ \textsc{Asg1}}{\tau_0, \underline{\mathtt{x = }e}, \to \tau_5, r'}\ \textsc{Asg}$$

(b) Adding Partial Traces

$$\textsc{Rule}\ \frac{\dfrac{\vdots}{\tau_1, \boldsymbol{M}, S, e \to \tau_2, \boldsymbol{M'}, r} \qquad \dfrac{\vdots}{\tau_3, \boldsymbol{M'}, S, \underline{\mathtt{x} =_1 r} \to \tau_4, \boldsymbol{M''}, r'}\ \textsc{Asg1}}{\tau_0, \boldsymbol{M}, \underline{\mathtt{x = }e}, \to \tau_5, \boldsymbol{M''}, r'}\ \textsc{Asg}$$

(c) Adding Last-Modified Place

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

(d) Adding Dependencies

Figure 7: Instrumentation Steps for ASG

the $\looparrowleft$ relation (see Section 4.3). 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 rules just propagate those new constructions. The two modified rules are given in Figure 10.

4.5 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 coherence properties between the instrumented semantics.

We first describe the relation between program points and traces in the derivations, and how to add program points to programs. 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 is a *context*, a list of atoms indicating where to find the term in the initial program. The program point after is a context followed by the name of 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

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

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

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

Figure 8: Adding Modified and Created Information

`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 context and assume that $\cdot/\textsc{Name}$ is equal to $\textsc{Name}$.

$$
\begin{aligned}
\Pi(\cdot, \mathtt{skip}) &= \mathrm{PP}, \mathtt{skip}, \mathrm{PP}/\textsc{Skip}\\
\Pi(\cdot, s_1; s_2) &= \mathrm{PP}, \Pi(\mathrm{PP}/\textsc{Seq1}, s_1); \Pi(\mathrm{PP}/\textsc{Seq2}, s_2), \mathrm{PP}/\textsc{Seq}\\
\Pi(\cdot, \mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2) &= \mathrm{PP}, \mathtt{if}\ \Pi(\mathrm{PP}/\textsc{IfE}, e)\ \mathtt{then}\ \Pi(\mathrm{PP}/\textsc{If1}, s_1)\ \mathtt{else}\ \Pi(\mathrm{PP}/\textsc{If2}, s_2), \mathrm{PP}/\textsc{If}\\
\Pi(\cdot, \mathtt{while}\ e\ \mathtt{do}\ s) &= \mathrm{PP}, \mathtt{while}\ \Pi(\mathrm{PP}/\textsc{WhileE}, e)\ \mathtt{do}\ \Pi(\mathrm{PP}/\textsc{WhileS}, s), \mathrm{PP}/\textsc{While}\\
\Pi(\cdot, \mathtt{x} = e) &= \mathrm{PP}, \mathtt{x} = \Pi(\mathrm{PP}/\textsc{AsgE}, e), \mathrm{PP}/\textsc{Asg}\\
\Pi(\cdot, e_1.\mathtt{f} = e_2) &= \mathrm{PP}, \Pi(\mathrm{PP}/\textsc{FldAsg1}, e_1).\mathtt{f} = \Pi(\mathrm{PP}/\textsc{FldAsg2}, e_2), \mathrm{PP}/\textsc{FldAsg}\\
\Pi(\cdot, \mathtt{delete}\ e.\mathtt{f}) &= \mathrm{PP}, \mathtt{delete}\ \Pi(\mathrm{PP}/\textsc{DelE}, e).\mathtt{f}, \mathrm{PP}/\textsc{Del}\\
\Pi(\cdot, c) &= \mathrm{PP}, c, \mathrm{PP}/\textsc{Cst}\\
\Pi(\cdot, \mathtt{x}) &= \mathrm{PP}, \mathtt{x}, \mathrm{PP}/\textsc{Var}\\
\Pi(\cdot, e_1\ \mathtt{op}\ e_2) &= \mathrm{PP}, \Pi(\mathrm{PP}/\textsc{Bin1}, e_1)\ \mathtt{op}\ \Pi(\mathrm{PP}/\textsc{Bin2}, e_2), \mathrm{PP}/\textsc{Bin}\\
\Pi(\cdot, \{\}) &= \mathrm{PP}, \{\}, \mathrm{PP}/\textsc{Obj}\\
\Pi(\cdot, e.\mathtt{f}) &= \mathrm{PP}, \Pi(\mathrm{PP}/\textsc{FldE}, e).\mathtt{f}, \mathrm{PP}/\textsc{Fld}
\end{aligned}
$$

We next define a function $\mathscr{T}$ from traces, list of unmatched trace points, and partial program points to program points and unmatched traces. In the following we write _ for any trace atom, as long as it has not been matched by a previous rule. This function relies on some additional helper functions whose rules can be found in Figure 15 at the end of the paper.

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

$$\frac{H'[l] = o \qquad o[\mathtt{f}] = v \qquad M[(l, M[l], f)] = \tau_0}{\tau, M, d, E, H, \underline{(E', H', l)\,._1\mathtt{f}} \to \tau', M, d \cup \{(l, M[l], f)^{\tau_0}\}, E', H', v}\ \text{FLD1}$$

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

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

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

$$\frac{E' = E[\mathtt{x} \mapsto v] \qquad M' = M[\mathtt{x} \mapsto \tau']}{\tau, M, d, S, \underline{\mathtt{x} =_1 (E, H, v)} \to \tau', M', \emptyset, E', H}\ \text{ASG1}$$

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

$$\frac{\tau_1, M, \emptyset, S', e \to \tau_2, M', d, r \qquad \tau_3, M', d, S', \underline{l.\mathtt{f} =_2 r} \to \tau_4, M'', \emptyset, r'}{\tau_0, M, \emptyset, S, \underline{(S', x).\mathtt{f} =_1 e} \to \tau_5, M'', \emptyset, r'}\ \text{FLDASG1}$$

$$\frac{H[l] = o \qquad o' = o[\mathtt{f} \mapsto v] \qquad H' = H'\left[l \mapsto o'\right] \qquad M' = M[(l, M[l], f) \mapsto \tau']}{\tau, M, d, S, \underline{l.\mathtt{f} =_2 (E, H, v)} \to \tau', M', \emptyset, E, H'}\ \text{FLDASG2}$$

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

Figure 9: Rules for Dependencies Annotations

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

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

Figure 10: Rules for Annotating Dependencies of Statements

$$\begin{aligned}
\mathscr{T}(\tau ++ [\text{Name}_o], [], \text{PP}) &= \mathscr{T}(\tau, \mathtt{Push}_o(\text{Name}) :: [], \mathtt{App}_o(\text{Name}, \text{PP})) \\
\mathscr{T}(\tau ++ [\text{Name}_o], \text{Name}_o :: l, \text{PP}) &= \mathscr{T}(\tau, \text{Name}_o :: \text{Name}_o :: l, \text{PP}) \\
\mathscr{T}(\tau ++ [\text{Name}_i], \text{Name}_o :: l, \text{PP}) &= \mathscr{T}(\tau, l, \text{PP}) \\
\mathscr{T}(\tau ++ [_], \text{Name}_o :: l, \text{PP}) &= \mathscr{T}(\tau, \text{Name}_o :: l, \text{PP}) \\
\mathscr{T}(\tau ++ [\text{Name}_i], [], \text{PP}) &= \mathscr{T}(\tau, \mathtt{Push}_i(\text{Name}), \mathtt{App}_i(\text{Name}, \text{PP}))
\end{aligned}$$

We call "normal names" the name of rules for the non-extended terms, *i.e.* SKIP, SEQ, IF, WHILE, ASG, FLDASG, DEL, CST, VAR, BIN, OBJ, and FLD. We call "extended names" the names of the other rules.

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. Note that only normal terms have exposed program points, of the form PP, t, PP'. Extended terms also contain program points but have none at toplevel.

Property 1 *Let t be a term and $t' = \Pi(\cdot, t)$. For any occurrence of a normal rule* NAME *of the form $\tau, S, PP, t, PP' \to \tau', r$ in any annotated derivation tree from t', we have $PP = \mathscr{T}(\tau, [], \cdot)$, $PP' = \mathscr{T}(\tau', [], \cdot)$, and $PP' = PP/\text{Name}$.*

We next 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 2 *For every instrumented derivation 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 $M_{\tau'}[l^{\tau_0}.\mathtt{f}] = \tau_1$, Then 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 3 *For every instrumented derivation 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 $\mathtt{x}^{\tau_1} \leadsto \mathtt{y}^{\tau_2} \in \Delta_{\tau'}$, 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 annotation.

More precisely, we write $l_0 \looparrowleft^*_\Delta l_n.\texttt{f}$ if there are stores $s_0 \ldots s_n$ such that:

- $s_0 = l_0^{\tau_0}$ for some τ_0, $s_n = l_n^{\tau_n}.\texttt{f}^{\tau'_n}$ for some τ_n and τ'_n, and for every i in $[1..n]$ we have either $s_i = l_i^{\tau_i}.\texttt{f}_i{}^{\tau'_i}$ for some l_i, $\texttt{f}_i$, τ_i, and τ'_i or $s_i = \texttt{x}_i{}^{\tau'_i}$ for some $\texttt{x}_i$ and τ'_i; and
- for every i, $s_i \looparrowleft s_{i+1} \in \Delta$.

Property 4 *For every instrumented derivation 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 $\texttt{st}(r) = E_{\tau'}, H_{\tau'}$, *we have:*

- *for every locations* l, l' *and field* $\texttt{f}$ *such that* $H_\tau[l'][\texttt{f}] = l$, *then* $l \looparrowleft^*_{\Delta_\tau} l'.\texttt{f}$,
- *for every locations* l, l' *and field* $\texttt{f}$ *such that* $H_{\tau'}[l'][\texttt{f}] = l$, *then* $l \looparrowleft^*_{\Delta_{\tau'}} l'.\texttt{f}$.

4.6 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` $\to$ `heap` $\to$ `ext_expr` $\to$ `out_expr` $\to$ `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 4.5 to ensure they define the property of interest.

The semantics is thus parametrised 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 11 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 annotations in our Coq development exactly follows the scheme presented in Section 4.2. 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 wildcard _ to deal with all the cases that just propagate the annotations. It has been written

```
1  Inductive red_expr : environment → heap_o → ext_expr → out_expr → Type :=
2
3    (* ... *)
4
5    | red_expr_expr_var : annot_e_l Annots → annot_e_r Annots →
6      ∀E H x v,
7      getvalue E x v →
8      red_expr E H (ext_expr_expr (expr_var x)) (out_expr E H v)
```

Figure 11: A Semantic Rule as Written in COQ

in a modular way, which is robust to changes. For example, a previous version of the annotations only used open rules for the partial traces and not closing one. 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 for this change.

Figure 12 shows the introduction of the last-modified annotation (see Figure 6c and 7c). This annotation is parameterized by another (traces for instance) here called `Locations`. In COQ, the heap M of Section 4.4 is represented by the record `LastChangeHeaps` defined on Line 5. Line 12 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 20 which, as in Figure 7c, stores the current location τ in the annotation. Line 34 sums up the rules, stating that every rule 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.

5 Dependency Analysis

The annotating process makes the property we want to track appear explicitly in derivation trees. However, the set of properties in question is still infinite so it might not be possible to compute the instrumented semantics of a program. The 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.

5.1 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

```
1  Section LastModified.
2  Variable Locations : Annotations.
3
4  Definition ModifiedAnnots := annot_s_r Locations.
5  Record LastModifiedHeaps : Type :=
6    makeLastModifiedHeaps {
7      LCEnvironment : heap var ModifiedAnnots;
8      LCHeap : heap loc (heap prop_name ModifiedAnnots)
9    }.
10
11 Definition LastModified :=
12   ConstantAnnotations LastModifiedHeaps.
13
14 Definition LastModifiedAxiom_s (r : LastModifiedHeaps)
15   E H t o (R : red_stat Locations E H t o) :=
16   let LCE := LCEnvironment r in
17   let LCH := LCHeap r in
18   let (_, tau) := extract_annot_s R in
19   match R with
20   | red_stat_ext_stat_assign_1 _ _ _ _ _ _ _ x _ _ ⇒
21     let LCE' := write LCE x tau
22     in makeLastModifiedHeaps LCE' LCH
23   | red_stat_stat_delete _ _ _ _ _ l _ f _ _ _ _ _ _ ⇒
24     let aob := read LCH l
25     in let LCH' := write LCH l (write aob f tau)
26     in makeLastModifiedHeaps LCE LCH'
27   | red_stat_ext_stat_set_2 _ _ _ _ _ _ _ _ l _ f _ _ _ _ _ _ ⇒
28     let aob := read LCH l
29     in let LCH' := write LCH l (write aob f tau)
30     in makeLastModifiedHeaps LCE LCH'
31   | _ ⇒makeLastModifiedHeaps LCE LCH
32   end.
33
34 Definition annotLastModified :=
35   makeIterativeAnnotations LastModified
36     (init_e Transmit) (axiom_e Transmit) (up_e Transmit) (down_e Transmit) (next_e Transmit)
37     (up_s_e Transmit) (next_e_s Transmit)
38     (init_s Transmit) LastModifiedAxiom_s (up_s Transmit) (down_s Transmit) (next_s Transmit).
39
40 End LastModified.
```

Figure 12: COQ Definitions of the Last-Modified Annotation

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 Section 4.5. Property 1 assures 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 were defined to be either basic values or locations. We shall ignore the basic values and focus on abstracting the unbounded set of heap locations. There are several ways of abstracting objects in the heap, but we shall content ourselves with the standard abstraction in which object locations are abstracted by the program points corresponding to the instruction that allocated the object. Thus

$$l^\sharp \in Loc^\sharp = \mathscr{P}(PP).$$

The abstraction of values should in addition contain a component for tracking variables $\mathtt{x}$ on which the value depends. Values are thus abstracted by a pair $v^\sharp$ of abstract location l and of the set $d^\sharp$ of variables that possibly flowed into this specific value, annotated with the program points of their definition.

$$v^\sharp \in Val^\sharp = Loc^\sharp \times \mathscr{P}(Var \times PP)$$

For objects stored at heap locations we keep trace of the values that the fields may reference. Environments and heaps are then abstracted as follows:

- $E^\sharp \in Env^\sharp = Var \to Val^\sharp$ maps variables to abstract values $v^\sharp$.
- $H^\sharp \in Heap^\sharp = Loc^\sharp \to Field \to Val^\sharp$ maps abstract locations to object abstractions (that map fields to abstract values).

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 $Loc^\sharp$ (which are sets of program points) to abstract object. As locations are abtracted by sets, each write of a value $v^\sharp$ in the abstract heap at abstract location $l^\sharp$ implicitely yields a join between $v^\sharp$ and every value associated to an $l'^\sharp \sqsubseteq l^\sharp$.

We recall the definition of *Dep*, *Store* and *Source*:

$$Dep = \mathscr{P}(Source \times Store) \qquad ALoc = Loc \times Trace$$

$$Store = (Var \times Trace) + (ALoc \times Field \times Trace) \qquad Source = ALoc + Store$$

We want to 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 $\Delta_\tau \in Dep$ (and the relation $\leftarrowtail$) by making the natural abstraction $\Delta^\sharp$ of those definitions:

$$\Delta^\sharp \in Dep^\sharp = \mathscr{P}\left(Source^\sharp \times Store^\sharp\right) \qquad ALoc^\sharp = PP$$

$$Store^\sharp = (Var \times PP) + (PP \times Field \times PP) \qquad Source^\sharp = PP + Store^\sharp$$

Abstract flows are written using the symbol $\leftarrowtail^\sharp$. To avoid confusion, program points $p \in PP$ interpreted as elements of $Source^\sharp$ (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 4.1 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 $Dep^\sharp$: for instance for each $\mathtt{x}^\tau \in Var \times Trace \subset Store$ such that $\tau \prec p$, we have $\mathtt{x}^\tau \prec \mathtt{x}^p \in Store^\sharp$. Similarly, this relation $\prec$ can also be defined over *Val* and $Val^\sharp$, *Env* and $Env^\sharp$, and *Heap* and $Heap^\sharp$.

5.2 Abstract Reduction Relation

We formalize the analysis as an abstract reduction relation $\rightarrow^\sharp$ for expressions and statements:

$$E^\sharp, H^\sharp, \underline{s} \rightarrow^\sharp E'^\sharp, H'^\sharp, \Delta^\sharp \qquad\qquad E^\sharp, H^\sharp, \underline{e} \rightarrow^\sharp l^\sharp, d^\sharp$$

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 $\underline{s}$ 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 13 shows the rules of this analysis. To avoid burdening notations, we denote by $d^\sharp \looparrowleft f$ 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 $\left\{p_l.\mathtt{f}^p \middle| p_l \in l^\sharp\right)$. As an example, here is the rule for assignments:

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

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

The Bin rule makes use of an abstract operation $op^\sharp$, which depends on the operators added in the language. Most of the time, it shall be either $\sqcup$, or the operation ignoring its operands and returning $\emptyset$. For instance, if the rules of $+$ forces its two operands to be integers, raising an uncatchable error if one of them is an object, it's safe to suppose $+^\sharp$ to be equal to $\lambda d_1^\sharp d_2^\sharp.\emptyset$ as its returned value cannot be an object. Figure 14 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 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.

5.3 Analysis in Coq

The abstract domains are essentially the same as the ones described in Section 5.1. They are straightforward to formalise as soon as basic constructions for lattices are available: the abstract domains are just specific instances of standard lattices from abstract interpretation (flat lattices, power set lattices...). For the certification of lattices we refer to the Coq developments by David Pichardie [10].

Similarly to Section 4.6, the rules of the analyser presented in Figure 13 are first defined as an inductive predicate of type

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

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

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

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

$$\frac{E^\sharp, H^\sharp, \underline{e} \rightarrow^\sharp l^\sharp, d^\sharp}{E^\sharp, H^\sharp, \underline{\texttt{x}^p\ \texttt{=}\ e} \rightarrow^\sharp E^\sharp[\texttt{x} \mapsto (l^\sharp, d^\sharp)], H^\sharp, \left(l^\sharp \cup d^\sharp\right) \rightsquigarrow^\sharp \texttt{x}^p}\ \textsc{Asg}$$

$$\frac{E^\sharp, H^\sharp, \underline{e_1} \rightarrow^\sharp l_1^\sharp, d_1^\sharp \qquad E^\sharp, H^\sharp, \underline{e_2} \rightarrow^\sharp l_2^\sharp, d_2^\sharp}{E^\sharp, H^\sharp, \underline{e_1\texttt{.f}^p\ \texttt{=}\ e_2} \rightarrow^\sharp E^\sharp, H^\sharp \sqcup \left\{\left(l_1^\sharp, \texttt{f}\right) \mapsto \left(l_2^\sharp, d_2^\sharp\right)\right\}, \left(l_2^\sharp \cup d_2^\sharp\right) \rightsquigarrow^\sharp l_1^\sharp\texttt{.f}^p}\ \textsc{FldAsg}$$

$$\frac{E^\sharp, H^\sharp, \underline{e} \rightarrow^\sharp l^\sharp, d^\sharp}{E^\sharp, H^\sharp, \underline{\texttt{delete}\ e\texttt{.f}} \rightarrow^\sharp E^\sharp, H^\sharp, d^\sharp \rightsquigarrow^\sharp l^\sharp\texttt{.f}}\ \textsc{Del} \qquad \frac{}{E^\sharp, H^\sharp, \underline{c} \rightarrow^\sharp \emptyset, \emptyset}\ \textsc{Cst}$$

$$\frac{E^\sharp[\texttt{x}] \sqsubseteq \left(l^\sharp, d^\sharp\right)}{E^\sharp, H^\sharp, \underline{\texttt{x}^p} \rightarrow^\sharp l^\sharp, \{\texttt{x}^p\} \cup d^\sharp}\ \textsc{Var} \qquad \frac{E^\sharp, H^\sharp, e_1 \rightarrow^\sharp l_1^\sharp, d_1^\sharp \qquad E^\sharp, H^\sharp, e_2 \rightarrow^\sharp l_2^\sharp, d_2^\sharp}{E^\sharp, H^\sharp, \underline{e_1\ \texttt{op}\ e_2} \rightarrow^\sharp l_1^\sharp\, op^\sharp\, l_2^\sharp, d_1^\sharp \cup d_2^\sharp}\ \textsc{Bin}$$

$$\frac{}{E^\sharp, H^\sharp, \underline{\texttt{\{\}}^p} \rightarrow^\sharp \{o^p\}, \emptyset}\ \textsc{Obj} \qquad \frac{E^\sharp, H^\sharp, \underline{e} \rightarrow^\sharp l^\sharp, d^\sharp \qquad H^\sharp[l^\sharp] \sqsubseteq o^\sharp \qquad o^\sharp[\texttt{f}] \sqsubseteq \left(l_f^\sharp, d_f^\sharp\right)}{E^\sharp, H^\sharp, \underline{e\texttt{.f}^p} \rightarrow^\sharp l_f^\sharp, \left(l^\sharp\texttt{.f}^p\right) \cup d^\sharp \cup d_f^\sharp}\ \textsc{Fld}$$

Figure 13: Rules for the Abstract Reduction Relation

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

$$E_3^\sharp = \{\mathtt{x} \mapsto (\{o^{p_1}\}, \emptyset), \mathtt{y} \mapsto (\{o^{p_3}\}, \emptyset)\} \qquad E_4^\sharp = \{\mathtt{x} \mapsto (\{o^{p_1}\}, \emptyset), \mathtt{y} \mapsto (\{o^{p_2}, o^{p_3}\}, \emptyset)\}$$

$$H_1^\sharp = \{(o^{p_1}, \mathtt{f}) \mapsto \{o^{p_2}\}, \emptyset\}$$

$$\textsc{Asg}\ \dfrac{\textsc{Fld}\ \dfrac{\textsc{Var}\ \dfrac{E_1^\sharp[\mathtt{x}] \sqsubseteq (\{o^{p_1}\}, \emptyset)}{E_1^\sharp, H_1^\sharp, \mathtt{x} \to^\sharp \{o^{p_1}\}, \emptyset} \qquad H_1^\sharp[\{o^{p_1}\}][\mathtt{f}] = (\{o^{p_2}\}, \emptyset)}{E_1^\sharp, H_1^\sharp, \mathtt{x.f} \to^\sharp \{o^{p_2}\}, \emptyset}}{E_1^\sharp, H_1^\sharp, \mathtt{y = x.f} \to^\sharp E_2^\sharp, H_1^\sharp, \{o^{p_2} \looparrowleft \mathtt{y}\}}$$

$$\dfrac{\vdots \qquad \dfrac{\dfrac{}{E_1^\sharp, H_1^\sharp, \mathtt{\{\}}^{p_3} \to^\sharp \{o^{p_3}\}, \emptyset}\ \textsc{Obj}}{E_1^\sharp, H_1^\sharp, \mathtt{y = \{\}} \to^\sharp E_3^\sharp, H_1^\sharp, \{o^{p_3} \looparrowleft \mathtt{y}\}}\ \textsc{Asg}}{E_1^\sharp, H_1^\sharp, \mathtt{if\ false\ then\ y = x.f\ else\ y = \{\}} \to^\sharp E_4^\sharp, H_1^\sharp, \{\{o^{p_2}, o^{p_3}\} \looparrowleft \mathtt{y}\}}\ \textsc{If}$$

$$\dfrac{\textsc{FldAsg}\ \dfrac{\textsc{Var}\ \dfrac{E_1^\sharp[\mathtt{x}] \sqsubseteq (\{o^{p_1}\}, \emptyset)}{E_1^\sharp, \bot, \mathtt{x}^{p_2'} \to^\sharp \{o^{p_1}\}, \left\{\mathtt{x}^{p_2'}\right\}} \qquad \dfrac{}{E_1^\sharp, \bot, \mathtt{\{\}}^{p_2} \to^\sharp \{o^{p_2}\}, \emptyset}\ \textsc{Obj}}{E_1^\sharp, \bot, \mathtt{x}^{p_2'}\mathtt{.f = \{\}}^{p_2} \to^\sharp E_1^\sharp, H_1^\sharp, \{o^{p_2} \looparrowleft o^{p_1}\mathtt{.f}\}} \qquad \vdots}{E_1^\sharp, \bot, \mathtt{x.f = \{\};if\ false\ then\ y = x.f\ else\ y = \{\}} \to^\sharp E_4^\sharp, H_1^\sharp, \{o^{p_2} \looparrowleft o^{p_1}\mathtt{.f}, \{o^{p_2}, o^{p_3}\} \looparrowleft \mathtt{y}\}}\ \textsc{Seq}$$

$$\dfrac{\textsc{Asg}\ \dfrac{\textsc{Obj}\ \dfrac{}{\bot, \bot, \mathtt{\{\}}^{p_1} \to^\sharp \{o^{p_1}\}, \emptyset}}{\bot, \bot, \mathtt{x = \{\}}^{p_1} \to^\sharp E_1^\sharp, \bot, \{o^{p_1} \looparrowleft \mathtt{x}\}} \qquad \vdots}{\bot, \bot, \mathtt{x = \{\}}^{p_1}\mathtt{;x.f = \{\}}^{p_2}\mathtt{;if\ false\ then\ y = x.f\ else\ y = \{\}}^{p_3} \to^\sharp E_4^\sharp, H_1^\sharp, \{o^{p_1} \looparrowleft \mathtt{x}, o^{p_2} \looparrowleft o^{p_1}\mathtt{.f}, \{o^{p_2}, o^{p_3}\} \looparrowleft \mathtt{y}\}}\ \textsc{Seq}$$

Figure 14: Analysis Example

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 analyser is then defined by an extractable function of similar type (excepting the final "$\rightarrow$ `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 5.2: if from an empty heap, a program reduces to a heap E_τ, H_τ and flows Δ_τ, then if from the $\perp$ abstraction, a program reduces to $E^\sharp$, $H^\sharp$ and abstract flows $\Delta^\sharp$; that is,

$$[\,], \emptyset, [\,], \emptyset, \emptyset, \underline{s} \rightarrow \tau, M_\tau, \Delta_\tau, E_\tau, H_\tau \qquad \text{and} \qquad \perp, \perp, \underline{s} \rightarrow^\sharp E^\sharp, H^\sharp, \Delta^\sharp$$

then $E \prec E^\sharp$, $H \prec H^\sharp$ and $\Delta_\tau \prec \Delta^\sharp$. This can be followed on [2].

6 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 intensional 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.

The semantics and its abstractions lend themselves well to being implemented in the proof assistant Coq. 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].

We hope to report on this in the next Festschrift to David Schmidt.

$$
\begin{aligned}
\mathtt{App}_o(\textsc{Name}, \mathrm{PP}) &= \textsc{Name}/\mathrm{PP} && \text{for normal names} \\
\mathtt{App}_o(\textsc{Name}, \mathrm{PP}) &= \mathrm{PP} && \text{for extended names} \\
\mathtt{App}_i(\textsc{Seq}, \mathrm{PP}) &= \textsc{Seq}1/\mathrm{PP} \\
\mathtt{App}_i(\textsc{Seq}1, \mathrm{PP}) &= \textsc{Seq}2/\mathrm{PP} \\
\mathtt{App}_i(\textsc{If}, \mathrm{PP}) &= \textsc{If}\mathrm{E}/\mathrm{PP} \\
\mathtt{App}_i(\textsc{IfTrue}, \mathrm{PP}) &= \textsc{If}1/\mathrm{PP} \\
\mathtt{App}_i(\textsc{IfFalse}, \mathrm{PP}) &= \textsc{If}2/\mathrm{PP} \\
\mathtt{App}_i(\textsc{While}, \mathrm{PP}) &= \textsc{While}\mathrm{E}/\mathrm{PP} \\
\mathtt{App}_i(\textsc{WhileTrue}1, \mathrm{PP}) &= \textsc{While}\mathrm{S}/\mathrm{PP} \\
\mathtt{App}_i(\textsc{Asg}, \mathrm{PP}) &= \textsc{Asg}\mathrm{E}/\mathrm{PP} \\
\mathtt{App}_i(\textsc{FldAsg}, \mathrm{PP}) &= \textsc{FldAsg}1/\mathrm{PP} \\
\mathtt{App}_i(\textsc{FldAsg}1, \mathrm{PP}) &= \textsc{FldAsg}2/\mathrm{PP} \\
\mathtt{App}_i(\textsc{Del}, \mathrm{PP}) &= \textsc{Del}\mathrm{E}/\mathrm{PP} \\
\mathtt{App}_i(\textsc{Bin}, \mathrm{PP}) &= \textsc{Bin}1/\mathrm{PP} \\
\mathtt{App}_i(\textsc{Bin}1, \mathrm{PP}) &= \textsc{Bin}2/\mathrm{PP} \\
\mathtt{App}_i(\textsc{Fld}, \mathrm{PP}) &= \textsc{Fld}\mathrm{E}/\mathrm{PP} \\
\mathtt{App}_i(\textsc{Name}, \mathrm{PP}) &= \mathrm{PP} && \text{otherwise} \\
\mathtt{Push}_o(\textsc{Name}) &= \textsc{Name}_o :: [] && \text{for normal names} \\
\mathtt{Push}_o(\textsc{Name}) &= [] && \text{for extended names} \\
\mathtt{Push}_i(\textsc{Seq}1) &= \textsc{Seq}_o :: [] \\
\mathtt{Push}_i(\textsc{IfTrue}) &= \textsc{If}_o :: [] \\
\mathtt{Push}_i(\textsc{IfFalse}) &= \textsc{If}_o :: [] \\
\mathtt{Push}_i(\textsc{WhileTrue}1) &= \textsc{While}_o :: [] \\
\mathtt{Push}_i(\textsc{WhileTrue}2) &= \textsc{While}_o :: [] \\
\mathtt{Push}_i(\textsc{Asg}1) &= \textsc{App}_o :: [] \\
\mathtt{Push}_i(\textsc{FldAsg}1) &= \textsc{FldAsg}_o :: [] \\
\mathtt{Push}_i(\textsc{FldAsg}2) &= \textsc{FldAsg}_o :: [] \\
\mathtt{Push}_i(\textsc{Bin}1) &= \textsc{Bin}_o :: [] \\
\mathtt{Push}_i(\textsc{Bin}2) &= \textsc{Bin}_o :: [] \\
\mathtt{Push}_i(\textsc{Del}1) &= \textsc{Del}_o :: [] \\
\mathtt{Push}_i(\textsc{Fld}1) &= \textsc{Fld}_o :: [] \\
\mathtt{Push}_i(\textsc{Name}) &= [] && \text{otherwise}
\end{aligned}
$$

Figure 15: Helper functions to convert traces to program points

References

[1] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt & G. Smith (2012): *JSCert: Certified JavaScript*. `http://jscert.org/`.

[2] M. Bodin, T. Jensen & A. Schmitt (2013): *Pretty-Big-Step Semantics-based Certified Abstract Interpretation, Source Code*. `http://www.irisa.fr/celtique/aschmitt/research/owhileflows/`.

[3] Arthur Charguéraud (2013): *Pretty-big-step semantics*. In: *Proceedings of the 22nd European Symposium on Programming (ESOP 2013)*, Springer, pp. 41–60, doi:10.1007/978-3-642-37036-6_3.

[4] P. Cousot (1999): *The Calculational Design of a Generic Abstract Interpreter*. In M. Broy & R. Steinbrüggen, editors: *Calculational System Design*, NATO ASI Series F. IOS Press, Amsterdam.

[5] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet & Ryan Berg (2011): *Saving the world wide web from vulnerable JavaScript*. In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, ACM Press, pp. 177–187, doi:10.1145/2001420.2001442.

[6] Daniel Hedin & Andrei Sabelfeld (2012): *Information-Flow Security for a Core of JavaScript*. In: *Proc. of the 25th Computer Security Foundations Symp. (CSF'12)*, IEEE, pp. 3–18, doi:10.1109/CSF.2012.19.

[7] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen & David Schmidt (2006): *Automata-based Confidentiality Monitoring*. In: *Proc. of the Annual Asian Computing Science Conference*, Springer LNCS vol. 4435, pp. 75–89, doi:10.1007/978-3-540-77505-8_7.

[8] Jan Midtgaard & Thomas Jensen (2008): *A Calculational Approach to Control-Flow Analysis by Abstract Interpretation*. In: *Proc. of the 15th Static Aanalysi Symposium, LNCS* 5079, Springer Verlag, pp. 347–362, doi:10.1007/978-3-540-69166-2_23.

[9] Jan Midtgaard & Thomas Jensen (2009): *Control-flow analysis of function calls and returns by abstract interpretation*. In: *Proc. of the 14th ACM international conference on Functional programming*, ICFP '09, ACM, pp. 287–298, doi:10.1145/1596550.1596592.

[10] David Pichardie (2008): *Building certified static analysers by modular construction of well-founded lattices*. In: *Proc. of the 1st International Conference on Foundations of Informatics, Computing and Software (FICS'08), Electronic Notes in Theoretical Computer Science* 212, pp. 225–239, doi:10.1016/j.entcs.2008.04.064.

[11] D.A. Schmidt (1995): *Natural-semantics-based abstract interpretation (preliminary version)*. In: *Proc. 2d Static Analysis Symposium (SAS'95)*, Springer LNCS vol. 983, pp. 1–18, doi:10.1007/3-540-60360-3_28.

[12] E. Schwartz, T. Avgerinos & D. Brumley (2010): *All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask)*. In: *Proc. of the 2010 IEEE Symp. on Security and Privacy*, doi:10.1109/SP.2010.26.

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

[14] David Van Horn & Matthew Might (2010): *Abstracting abstract machines*. In: *Proc. of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, ACM, pp. 51–62, doi:10.1145/1995376.1995399.

[15] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel & G. Vigna (2007): *Cross-site scripting prevention with dynamic data tainting and static analysis*. In: *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, 42.