Paper **two**

On-the-fly Inlining of Dynamic Dependency Monitors for Secure Information Flow

Luciano Bello and Eduardo Bonelli

Information flow analysis (IFA) in the setting of programming languages is steadily veering towards the adoption of dynamic techniques. This is particularly attractive for scripting languages for web applications programming. A common manifestation of dynamic techniques is that of run-time monitors, which should block program execution in the presence of an insecure run. Significant efforts are still required before practical, scalable monitors for secure IFA of industrial scale languages such as JavaScript can be achieved. Such monitors ideally should compensate for the absence of the traces they do not track, should not require modifications of the VM and should provide a fair compromise between security and usability among other things. This paper discusses on-the-fly inlining of monitors that track dependencies as a prospective candidate.

Formal Aspects of Security and Trust Lecture Notes in Computer Science
Volume 7140, 2012, pp 55-69

1 Introduction

Secure IFA in the setting of programming languages [17] is steadily veering towards the adoption of dynamic techniques [2, 3, 11, 12, 15, 18–20]. There are numerous reasons for this among which we can mention the following. First they are attractive from the perspective of scripting languages for the web such as JavaScript which are complex subjects of study for static-based techniques. Second, they allow dealing with inherently run-time issues such as dynamic object creation and `eval` run-time code evaluation mechanism. Last but not least, recent work has suggested that a mix of both static and dynamic flavors of IFA will probably strike the balance between correct, usable and scalable tools in practice.

Language-based secure IFA is achieved by assigning variables a security level such as public or secret and then determining whether those that are labeled as secret affect the contents of public ones during execution. This security property is formalised as *noninterference*. In this paper, we are concerned in particular with *termination-insensitive noninterference* [17, 21]: starting with two identical run-time states that only differ in the contents of secret variables, the final states attained after any given pair of terminating runs differ at most in the contents of the secret variables. Thus in this paper we ignore covert channels.

IFA Monitors. Dynamic IFA monitors track the security level of data during execution. If the level of the data contained in a variable may vary during execution we speak of a *flow-sensitive* analysis [13]. Flow-sensitivity provides a more flexible setting than the flow-insensitive one when it comes to practical enforcement of security policies. Purely dynamic flow-sensitive monitors can leak information related to control flow [16]. Such monitors keep track of the security label of each variable and update these labels when variables are assigned. Information leak occurs essentially because these monitors cannot track traces that are not taken (such as branches that are not executed). Consider the example in Fig. 1 taken from [18] (the subscripts may be ignored for now). Assume that `sec` is initially labeled as secret. The monitor labels variables `tmp` and `pub` as public (since constants are considered public values) after executing the first two assignments. If `sec` is nonzero, the label of `tmp` is updated to secret since the assignment in line 3 depends on the value of `sec`. The "then" branch of the second conditional is not executed. If `sec` is zero, then the "then" branch of the second conditional is executed. Either way, the value

```
1 tmp := 1; pub := 1;
2 if_p1 sec then
3     tmp := 0;
4 if_p2 tmp then
5     pub := 0;
6 ret_p3 (pub)
```

Fig. 1. Monitor attack, from [16]

of sec, a secret variable, leaks to the returned value and the monitor is incapable of detecting it.

Purely dynamic flow-sensitive monitors must therefore be supplied with additional information in order to compensate for this deficiency. One option is to supply the monitor with information on the branches not taken. This is the approach taken for example in [16]. In the example of Fig. 1, when execution reaches the conditional in line 4, although the "then" branch is not taken the label of pub would be updated to secret since this variable would have been written in the branch that was not taken and that branch depends on a secret variable. In order to avoid the need for performing static analysis [4] proposed the *no-sensitive upgrade* scheme where execution gets stuck on attempting to assign a public variable in a secret context. Returning to our example, when sec is nonzero and execution reaches the assignment in line 3, it would get stuck. A minor variant of that scheme is the *permissive upgrade* [5] scheme where, although assignment of public variables in a secret contexts is allowed, branching on expressions that depend on such variables is disallowed. In our example, when sec is nonzero and execution reaches the assignment in line 3, it would be allowed. However, execution would get stuck at line 4. As stated in [6], not only can these schemes reject secure programs, but also their practical applicability is yet to be determined.

Dynamic dependency tracking. An alternative to supplying a monitor that is flow-sensitive with either static information or resorting to the *no-sensitive upgrade* or *permissive upgrade* schemes is *dependency analysis* [19]. Shroff et al. introduce a run-time IFA monitor that assigns program points to branches and maintains a cache of dependencies of *indirect flows* towards program points and a cache of *direct flows* towards program points. These caches are called κ and δ, respectively. The former is persistent over successive runs. Indeed, when execution takes a branch which has hitherto been unexplored, the monitor collects information associated with it and adds it to the current indirect dependencies. Thus, although an initial run may not spot an insecure flow, it will eventually be spotted in subsequent runs.

In order to illustrate this approach, we briefly revisit the example of Fig. 1 (further details are supplied in Sec. 2). We abbreviate the security level "secret" with the letter H and "public" with L, as is standard. Values in this setting are tagged with both a set of dependencies (set of program points p, p_i, etc.) and a security level. When the level is not important but the dependency is, we annotate the value just with the dependency: e.g. 0^p (in our example dependencies are singletons, hence we write p rather that $\{p\}$). Likewise, when it is the security level that is relevant we write for e.g. 0^L or 0^H. After initialization of the variables and their security levels, the guard in line 2 is checked. Here two operations take place. First

	First run					Second run				
line	1	2	3	4	6	1	2	4	5	6
`sec`	1^H	1^H	1^H	1^H	1^H	0^H	0^H	0^H	0^H	0^H
`tmp`	1^L	1^L	0^{p1}	0^{p1}	0^{p1}	1^L	1^L	1^L	1^L	1^L
`pub`	1^L	1^L	1^L	1^L	1^L	1^L	1^L	1^L	0^{p2}	0^{p2}
$p1$		H	H	H	H		H	H	H	H
$p2$				L	L			L	L	L
$p3$					L					L
ret					1^{p3}					0^{p3}
κ				$p1 \uparrow p2$	$p1 \uparrow p2$	$p1 \uparrow p2$	$p1 \uparrow p2$	$p1 \uparrow p2$	$p1 \uparrow p2$	$p1 \uparrow p2 \uparrow p3$

Table 1. Dependency tracking on two runs of Fig. 1.

the level of program point $p1$ is set to H reflecting a direct dependency of $p1$ with `sec`. This is stored in δ, the cache of direct dependencies. The body of the condition is executed (since the guard is true) and `tmp` is updated to 0^{p1}, indicating that the assigned value depends on the guard in $p1$. When the guard from the fourth line is evaluated, in κ (the cache of indirect dependencies, which is initially empty) the system stores that $p2$ depends on $p1$ (written $p_2 \mapsto p_1$), since the value of the variable involved in the condition depends on $p1$. At this point `pub` has the same value, namely 1, as `sec`, and hence leaks this fact. The key of the technique is to retain κ for future runs. Suppose that in a successive run `sec` is 0^H. The condition from line 2 is evaluated and the direct dependency $p1 \mapsto H$ is registered in δ. The third line is skipped and the condition pointed by $p2$ is checked. This condition refers to `tmp` whose value is 1^L. The body in line 5 is executed and `pub` is updated with 0^{p2}. At this point, it is possible to detect that `pub` depends on H as follows: variable `pub` depends on $p2$ (using the cache κ); $p2$ depends on $p1$; and the level of the latter program point is H according to the direct dependency cache. Table 1 summarizes both runs as explained above.

Inlining Monitors. An alternative to implementing a monitor as part of a custom virtual machine or modifying the interpreter [8, 10] is to resort to *inlining* [6, 9, 14, 20]. The main advantage behind this option is that no modification of the host run-time environment is needed, hence achieving a greater degree of portability. This is particularly important in web applications. Also, such an inlining can take place either at the browser level or at the proxy level, thus allowing dedicated hardware to inline system wide. Magazinius et al. [14] introduce the notion of *on-the-fly* inlining. The monitor in charge of enforcing the security policy uses a function *trans* to inline a monitored code. This function is also available

at run-time and can be used to transform code only known immediately before its execution. The best example of this dynamic source is the `eval` primitive.

Contribution. This paper takes the first steps in *inlining* the dependency analysis [19] as a viable alternative to supplying a flow-sensitive monitor with either static information or resorting to the *no-sensitive upgrade* or *permissive upgrade* schemes. Given that we aim at applying our monitor to JavaScript, we incorporate `eval` into our analysis. Since the code evaluated by `eval` is generated at run-time and, at the same time, the dependency tracking technique requires that program points be persisted, we resort to hashing to associate program points to dynamically generated code. We define and prove correct an on-the-fly inlining transformation, in the style of [14], of a security monitor which is based on dependency analysis that incorporates these extensions.

Paper Structure. Sec. 2 recasts the theory of [19] originally developed for a lambda calculus with references to a simple imperative language. Sec. 3 briefly describes the target language of the inlining transformation and defines the transformation itself. Sec. 4 extends the transformation to `eval`. The properties of the transformation are developed in Sec. 5. Finally, we present conclusions and possible lines of additional work. A prototype in Python is available at [1], as well as the formal definitions and proof.

2 Dependency Analysis for a Simple Imperative Language

We adapt the dependency analysis framework of Shroff et al. [19] to a simple imperative language $\mathcal{W}^{deps}$ prior to considering an inlining transformation for it. Its syntax is given in Fig. 2. There are two main syntactic categories, *expressions* and *commands*. An expression is either a variable, a labeled value, a binary expression, an application (of a user-defined function to an argument expression) or a case expression. A *labeled value* is a tuple consisting of a value (an integer or a string), a set of program points and a security level. We assume a set of *program points* $p_1, p_2, \ldots$. *Security levels* are taken from a lattice $(\mathcal{L}, \sqsubseteq)$. We write $\sqcup$ for the supremum. Commands are standard. For technical purposes, it is convenient to assume that the program to be executed ends in a return command `ret`, and that moreover this is the unique occurrence of `ret` in the program. Note however that this assumption may be dropped at the expense of slightly complicating the statement of *information leak* (Def. 1) and *delayed leak detection* (Prop. 1). The `while`, `if` and `ret` commands are sub-scripted with a program point.

The operational semantics of $\mathcal{W}^{deps}$ is defined in terms of a binary relation over *configurations*, tuples of the form $\langle E, \kappa, \delta, \pi, \mu, c\rangle$ where E

$$
\begin{array}{rll}
P, \pi & ::= \{\overline{p}\} & \text{(set of ppids, program counter)} \\
v & ::= i \,\big|\, s & \text{(value)} \\
\sigma & ::= \langle v, P, L\rangle & \text{(labeled value)} \\
e & ::= x \,\big|\, \sigma \,\big|\, e \oplus e \,\big|\, f(e) \,\big|\, \texttt{case}\ e\ \texttt{of}\ (e : e)^{+} & \text{(expression)} \\
c & ::= \texttt{skip} \,\big|\, x := e \,\big|\, \texttt{let}\, x = e\, \texttt{in}\, c \,\big|\, c; c \,\big|\, \texttt{while}_p\, e\, \texttt{do}\, c & \text{(command)} \\
 & \quad \big|\ \texttt{if}_p\, e\, \texttt{then}\, c\, \texttt{else}\, c \,\big|\, \texttt{ret}_p(e) \,\big|\, \mathit{stop} & \\
E & ::= \emptyset \,\big|\, f(x) \doteq e; E & \text{(expr. environment)} \\
\mu & ::= \{\overline{x \mapsto \sigma}\} & \text{(memory)} \\
\kappa & ::= \{\overline{p \mapsto P}\} & \text{(cache of dependencies)} \\
\delta & ::= \{\overline{p \mapsto L}\} & \text{(cache of direct flows)}
\end{array}
$$

Fig. 2. Syntax of $\mathcal{W}^{deps}$

is an *expression environment*, κ is a *cache of indirect flows*, δ is a *cache of direct flows*, π is the *program counter* (a set of program points), μ is a (partial) function from variables to labeled values and c is the current command. We use $\mathcal{D}, \mathcal{D}_i, etc$ for configurations. We write $\mu[x \mapsto \sigma]$ for the memory that behaves as μ except on x to which it associates σ. Also, $\mu \setminus x$ undefines μ on x. The domain of μ includes a special variable *ret* that holds the return value. The expression environment declares all available user-defined functions. We omit writing it in configurations and assume it is implicitly present. *Expression evaluation* is introduced in terms of *closed expression evaluation* and then *(open) expression evaluation*. *Closed expression evaluation* is defined as follows,

$$
\begin{aligned}
\mathcal{I}(\langle v, P, L\rangle) &\stackrel{def}{=} \langle v, P, L\rangle \\
\mathcal{I}(f(e)) &\stackrel{def}{=} \hat{f}(\mathcal{I}(e)) \\
\mathcal{I}(\texttt{case}\ e\ \texttt{of}\ \boldsymbol{e : e'}) &\stackrel{def}{=} \hat{\mathit{case}}\ \mathcal{I}(e)\ \mathit{of}\ \boldsymbol{e'_i : e'} \\
\mathcal{I}(e_1 \oplus e_2) &\stackrel{def}{=} \mathcal{I}(e_1) \hat{\oplus} \mathcal{I}(e_2)
\end{aligned}
$$

where we assume
$\hat{f}(\langle v, P, L\rangle) \stackrel{def}{=} \mathcal{I}(e[x := \langle v, P, L\rangle])$, if $f(x) \doteq e \in E$; $\hat{\mathit{case}}\ \langle u, P, L\rangle\ \mathit{of}\ \boldsymbol{e : e'} \stackrel{def}{=} \langle v, P \cup P', L \sqcup L'\rangle$ if u matches[1] e_i with substitution σ and $\mathcal{I}(\sigma e'_i) = \langle v, P', L'\rangle$; and $\langle i_1, P_1, L_1\rangle \hat{\oplus} \langle i_2, P_2, L_2\rangle \stackrel{def}{=} \langle i_1 \oplus i_2, P_1 \cup P_2, L_1 \sqcup L_2\rangle$. We assume that in a case-expression exactly one branch applies. Moreover, we leave it to the user to guarantee that user-defined functions are terminating.

Given a memory μ, the *variable replacement* function, also written μ, applies to expressions: it traverses expressions replacing variables by their values. It is defined only if the free variables of its argument are in

[1] Here we mean the standard notion of matching of a closed term e_1 against an algebraic pattern e_2; if successful, it produces a substitution σ for the variables of e_2 s.t. $\sigma(e_2) = e_1$.

the domain of μ. Finally, *open expression evaluation* is defined as $\mathcal{I} \circ \mu$, the composition of $\mathcal{I}$ and μ, and abbreviated $\hat{\mu}$.

The *reduction judgement* $\mathcal{D}_1 \rightarrowtail \mathcal{D}_2$ states that the former configuration *reduces* to the latter. This judgement is defined by means of the *reduction schemes* of Fig. 3. It is a mixed-step semantics in the sense that it mixes both small and big-step semantics. Thus $\mathcal{D}_1 \rightarrowtail \mathcal{D}_2$ may be read as $\mathcal{D}_2$ may be obtained from $\mathcal{D}_1$ in some number of small reduction steps. We write $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$ for the n-fold composition of $\rightarrowtail$. Rule SKIP is straightforward; *stop* is a run-time command to indicate the end of execution. The LET scheme is standard; we resort to $[x := e]$ for capture avoiding substitution of all occurrences of the free variable x by e. The ASSIGN scheme updates memory μ by associating x with the labeled value of e, augmenting the indirect dependencies with the program counter π. We omit the description of WHILE-T and WHILE-F and describe the schemes for the conditional (which are similar). If the condition is true (the reduction scheme when the condition is false, namely IF-F, is identical except that it reduces c_2, hence it is omitted), then before executing the corresponding branch the configuration is updated. First the program counter is updated to include the program point p. A new dependency is added to the cache of indirect dependencies for p, namely $\pi \cup P$, indicating that there is an *indirect* flow from the current security context under which the conditional is being reduced and the condition e (via its dependencies). The union operator $\kappa \uplus \kappa'$ is defined as κ'' iff κ'' is the smallest cache such that $\kappa, \kappa' \leq \kappa''$. Here the ordering relation on caches is defined as $\kappa \leq \kappa'$ iff $\forall p \in dom(\kappa).\kappa(p) \subseteq \kappa'(p)$. Finally, the security level L of the condition is recorded in δ', reflecting the *direct* dependency of the branch on e. The scheme for `ret` updates the cache of indirect dependencies indicating that there is an *indirect* flow from the program counter and e (via its dependencies) towards the value that is returned. Finally, we note that $\langle \kappa, \delta, \pi, \mu, c \rangle \rightarrowtail \langle \kappa', \delta', \pi', \mu', c' \rangle$ implies $\kappa \leq \kappa'$ and $\pi' = \pi$.

2.1 Properties

Delayed leak detection (Prop. 1), the main property that the monitor enjoys, is presented in this section. Before doing so however, we require some definitions. The transitive closure of cache look-up is defined as $\kappa(p) \overset{def}{=} P \cup \kappa(P)^+$, where $\kappa(p) = P$. Suppose $P = \{p_1, \ldots, p_k\}$. Then $\kappa(P) \overset{def}{=} \bigcup_{i \in 1..k} k(p_i)$ and $\kappa(P)^+ \overset{def}{=} \bigcup_{i \in 1..k} k(p_i)^+$. We define $\mathsf{secLevel}^{\kappa,\delta} P \overset{def}{=} \delta(P \cup \kappa(P)^+)$, the join of all security levels associated to the transitive closure of P according to the direct dependencies recorded in δ. We write $\mu[\overline{x_k \mapsto \langle v_k, \emptyset, L_{high} \rangle}]$ for $\mu[x_1 \mapsto \langle v_1, \emptyset, L_{high} \rangle] \ldots [x_k \mapsto \langle v_k, \emptyset, L_{high} \rangle]$. We fix L_{low} and L_{high} to be any two distinct levels. A terminating run leaks information via its return value, if this return value is visible to an attacker as determined by the schemes in Fig. 3 and there is another run of the

$$\frac{}{\langle \kappa, \delta, \pi, \mu, \texttt{skip} \rangle \rightarrowtail \langle \kappa, \delta, \pi, \mu, stop \rangle} \textsc{Skip}$$

$$\frac{\langle \kappa, \delta, \pi, \mu[z \mapsto \hat{\mu}(e)], c[x := z] \rangle \overset{n}{\rightarrowtail} \langle \kappa', \delta', \pi, \mu', stop \rangle \quad z \text{ fresh}}{\langle \kappa, \delta, \pi, \mu, \texttt{let}\, x = e\, \texttt{in}\, c \rangle \rightarrowtail \langle \kappa', \delta', \pi, \mu' \setminus z, stop \rangle} \textsc{Let}$$

$$\frac{\langle \kappa, \delta, \pi, \mu, c_1 \rangle \overset{n}{\rightarrowtail} \langle \kappa', \delta', \pi, \mu', stop \rangle}{\langle \kappa, \delta, \pi, \mu, c_1; c_2 \rangle \rightarrowtail \langle \kappa', \delta', \pi, \mu', c_2 \rangle} \textsc{Seq}$$

$$\frac{\hat{\mu}(e) = \langle v, P, L \rangle \quad \mu' = \mu[x \mapsto \langle v, P \cup \pi, L \rangle]}{\langle \kappa, \delta, \pi, \mu, x := e \rangle \rightarrowtail \langle \kappa, \delta, \pi, \mu', stop \rangle} \textsc{Assign}$$

$$\frac{\begin{array}{c}\hat{\mu}(e) = \langle i, P, L \rangle \quad i \neq 0 \quad \pi' = \pi \cup \{p\} \quad \kappa' = \kappa \uplus \{p \mapsto \pi \cup P\} \\ \delta' = \delta \uplus \{p \mapsto L\} \quad \langle \kappa', \delta', \pi', \mu, c \rangle \overset{n}{\rightarrowtail} \langle \kappa'', \delta'', \pi', \mu'', stop \rangle\end{array}}{\langle \kappa, \delta, \pi, \mu, \texttt{while}_p\, e\, \texttt{do}\, c \rangle \rightarrowtail \langle \kappa'', \delta'', \pi, \mu', \texttt{while}_p\, e\, \texttt{do}\, c \rangle} \textsc{While-T}$$

$$\frac{\hat{\mu}(e) = \langle 0, P, L \rangle \quad \kappa' = \kappa \uplus \{p \mapsto \pi \cup P\} \quad \delta' = \delta \uplus \{p \mapsto L\}}{\langle \kappa, \delta, \pi, \mu, \texttt{while}_p\, e\, \texttt{do}\, c \rangle \rightarrowtail \langle \kappa', \delta', \pi, \mu, stop \rangle} \textsc{While-F}$$

$$\frac{\begin{array}{c}\hat{\mu}(e) = \langle i, P, L \rangle \quad i \neq 0 \quad \pi' = \pi \cup \{p\} \quad \kappa' = \kappa \uplus \{p \mapsto \pi \cup P\} \\ \delta' = \delta \uplus \{p \mapsto L\} \quad \langle \kappa', \delta', \pi', \mu, c_1 \rangle \overset{n}{\rightarrowtail} \langle \kappa'', \delta'', \pi', \mu', stop \rangle\end{array}}{\langle \kappa, \delta, \pi, \mu, \texttt{if}_p e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \rangle \rightarrowtail \langle \kappa'', \delta'', \pi, \mu', stop \rangle} \textsc{If-T}$$

$$\frac{\hat{\mu}(e) = \langle v, P, L \rangle \quad \kappa' = \kappa \uplus \{p \mapsto \pi \cup P\} \quad \delta' = \delta \uplus \{p \mapsto L\}}{\langle \kappa, \delta, \pi, \mu, \texttt{ret}_p(e) \rangle \rightarrowtail \langle \kappa', \delta', \pi, \mu[ret \mapsto \langle v, P \cup \pi, L \rangle], stop \rangle} \textsc{Ret}$$

Fig. 3. Mixed-step semantics for $\mathcal{W}^{deps}$

same command, whose initial memory differs only in secret values w.r.t. that of the first run, that produces a different return value. Moreover, this second run has the final cache of indirect dependencies of the first run (κ_1) as its *initial* cache of indirect dependencies.

Definition 1 (Information Leak [19]). *Let* $\mu_0 \overset{def}{=} \mu[\overline{x_k \mapsto \langle v_k, \emptyset, L_{high} \rangle}]$ *for some memory* μ*. A run* $\langle \kappa_0, \delta_0, \pi, \mu_0, c \rangle \overset{n_1}{\rightarrowtail} \langle \kappa_1, \delta_1, \pi, \mu_1, stop \rangle$ leaks information w.r.t. security level L_{low}*, with* $L_{high} \not\sqsubseteq L_{low}$ *iff*

1. $\mu_1(ret) = \langle i_1, P_1, L_1 \rangle$;
2. $(\mathsf{secLevel}^{\kappa_1, \delta_1} P_1) \sqcup L_1 \sqsubseteq L_{low}$; *and*

3. *there exists k labeled values* $\overline{\langle v'_k, \emptyset, L_{high}\rangle}$ *s.t.*
   $\mu'_0 = \mu[\overline{x_k \mapsto \langle v'_k, \emptyset, L_{high}\rangle}]$ *and* $\langle \kappa_1, \delta_0, \pi, \mu'_0, c\rangle \overset{n_2}{\rightarrowtail} \langle \kappa_2, \delta_2, \pi, \mu_2, stop\rangle$ *and* $\mu_2(ret) = \langle i_2, P_2, L_2\rangle$ *with* $i_1 \neq i_2$.

Delayed leak detection is proved in [19] in the setting of a higher-order functional language and may be adapted to our simple imperative language.

Proposition 1. *If*

- $\mu_0 = \mu[\overline{x_k \mapsto \langle v_k, \emptyset, L_k\rangle}]$;
- *the run* $\langle \kappa_0, \delta_0, \pi, \mu_0, c\rangle \overset{n_1}{\rightarrowtail} \langle \kappa_1, \delta_1, \pi, \mu_1, stop\rangle$ *leaks information w.r.t. security level* L_{low}; *and*
- $\mu_1(ret) = \langle i_1, P_1, L_1\rangle$

then there exists $\overline{\langle v'_k, \emptyset, L'_k\rangle}$ *s.t.*

- $\mu'_0 = \mu[\overline{x_k \mapsto \langle v'_k, \emptyset, L'_k\rangle}]$;
- $\langle \kappa_1, \delta_0, \pi, \mu'_0, c\rangle \overset{n_2}{\rightarrowtail} \langle \kappa_2, \delta_2, \pi, \mu_2, stop\rangle$; *and*
- $\mathsf{secLevel}^{\kappa_2,\delta_1} P_1 \not\sqsubseteq L_{low}$.

The labeled values $\overline{\langle v'_k, \emptyset, L'_k\rangle}$ may be either public or secret since, if the first run leaks information, then appropriate input values of any required level must be supplied in order for the second run to gather the necessary dependencies that allow it to detect the leak.

3 Inlining the Dependency Analysis

The inlining transformation *trans* inserts code that allows dependencies to be tracked during execution. The target of the transformation is a simple imperative language we call $\mathcal{W}$ whose syntax is defined as follows:

$$
\begin{array}{rll}
v ::= & i \,|\, s \,|\, P \,|\, L & \text{(value)} \\
e ::= & x \,|\, v \,|\, e \oplus e \,|\, f(e) \,|\, \texttt{case}\ e\ \texttt{of}\ (e : e)^+ & \text{(expression)} \\
c ::= & \texttt{skip} \,|\, c; c \,|\, \texttt{let}\, x = e\, \texttt{in}\, c \,|\, x := e \,|\, \texttt{while}\, e\, \texttt{do}\, c \,| & \text{(command)} \\
 & \,|\, \texttt{if}\, e\, \texttt{then}\, c\, \texttt{else}\, c \,|\, \texttt{ret}(e) \,|\, stop & \\
M ::= & \{\overline{x \mapsto v}\} & \text{(memory)}
\end{array}
$$

In contrast to $\mathcal{W}^{deps}$, it operates on standard, unlabeled values and also includes sets of program points and security levels as values, since they will be manipulated by the inlined monitor. Moreover, branches, loops and return commands are no longer decorated with program points. *Expression evaluation* is defined similarly to $\mathcal{W}^{deps}$. A *$\mathcal{W}$-(run-time) configuration* is an expression of the form $\langle E, M, c\rangle$ (as usual E shall be dropped for the sake of readability) denoted with letters C, C_i, etc. The

```
trans(y) =
 case y of
    "skip": "skip"
    "x:=e":
        "x_L:= lev(" ++ vars("e") ++ ");" ++
        "x_P:= dep(" ++ vars("e") ++ ") | pc;" ++
        "x :=e"
    "let x=e in c":
        "let x=e in " ++
          "x_L:= lev(" ++ vars("e") ++ ");" ++
          "x_P:= dep(" ++ vars("e") ++ ") | pc;" ++
          trans(c)
    "c_1;c_2":
         trans(c_1) ++ ";" ++ trans(c_2)
    # continued below
```

Fig. 4. Inlining transformation (1/2)

small-step[2] semantics of $\mathcal{W}$ commands is standard and hence omitted. We write $C \rightarrow C'$ when C' is obtained from C via a reduction step. The transformation *trans* is a user-defined function that resides in E; when applied to a string it produces a new one. We use double-quotes for string constants and ++ for string concatenation.

We now describe the inlining transformation depicted in Fig. 4 and Fig. 5. The inlining of `skip` is immediate. Regarding assignment $x := e$, the transformation introduces two shadow variables x_P and x_L. The former is for tracking the indirect dependencies of x while the latter is for tracking its security level. As may be perceived from the inlining of assignment, the transformation *trans* is in fact defined together with three other user-defined functions, namely *vars*, *lev* and *dep*. The first extracts the variables in a string returning a new string listing the comma-separated variables. Eg. $vars(\text{"}x \oplus f(2 \oplus y)\text{"})$ would return, after evaluation, the string "x, y". The second user-defined function computes the least upper bound of the security levels of the variables in a string and the last computes the union of the implicit dependencies of the variables in a string. The level of e and its indirect dependencies are registered in x_L and x_P, respectively. In the case of x_P, the current program counter is included by means of the variable *pc*. The binary operator | denotes the union between sets. In contrast to $vars(\text{"}e\text{"})$, which is computed at inlining time, *lev* and *dep* are computed when the inlined code is executed. We close the description of the inlining of assignment by noting that the transformed code adopts *flow-sensitivity* in the sense that the security level of the values stored

[2] Hence not mixed-step but rather the standard notion.

```
1  # continued from above
2  "while_p e do c":
3       "k_p := k_p | dep(" ++ vars("e") ++ ") | pc;" ++
4       "d_p := d_p | lev(" ++ vars("e") ++ ");" ++
5       "while e do " ++
6           "(let pc'= pc in " ++
7             "pc := pc | {p};" ++
8              trans(c) ++
9              "pc := pc';" ++
10             "k_p := k_p | dep(" ++ vars("e") ++ ") | pc;" ++
11             "d_p := d_p | lev(" ++ vars("e") ++ "));"
12 "if_p e then c_1 else c_2":
13       "k_p := k_p | dep(" ++ vars("e") ++ ") | pc;" ++
14       "d_p := d_p | lev(" ++ vars("e") ++ ");" ++
15       "let pc'= pc in " ++
16        "pc := pc | {p};" ++
17         "if e then " ++ trans(c_1) ++ "else" ++ trans(c_2)++ ";" ++
18         "pc := pc'"
19 "ret_p (e)":
20        "k_p := k_p | dep(" ++ vars("e") ++ ") | pc;" ++
21        "d_p := d_p | lev(" ++ vars("e") ++ ");" ++
22         "ret (e)"
```

Fig. 5. Inlining transformation (2/2)

in variables may vary during execution. It should also be noted that rather than resort to the *no sensitive upgrade* discipline of Austin and Flanagan [4] to avoid the attack of Fig. 1 (which is also adopted by [14] in their inlining transformation), the dependency monitor silently tracks dependencies without getting stuck.

The let construct is similar to assignment but also resorts to the let construct of $\mathcal{W}$. Here we incur in an abuse of notation since in practice we expect x_L and x_P to be implemented in terms of dictionaries $L[x]$ and $P[x]$. Hence we assume that the declared variable x also binds the x in x_L and x_P. The inlining of command composition is simply the inlining of each command. In the case of `while` (Fig. 5) first we have to update the current indirect dependencies cache and the cache of direct flows (lines 3 and 4, respectively). This is because evaluation of e will take place at least once in order to determine whether program execution skips the body of the while-loop or enters it. For that purpose we assume that we have at our disposal global variables k_p and d_p, for each program point p in the command to inline. Once inside the body, a copy of the program counter is stored in pc' and then the program counter is updated (line 7) with the program point of the condition of the `while`. Upon completing

the execution of *trans*(c), it is restored and then the dependencies are updated reflecting that a new evaluation of e takes place. The clause for the conditional is similar to the one for `while`. The clause for `ret` follows a similar description.

4 Incorporating `eval`

This section considers the extension of $\mathcal{W}^{deps}$ with the command `eval`(e). Many modern languages, including JavaScript, perform dynamic code evaluation. IFA studies have recently begun including it [2,7,14].

The argument of `eval` is an expression that denotes a string that parses to a program and is generated at run-time. Therefore its set of program points may vary. Since the monitor must persist the cache of indirect flows across different runs, we introduce a new element to $\mathcal{W}^{deps}$-configurations, namely a family of caches indexed by the codomain of a hash function: $\mathcal{K}$ is a mapping from the hash of the source code to a cache of indirect flows (i.e. $\mathcal{K} ::= \{\overline{h \mapsto \kappa}\}$ where h are elements of the codomain of the hash function). $\mathcal{W}^{deps}$-configurations thus take the new form $\langle \mathcal{K}, \kappa, \delta, \pi, \mu, c \rangle$. The reduction schemes of Fig. 3 are extended by (inductively) dragging along the new component; the following new reduction scheme, Eval, will be in charge of updating it. A quick word on notation before proceeding: we write $\mathcal{K}(h)$ for the cache of indirect dependencies of s, where s is a string that parses to a command and $\mathsf{hash}(s) = h$. Also, given a cache κ and a command c, the expression $\kappa|_c$ is defined as follows (where $\mathsf{programPoints}(c)$ is the set of program points in c): $\kappa|_c \stackrel{def}{=} \{p \mapsto P \mid p \in \mathsf{programPoints}(c) \wedge \kappa(p) = P\}$. The Eval reduction scheme is as follows:

$$\frac{\begin{array}{c}\hat{\mu}(e) = \langle s, P, L \rangle \quad \pi' = \pi \cup \{p\} \quad h = \mathsf{hash}(s) \\ \kappa' = \kappa \uplus \mathcal{K}(h) \uplus \{p \mapsto \pi \cup P\} \quad \delta' = \delta \uplus \{p \mapsto L\} \\ \langle \mathcal{K}, \kappa', \delta', \pi', \mu, \mathsf{parse}(s) \rangle \overset{n}{\rightsquigarrow} \langle \mathcal{K}', \kappa'', \delta'', \pi', \mu'', stop \rangle\end{array}}{\langle \mathcal{K}, \kappa, \delta, \pi, \mu, \mathtt{eval}_p(e) \rangle \rightsquigarrow \langle \mathcal{K}'[h \mapsto \mathcal{K}'(h) \uplus \kappa''|_{\mathsf{parse}(s)}], \kappa'', \delta'', \pi, \mu'', stop \rangle} \ \textsc{Eval}$$

This reduction scheme looks up the cache for the hash of s (that is $\mathcal{K}(h)$) and then adds it to the current indirect cache. Also added to this cache is the dependency of the code to be evaluated on the level of the context and the dependencies of the expression e itself. The resulting cache is called κ'. After reduction, $\mathcal{K}'$ is updated with any new dependencies that may have arised (recursively[3]) for s (written $\mathcal{K}'(h)$ above) together with the set of program points affected to parse(s) by the outermost (i.e. non-recursive) reduction (written $\kappa''|_{\mathsf{parse}(s)}$ above). Eval may be inlined as indicated in Fig. 6 where *dep*(k, e) represents the user-defined

[3] When $\mathsf{parse}(s)$ itself has an occurrence of `eval` whose argument evaluates to s.

```
"eval_p(e)":
  "let pc′ = pc in " ++
     "pc := pc | {p}" ++
     "k_p := k_p | dep(" ++ vars("e") ++ ") | pc′" ++
     "d_p := d_p | lev(" ++ vars("e") ++ ")" ++
     "let h = hash(e) in " ++
        "k := k | K_h;" ++
        "eval(trans(e));" ++
        "K_h := K_h | depsIn(k, e);" ++
     "pc := pc′"
```

Fig. 6. Inlining of $\texttt{eval}_p(e)$

function that computes $\kappa|_c$. Note that c here is the code that results from parsing the value denoted by e.

This approach has a downside. When the attacker has enough control over e, she can manipulate it in order to always generate different hashes. This affects the accumulation of dependencies (the cache of indirect flows will never be augmented across different runs) and hence the effectiveness of the monitor in identifying leaks. Since the monitor can leak during early runs, this may not be desirable. The following code exemplifies this situation:

```
tmp := 1; pub := 1;
eval_p(x ++ " if_q1 sec then tmp := 0;
             if_q2 tmp then pub := 0");
ret_q3 (pub)
```

The attacker may have control over `x`, affecting the hash and, therefore, avoid indirect dependencies from accumulating across different runs. Fig 7 represents a dependency chain of this code. The shaded box represents the `eval` context. Notice that q_1 and q_2 point to p because π had been extended with the latter. The edges a and b are created separately in two different runs, when `sec` is *1* or *0* respectively. The monitor should be able to capture the leak by accumulating both edges in κ, just like in the example in Fig. 1, because there is a path that connects q_3 with the high labeled q_1. But, since the attacker may manipulate the hash function output via the variable `x`, it is possible to avoid the accumulative effect in κ thus a and b will not exist simultaneously in any run.

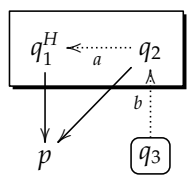

Fig. 7. Edges a and b are *both* needed to detect the leak in q_3

```
"eval_p(e)":
  "let pc′ = pc in:" ++
     "pc := pc | {p}" ++
     "k_p := k_p | dep(" ++ vars("e") ++ ") | pc′" ++
     "d_p := d_p | lev(" ++ vars("e") ++ ")" ++
     "let h = hash(e) in:" ++
        "k := k | K_h;" ++
        "eval(trans(e));" ++
        "d_p := d_p | secLevel(k,d,dom(depsIn(k,e));" ++
        "K_h := K_h | depsIn(k,e);" ++
     "pc := pc′"
```

Fig. 8. External anchor for $\texttt{eval}_p(e)$

One approach to this situation is to allow the program point p in the $\texttt{eval}_p(e)$ command to absorb all program points in the code denoted by e. Consequently, if a high node is created in the `eval` context, p will be raised to *high* just after the execution of `eval`. The reduction scheme Eval would have to be replaced by Eval′:

$$\frac{\begin{array}{c}\hat{\mu}(e) = \langle s, P, L\rangle \quad h = \mathsf{hash}(s) \quad \pi' = \pi \cup \{p\} \\ \delta' = \delta \uplus \{p \mapsto L\} \quad \kappa' = \kappa \uplus \mathcal{K}(h) \uplus \{p \mapsto \pi \cup P\} \\ \delta''' = \delta''[p \mapsto \mathsf{secLevel}^{\kappa'',\delta''}\, dom(\kappa''|_{\mathsf{parse}(s)})] \\ \langle \mathcal{K}, \kappa', \delta', \pi', \mu, \mathsf{parse}(s)\rangle \overset{n}{\rightarrowtail} \langle \mathcal{K}', \kappa'', \delta'', \pi', \mu'', stop\rangle\end{array}}{\langle \mathcal{K}, \kappa, \delta, \pi, \mu, \texttt{eval}_p(e)\rangle \rightarrowtail \langle \mathcal{K}'[h \mapsto \mathcal{K}'(h) \uplus \kappa''|_{\mathsf{parse}(s)}], \kappa'', \delta''', \pi, \mu'', stop\rangle}\ \textsc{Eval}'$$

Intuitively, every node associated to the program argument of `eval` passes on to p its level which hence works as an external anchor. In this way, if any node has the chance to be in the path of a leak, every low variable depending on them is considered dangerous. The new dependency chain for the above mentioned example is shown in Fig. 9, where the leak is detected. More precisely, when $\texttt{eval}_p(e)$ concludes, δ'' is upgraded to $\mathsf{secLevel}^{\kappa,\delta} dom(\kappa''|_c)$ (where *dom* is the domain of the mapping). Since $q1$ is assigned level secret by δ'', this bumps the level of p to secret. The proposed inlining is given in Fig. 8. In this approach the `ret` statement should not be allowed inside the `eval`, since the bumping of the security level of p is produced *a posteriori* to the execution of the argument of `eval`.

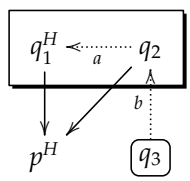

Fig. 9. Dependency chain with external anchor for $\texttt{eval}_p(e)$

5 Properties of the Inlining Transformation

This section addresses the *correctness* of the inlined transformation. We show that the inlined transformation of a command c simulates the execution of the monitor. First we define what it means for a $\mathcal{W}$-configuration to *simulate* a $\mathcal{W}^{deps}$-configuration. We write $\mathsf{trans}(c)$ for the result of applying the *recursive function* determined by the code for *trans* to the argument "c" and then parsing the result. Two sample clauses of trans are: $\mathsf{trans}(c_1; c_2) \stackrel{def}{=} \mathsf{trans}(c_1); \mathsf{trans}(c_2)$ for command composition and $\mathsf{trans}(\texttt{eval}(e)) \stackrel{def}{=}$ $\texttt{let}\ h = \ hash(e)\ \texttt{in}\ (k := k \,|\, K_h;\ \texttt{eval}(trans(e));\ K_h := K_h \,|\, depsIn(k, e))$ for $\texttt{eval}$. We also extend this definition with the clause: $\mathsf{trans}(stop) \stackrel{def}{=} stop$.

Definition 2. *A $\mathcal{W}$-configuration C simulates a $\mathcal{W}^{deps}$-configuration $\mathcal{D}$, written $\mathcal{D} \prec C$, iff*

1. $\mathcal{D} = \langle \mathcal{K}, \kappa, \delta, \pi, \mu, c \rangle$;
2. $C = \langle M, \mathsf{trans}(c) \rangle$;
3. $M(K) = \mathcal{K}$, $M(k) = \kappa$, $M(d) = \delta$, $M(pc) = \pi$; *and*
4. $\mu(x) = \langle M(x), M(x_P), M(x_L) \rangle$, *for all* $x \in dom(\mu)$.

In the expression '$M(K) = \mathcal{K}$' by abuse of notation we view $M(K)$ as a "dictionary" and therefore understand this expression as signifying that for all $h \in dom(\mathcal{K})$, $M(K_h) = \mathcal{K}(h)$. Similar comments apply to $M(k) = \kappa$ and $M(d) = \delta$. In the case of $M(pc) = \pi$, both sets of program points are tested for equality.

The following correctness property is proved by induction on an appropriate notion of *depth* of the reduction sequence $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$.

Proposition 2. *If (1) $\mathcal{D}_1 = \langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, c \rangle$; (2) $C_1 = \langle M_1, \mathsf{trans}(c) \rangle$; (3) $\mathcal{D}_1 \prec C_1$; and (4) $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$, $n \geq 0$; then there exists C_2 s.t. $C_1 \twoheadrightarrow C_2$ and $\mathcal{D}_2 \prec C_2$.*

$$\begin{array}{ccc} \mathcal{D}_1 & \prec & C_1 \\ {\scriptstyle n}\downarrow & & \vdots\downarrow{\scriptstyle *} \\ \mathcal{D}_2 & \prec & C_2 \end{array}$$

Remark 1. A converse result also holds: modulo the administrative commands inserted by trans, reduction from C_1 originates from corresponding commands in c. This may be formalised by requiring the inlining transformation to insert a form of labeled `skip` command to signal the correspondence of inlined commands with their original counterparts (cf. Thm.2(b) in [6]).

6 Conclusions and Future Work

We recast the dependency analysis monitor of Shroff et al. [19] to a simple imperative language and propose a transformation for inlining this monitor on-the-fly. The purpose is to explore the viability of a completely dynamic inlined dependency analysis as an alternative to other run-time approaches that either require additional information from the source code (such as branches not taken [6]) or resort to rather restrictive mechanisms such as *no sensitive upgrade* [4] (where execution gets stuck on attempting to assign a public variable in a secret context) or *permissive upgrade* [5] (where, although assignment of public variables in a secret contexts is not allowed, branching on expressions that depend on such variables is disallowed).

This paper reports work in progress, hence we mention some of the lines we are currently following. First we would like to gain some experience with a prototype implementation of the inlined transformation as a means of foreseeing issues related to usability and scaling. Second, we are considering the inclusion of an output command and an analysis of how the notion of progress-sensitivity [2] adapts to the dependency tracking setting. Finally, inlining declassification mechanisms will surely prove crucial for any practical tool based on IFA.

Acknowledgements: To the referees for supplying helpful feedback.

References

1. `http://www.cse.chalmers.se/~bello/inlining`.
2. Askarov, A., and Sabelfeld, A. Tight enforcement of information-release policies for dynamic languages. In *Computer Security Foundations Workshop* (2009), pp. 43–59.
3. Austin, T. H., and Flanagan, C. Efficient purely-dynamic information flow analysis. In *SIGPLAN Conference on Programming Language Design and Implementation* (2009), pp. 113–124.
4. Austin, T. H., and Flanagan, C. Efficient purely-dynamic information flow analysis. *SIGPLAN Not. 44* (December 2009), 20–31.
5. Austin, T. H., and Flanagan, C. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security* (New York, NY, USA, 2010), PLAS '10, ACM, pp. 3:1–3:12.
6. Chudnov, A., and Naumann, D. A. Information flow monitor inlining. In *Computer Security Foundations Workshop* (2010), pp. 200–214.
7. Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. Staged information flow for javascript. In *SIGPLAN Conference on Programming Language Design and Implementation* (2009), pp. 50–62.
8. Dhawan, M., and Ganapathy, V. Analyzing information flow in javascript-based browser extensions. In *Annual Comp. Sec. App. Conference* (2009), pp. 382–391.

9. ERLINGSSON, U. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Department of Computer Science, Cornell University, 2003. TR 2003-1916.
10. FUTORANSKY, A., GUTESMAN, E., AND WAISSBEIN, A. A dynamic technique for enhancing the security and privacy of web applications. Black Hat USA 2007 Briefings. Las Vegas, NV, USA, August 1-2 2007.
11. GUERNIC, G. L. Automaton-based confidentiality monitoring of concurrent programs. In *Computer Security Foundations Workshop* (2007), pp. 218–232.
12. GUERNIC, G. L., BANERJEE, A., JENSEN, T. P., AND SCHMIDT, D. A. Automata-based confidentiality monitoring. In *Asian Computing Science Conference* (2006), pp. 75–89.
13. HUNT, S., AND SANDS, D. On flow-sensitive security types. In *POPL* (2006), J. G. Morrisett and S. L. P. Jones, Eds., ACM, pp. 79–90.
14. MAGAZINIUS, J., RUSSO, R., AND SABELFELD, A. On-the-fly inlining of dynamic security monitors. In *In Proc. IFIP International Information Security Conference* (2010).
15. MCCAMANT, S., AND ERNST, M. D. Quantitative information flow as network flow capacity. In *SIGPLAN Conference on Programming Language Design and Implementation* (2008), pp. 193–205.
16. RUSSO, A., AND SABELFELD, A. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium* (Washington, DC, USA, 2010), CSF '10, IEEE Computer Society, pp. 186–199.
17. SABELFELD, A., AND MYERS, A. C. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*.
18. SABELFELD, A., AND RUSSO, A. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Ershov Memorial Conf.* (2009), pp. 352–365.
19. SHROFF, P., SMITH, S., AND THOBER, M. Dynamic dependency monitoring to secure information flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 203–217.
20. VENKATAKRISHNAN, V. N., XU, W., DUVARNEY, D. C., AND SEKAR, R. Provably correct runtime enforcement of non-interference properties. In *International Conference on Information and Communication Security* (2006), pp. 332–351.
21. VOLPANO, D. M., IRVINE, C. E., AND SMITH, G. A sound type system for secure flow analysis. *Journal of Computer Security 4*, 167–188.

Paper **two**

Appendix

1 A simple *while* language ($\mathcal{W}$)

This section presents $\mathcal{W}$, a simple imperative language in which the inlined transformation executes. The syntax of $\mathcal{W}$ is defined as follows:

$$
\begin{array}{rll}
P, pc & ::= \{\overline{p}\} & \text{(set of ppids, program counter)} \\
v & ::= i \,\big|\, s \,\big|\, P \,\big|\, L & \text{(value)} \\
e & ::= x \,\big|\, v \,\big|\, e \oplus e \,\big|\, f(e) \,\big|\, \texttt{case}\ e\ \texttt{of}\ (e : e)^{+} & \text{(expression)} \\
c & ::= \texttt{skip} \,\big|\, x := e \,\big|\, \texttt{let}\, x = e\, \texttt{in}\, c \,\big|\, c; c \,\big|\, \texttt{while}\, e\, \texttt{do}\, c \,\big| & \text{(command)} \\
 & \quad \big|\ \texttt{if}\, e\, \texttt{then}\, c\, \texttt{else}\, c \,\big|\, \texttt{ret}(e) \,\big|\, \texttt{eval}(e) \,\big|\, \mathit{stop} & \\
E & ::= \emptyset \,\big|\, f(x) \doteq e; E & \text{(expr. environment)} \\
M & ::= \{\overline{x \mapsto v}\} & \text{(memory)}
\end{array}
$$

In contrast to $\mathcal{W}^{deps}$, it operates on standard, unlabeled values. However, it is necessary to add sets of program points and security levels as values, since they will be manipulated by the inlined monitor.

Definition 1. Closed expression evaluation *is defined as follows,*

$$
\begin{aligned}
\mathcal{I}(i) &\stackrel{def}{=} i \\
\mathcal{I}(s) &\stackrel{def}{=} s \\
\mathcal{I}(f(e)) &\stackrel{def}{=} \hat{f}(\mathcal{I}(e)) \\
\mathcal{I}(\texttt{case}\ e\ \texttt{of}\ \boldsymbol{e : e'}) &\stackrel{def}{=} \mathit{câse}\ \mathcal{I}(e)\ \mathit{of}\ \boldsymbol{e'_i : e'} \\
\mathcal{I}(e_1 \oplus e_2) &\stackrel{def}{=} \mathcal{I}(e_1) \hat{\oplus} \mathcal{I}(e_2)
\end{aligned}
$$

where $\hat{f}(v) \stackrel{def}{=} \mathcal{I}(e[x := v])$*, if* $f(x) \doteq e \in E$ *and* $\mathit{câse}\ u\ \mathit{of}\ \boldsymbol{e : e'} = \mathcal{I}(\sigma(e'_i))$ *if* u *matches* e_i *with substitution* σ*. We leave it to the user to guarantee that user-defined functions are terminating.*

Given a memory M, the *variable replacement* function, also written M, applies to expressions: it traverses expressions replacing variables by their values.It is defined only if the free variables of its argument are in the domain of M. Finally, *open expression evaluation* is defined as $\mathcal{I} \circ M$ and abbreviated $\hat{M}$.

$$\frac{}{\langle M, \texttt{skip}\rangle \to \langle M, \mathit{stop}\rangle}\ \textsc{Skip}$$

$$\frac{\hat{M}(e) = v \quad y \text{ fresh}}{\langle M, \texttt{let}\, x = e\, \texttt{in}\, c\rangle \to \langle M[y \mapsto v], c[x := y]\rangle}\ \textsc{Let}$$

$$\frac{\langle M, c_1\rangle \longrightarrow \langle M', c_1'\rangle \quad c_1' \neq \mathit{stop}}{\langle M, c_1; c_2\rangle \to \langle M', c_1'; c_2\rangle}\ \textsc{Seq-L} \qquad \frac{\langle M, c_1\rangle \longrightarrow \langle M', \mathit{stop}\rangle}{\langle M, c_1; c_2\rangle \to \langle M', c_2\rangle}\ \textsc{Seq-R}$$

$$\frac{\hat{M}(e) = v}{\langle M, x\texttt{:=}e\rangle \to \langle M[x := v], \mathit{stop}\rangle}\ \textsc{Assign}$$

$$\frac{\hat{M}(e) \neq 0}{\langle M, \texttt{while}\, e\, \texttt{do}\, c\rangle \to \langle M, c; \texttt{while}\, e\, \texttt{do}\, c\rangle}\ \textsc{While-T}$$

$$\frac{\hat{M}(e) = 0}{\langle M, \texttt{while}\, e\, \texttt{do}\, c\rangle \to \langle M, \mathit{stop}\rangle}\ \textsc{While-F}$$

$$\frac{\hat{M}(e) \neq 0}{\langle M, \texttt{if}\, e\, \texttt{then}\, c_1\, \texttt{else}\, c_2\rangle \to \langle M, c_1\rangle}\ \textsc{If-T} \qquad \frac{\hat{M}(e) = 0}{\langle M, \texttt{if}\, e\, \texttt{then}\, c_1\, \texttt{else}\, c_2\rangle \to \langle M, c_2\rangle}\ \textsc{If-F}$$

$$\frac{\hat{M}(e) = v}{\langle M, \texttt{ret}(e)\rangle \to \langle M[\mathit{ret} \mapsto v], \mathit{stop}\rangle}\ \textsc{Ret} \qquad \frac{\hat{M}(e) = s \quad \mathsf{parse}(s) = c}{\langle M, \texttt{eval}(e)\rangle \to \langle M, c\rangle}\ \textsc{Eval}$$

Fig. 10. Semantics for $\mathcal{W}$

A $\mathcal{W}$*-(run-time) configuration* is an expression of the form $\langle E, M, c\rangle$ where E a user-defined expression environment, M is a memory and c is a command. Since E is never modified during execution, we drop it for the sake of readability and assume it is implicit. The small-step semantics of $\mathcal{W}$ commands is given in terms of a binary *reduction* relation over configurations witnessed by the judgement $\langle M, c\rangle \to \langle M', c'\rangle$. This judgement is defined by means of the *reduction schemes* of Fig. 10, which are standard.

Since the semantics of $\mathcal{W}^{deps}$ is defined in terms of a mixed-step relation whereas $\mathcal{W}$ uses a small-step one, the following simple result will be useful for establishing the correctness property of the inlined monitor.

Lemma 1. *1.* $\langle M_1, c_1\rangle \twoheadrightarrow \langle M_2, \mathit{stop}\rangle$ *and* $\langle M_2, c_2\rangle \twoheadrightarrow \langle M_3, \mathit{stop}\rangle$ *implies* $\langle M_1, c_1; c_2\rangle \twoheadrightarrow \langle M_3, \mathit{stop}\rangle$.

2. $\langle M_1, c\rangle \twoheadrightarrow \langle M_2, stop\rangle$ *and* $\hat{M}_1(e) \neq 0$ *implies* $\langle M_1, \texttt{while}\, e\, \texttt{do}\, c\rangle \twoheadrightarrow \langle M_2, \texttt{while}\, e\, \texttt{do}\, c\rangle$.

2 Correspondence between inlined monitor and the monitor

We write $\mathsf{trans}(c)$ for the result of applying the *recursive function* determined by the code for *trans* to the argument "c" and then parsing the result. It resorts to a similar recursive function vars associated to the code for *vars* only this function applies to expressions rather than strings that parse to expressions.

Definition 2. *The command translation function* trans*, is defined as follows:*

$$
\begin{aligned}
\mathsf{trans}(stop) &\stackrel{def}{=} stop \\
\mathsf{trans}(\texttt{skip}) &\stackrel{def}{=} \texttt{skip} \\
\mathsf{trans}(x := e) &\stackrel{def}{=} x_L := lev(\mathsf{vars}(e)); \\
&\quad\; x_P := dep(\mathsf{vars}(e)) \mid pc; \\
&\quad\; x := e; \\
\mathsf{trans}(\texttt{let}\, x = e\, \texttt{in}\, c) &\stackrel{def}{=} let\ x = e\ in \\
&\quad\; x_L := lev(\mathsf{vars}(e)); \\
&\quad\; x_P := dep(\mathsf{vars}(e)) \mid pc; \\
&\quad\; \mathsf{trans}(c) \\
\mathsf{trans}(c_1; c_2) &\stackrel{def}{=} \mathsf{trans}(c_1); \mathsf{trans}(c_2) \\
\mathsf{trans}(\texttt{while}_p\, e\, \texttt{do}\, c) &\stackrel{def}{=} k_p := k_p \mid dep(\mathsf{vars}(e)) \mid pc; \\
&\quad\; d_p := d_p \mid lev(\mathsf{vars}(e)); \\
&\quad\; while\ e\ do \\
&\quad\; (let\ pc' = pc\ in \\
&\quad\;\; pc := pc \mid \{p\}; \\
&\quad\;\; \mathsf{trans}(c) \\
&\quad\;\; pc := pc'; \\
&\quad\;\; k_p := k_p \mid dep(\mathsf{vars}(e)) \mid pc; \\
&\quad\;\; d_p := d_p \mid lev(\mathsf{vars}(e))); \\
\mathsf{trans}(\texttt{if}_p\, e\, \texttt{then}\, c_1\, \texttt{else}\, c_2) &\stackrel{def}{=} k_p := k_p \mid dep(\mathsf{vars}(e)) \mid pc; \\
&\quad\; d_p := d_p \mid lev(\mathsf{vars}(e)); \\
&\quad\; let\ pc' = pc\ in \\
&\quad\;\; pc := pc \mid \{p\}; \\
&\quad\;\; \texttt{if}\, e\, \texttt{then}\, \mathsf{trans}(c_1)\, \texttt{else}\, \mathsf{trans}(c_2); \\
&\quad\;\; pc := pc' \\
\mathsf{trans}(\texttt{eval}(e)) &\stackrel{def}{=} let\ h\ =\ hash(e)\ in \\
&\quad\; k := k \mid K_h; \\
&\quad\; \texttt{eval}(trans(e)); \\
&\quad\; K_h := K_h \mid depsIn(k, e)
\end{aligned}
$$

Note that the clause that defines trans for eval is not recursive.

Definition 3. *A $\mathcal{W}$-configuration C simulates a $\mathcal{W}^{deps}$-configuration $\mathcal{D}$, written $\mathcal{D} \prec C$, iff*

1. $\mathcal{D} = \langle \kappa, \delta, \pi, \mu, c \rangle$;
2. $C = \langle M, \mathsf{trans}(c) \rangle$;
3. $M(pc) = \pi$;
4. $M(k) = \kappa$;
5. $M(d) = \delta$; *and*
6. $\mu(x) = \langle M(x), M(x_P), M(x_L) \rangle$, *for all* $x \in dom(\mu)$.

Before addressing the main result (Prop. 2) an auxiliary result.

Lemma 2. *Suppose $\mathcal{D} = \langle \kappa, \delta, \pi, \mu, c \rangle$, e is an expression, $\widehat{\mu}(e) = \langle i, P, L \rangle$ and $\mathcal{D} \prec C$. Let $vars(e) = \boldsymbol{x}$. Then*

1. $dep(\boldsymbol{x}) = P$; *and*
2. $lev(\boldsymbol{x}) = L$.

Proof. By induction on e.

Definition 4. *A $\mathcal{W}$-configuration C simulates a $\mathcal{W}^{deps}$-configuration $\mathcal{D}$, written $\mathcal{D} \prec C$, iff*

1. $\mathcal{D} = \langle \mathcal{K}, \kappa, \delta, \pi, \mu, c \rangle$;
2. $C = \langle M, \mathsf{trans}(c) \rangle$;
3. $M(K) = \mathcal{K}$, $M(k) = \kappa$, $M(d) = \delta$, $M(pc) = \pi$; *and*
4. $\mu(x) = \langle M(x), M(x_P), M(x_L) \rangle$, *for all* $x \in dom(\mu)$.

Remark 1. $\mathcal{D} \prec C$, implies $\hat{\mu}(e) = \hat{M}(e)$ for all e s.t. $\mathsf{vars}(e) \subseteq dom(\mu)$.

Given $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_{n+1}$ consisting of a sequence of composable mixed-step judgements $\mathcal{D}_1 \rightarrowtail \mathcal{D}_2 \rightarrowtail \ldots \rightarrowtail \mathcal{D}_{n+1}$ and assume $\pi_1, \ldots, \pi_n$ are the derivations of each of these judgements (i.e. π_i is a derivation of the judgement $\mathcal{D}_i \rightarrowtail \mathcal{D}_{i+1}$, $i \in 1..n$). We define the **depth** of this sequence, written $\mathsf{depth}(\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_{n+1})$, as the maximum of the depths of the π_is. If $n = 0$ (i.e. the sequence is empty), then we declare the depth to be 0.

The following result is required for the proof of Prop. 2 (case of eval) and is proved by induction on the strings that may be correctly parsed as commands:

$$s ::= \text{"}\texttt{skip}\text{"} \,\Big|\, \text{"}x := e\text{"} \,\Big|\, \text{"}\texttt{eval}(e)\text{"} \,\Big|\, \text{"}let\ x = e\ in\ \text{"} +\!\!+ s \,\Big|\, s +\!\!+ \text{";"} +\!\!+ s \,\Big|\, \text{"}if\ e\ then\ \text{"} +\!\!+ s +\!\!+ \text{"}\ then\ \text{"} +\!\!+ s \,\Big|\, \text{"}while\ e\ do\ \text{"} +\!\!+ s$$

Lemma 3. $parse(\hat{trans}(s)) = trans(parse(s))$

Proof. – $s = \text{"}\texttt{skip}\text{"}$.

$$
\begin{aligned}
& \mathsf{parse}(\hat{trans}(\text{"}\texttt{skip}\text{"})) \\
= \; & \mathsf{parse}(\mathcal{I}(\texttt{case}\ \text{"}\texttt{skip}\text{"}\ \texttt{of}\ \ldots)) \\
= \; & \mathsf{parse}(\text{"}\texttt{skip}\text{"}) \\
= \; & \texttt{skip} \\
= \; & \mathsf{trans}(\texttt{skip}) \\
= \; & \mathsf{trans}(\mathsf{parse}(\text{"}\texttt{skip}\text{"}))
\end{aligned}
$$

– $s = \text{"}x := e\text{"}$. Let t denote the string

$$\text{"}x_L := lev(\boldsymbol{x}); x_P := dep(\boldsymbol{x})|pc; x := e\text{"}$$

where $\boldsymbol{x}$ is the string of comma separated variables "$x_1, \ldots, x_n$" returned by $\mathcal{I}(vars(\text{"}e\text{"}))$.

$$
\begin{aligned}
& \mathsf{parse}(\hat{trans}(\text{"}x := e\text{"})) \\
= \; & \mathsf{parse}(\mathcal{I}(\texttt{case}\ \text{"}x := e\text{"}\ \texttt{of}\ \ldots)) \\
= \; & \mathsf{parse}(t) \\
= \; & x_L := lev(\boldsymbol{x}); x_P := dep(\boldsymbol{x})|pc; x := e \\
= \; & x_L := lev(\mathsf{vars}(e)); x_P := dep(\mathsf{vars}(e))|pc; x := e \\
= \; & \mathsf{trans}(x := e) \\
= \; & \mathsf{trans}(\mathsf{parse}(\text{"}x := e\text{"}))
\end{aligned}
$$

– $s = \text{"}\texttt{eval}(e)\text{"}$. Let t denote the string

$$\text{"}let\ h\ =\ hash(e)\ in(k := k|K_h; \texttt{eval}(trans(e)); K_h := K_h|depsIn(k, e))\text{"}$$

$$
\begin{aligned}
& \mathsf{parse}(\hat{trans}(\text{"}\texttt{eval}(e)\text{"})) \\
= \; & \mathsf{parse}(\mathcal{I}(\texttt{case}\ \text{"}\texttt{eval}(e)\text{"}\ \texttt{of}\ \ldots)) \\
= \; & \mathsf{parse}(t) \\
= \; & let\ h\ =\ hash(e)\ in \\
& \quad k := k|K_h; \\
& \quad \texttt{eval}(trans(e)); \\
& \quad K_h := K_h|depsIn(k, e) \\
= \; & \mathsf{trans}(\texttt{eval}(e)) \\
= \; & \mathsf{trans}(\mathsf{parse}(\text{"}\texttt{eval}(e)\text{"}))
\end{aligned}
$$

– $s = \text{"}let\ x = e\ in\ \text{"} +\!\!+ s_1$. Let t be the string

$$\text{"}let\ x = e\ in\ x_L := lev(\boldsymbol{x}); x_P := dep(\boldsymbol{x})|pc;\text{"} +\!\!+ \mathcal{I}(trans(s_1))$$

where $\boldsymbol{x}$ is the string of comma separated variables "$x_1, \ldots, x_n$" returned by $\mathcal{I}(vars(\text{"}e\text{"}))$.

$$
\begin{aligned}
& \mathsf{parse}(\hat{trans}(\text{"}let\ x = e\ in\ \text{"} +\!\!+ s_1)) \\
= \; & \mathsf{parse}(\mathcal{I}(\texttt{case}\ \text{"}let\ x = e\ in\ \text{"} +\!\!+ s_1\ \texttt{of}\ \ldots)) \\
= \; & \mathsf{parse}(t) \\
= \; & let\ x = e\ in\ x_L := lev(\boldsymbol{x}); x_P := dep(\boldsymbol{x})|pc; \mathsf{parse}(\mathcal{I}(trans(s_1))) \\
= \; & let\ x = e\ in\ x_L := lev(\mathsf{vars}(e)); x_P := dep(\mathsf{vars}(e))|pc; \mathsf{parse}(\mathcal{I}(trans(s_1))) \\
= \; & let\ x = e\ in\ x_L := lev(\mathsf{vars}(e)); x_P := dep(\mathsf{vars}(e))|pc; \mathsf{parse}(\hat{trans}(\mathcal{I}(s_1))) \\
= \; & let\ x = e\ in\ x_L := lev(\mathsf{vars}(e)); x_P := dep(\mathsf{vars}(e))|pc; \mathsf{parse}(\hat{trans}(s_1)) \\
=_{IH} \; & let\ x = e\ in\ x_L := lev(\mathsf{vars}(e)); x_P := dep(\mathsf{vars}(e))|pc; \mathsf{trans}(\mathsf{parse}(s_1)) \\
= \; & \mathsf{trans}(\texttt{let}\ x = e\ \texttt{in}\ \mathsf{parse}(s_1)) \\
= \; & \mathsf{trans}(\mathsf{parse}(\text{"}\texttt{let}\ x = e\ \texttt{in}\ \text{"} +\!\!+ s_1))
\end{aligned}
$$

– $s = s_1 \mathbin{+\!\!+} ";" \mathbin{+\!\!+} s_2$.

$$
\begin{aligned}
& \mathsf{parse}(\hat{trans}(s_1 \mathbin{+\!\!+} ";" \mathbin{+\!\!+} s_2)) \\
= \; & \mathsf{parse}(\mathcal{I}(\texttt{case}\ (s_1 \mathbin{+\!\!+} ";" \mathbin{+\!\!+} s_2)\ \texttt{of}\ \ldots)) \\
= \; & \mathsf{parse}(\mathcal{I}(trans(s_1)) \mathbin{+\!\!+} ";" \mathbin{+\!\!+} \mathcal{I}(trans(s_2))) \\
= \; & \mathsf{parse}(\mathcal{I}(trans(s_1))); \mathsf{parse}(\mathcal{I}(trans(s_2))) \\
= \; & \mathsf{parse}(\hat{trans}(\mathcal{I}(s_1))); \mathsf{parse}(\hat{trans}(\mathcal{I}(s_2))) \\
= \; & \mathsf{parse}(\hat{trans}(s_1)); \mathsf{parse}(\hat{trans}(s_2)) \\
=_{IH} \; & \mathsf{trans}(\mathsf{parse}(s_1)); \mathsf{trans}(\mathsf{parse}(s_1)) \\
= \; & \mathsf{trans}(\mathsf{parse}(s_1 \mathbin{+\!\!+} ";" \mathbin{+\!\!+} s_1))
\end{aligned}
$$

– $s = "if\ e\ then\ " \mathbin{+\!\!+} s_1 \mathbin{+\!\!+} "\ then\ " \mathbin{+\!\!+} s_2$. Let t be the string

$$
\begin{aligned}
& "k_p := k_p|dep(\boldsymbol{x})|pc;" \mathbin{+\!\!+} \\
& "d_p := d_p|lev(\boldsymbol{x});" \mathbin{+\!\!+} \\
& "let pc' = pc in" \mathbin{+\!\!+} \\
& \quad "pc := pc|\{p\};" \mathbin{+\!\!+} \\
& \quad "\texttt{if}\ e\ \texttt{then}\ " \mathbin{+\!\!+} trans(s_1) \mathbin{+\!\!+} "\ \texttt{else}\ " \mathbin{+\!\!+} trans(s_2); \mathbin{+\!\!+} \\
& \quad "pc := pc'"
\end{aligned}
$$

where $\boldsymbol{x}$ is the string of comma separated variables "$x_1, \ldots, x_n$" returned by $\mathcal{I}(vars("e"))$.

$$
\begin{aligned}
& \mathsf{parse}(\hat{trans}("if\ e\ then\ " \mathbin{+\!\!+} s_1 \mathbin{+\!\!+} "\ then\ " \mathbin{+\!\!+} s_2)) \\
= \; & \mathsf{parse}(\mathcal{I}(\texttt{case}\ "if\ e\ then\ " \mathbin{+\!\!+} s_1 \mathbin{+\!\!+} "\ then\ " \mathbin{+\!\!+} s_2\ \texttt{of}\ \ldots)) \\
= \; & \mathsf{parse}(t) \\
= \; & k_p := k_p|dep(\boldsymbol{x})|pc; d_p := d_p|lev(\boldsymbol{x}); \\
& let\ pc' = pc\ in \\
& \quad pc := pc|\{p\}; \\
& \quad \texttt{if}\ e\ \texttt{then}\ \mathsf{parse}(\mathcal{I}(trans(s_1)))\ \texttt{else}\ \mathsf{parse}(\mathcal{I}(trans(s_1))); \\
& \quad pc := pc' \\
= \; & k_p := k_p|dep(\boldsymbol{x})|pc; d_p := d_p|lev(\boldsymbol{x}); \\
& let\ pc' = pc\ in \\
& \quad pc := pc|\{p\}; \\
& \quad \texttt{if}\ e\ \texttt{then}\ \mathsf{parse}(\hat{trans}(\mathcal{I}(s_1)))\ \texttt{else}\ \mathsf{parse}(\hat{trans}(\mathcal{I}(s_1))); \\
& \quad pc := pc' \\
= \; & k_p := k_p|dep(\boldsymbol{x})|pc; d_p := d_p|lev(\boldsymbol{x}); \\
& let\ pc' = pc\ in \\
& \quad pc := pc|\{p\}; \\
& \quad \texttt{if}\ e\ \texttt{then}\ \mathsf{parse}(\hat{trans}(s_1))\ \texttt{else}\ \mathsf{parse}(\hat{trans}(s_2)); \\
& \quad pc := pc' \\
=_{IH} \; & k_p := k_p|dep(\boldsymbol{x})|pc; d_p := d_p|lev(\boldsymbol{x}); \\
& let\ pc' = pc\ in \\
& \quad pc := pc|\{p\}; \\
& \quad \texttt{if}\ e\ \texttt{then}\ \mathsf{trans}(\mathsf{parse}(s_1))\ \texttt{else}\ \mathsf{trans}(\mathsf{parse}(s_2)); \\
& \quad pc := pc' \\
= \; & \mathsf{trans}(\texttt{if}\ e\ \texttt{then}\ \mathsf{parse}(s_1)\ \texttt{else}\ \mathsf{parse}(s_2)) \\
= \; & \mathsf{trans}(\mathsf{parse}("\texttt{if}\ e\ \texttt{then}\ " \mathbin{+\!\!+} s_1 \mathbin{+\!\!+} "\ \texttt{else}\ " \mathbin{+\!\!+} s_2))
\end{aligned}
$$

– $s = \text{"}\mathit{while\ e\ do}\text{"} +\!\!+ s_1$. Similar to previous item.

Proposition 1. *If*

– $\mathcal{D}_1 = \langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, c \rangle$;
– $C_1 = \langle M_1, \mathsf{trans}(c) \rangle$;
– $\mathcal{D}_1 \prec C_1$;
– $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$, $n \geq 0$;

then there exists C_2 *s.t.* $C_1 \twoheadrightarrow C_2$ *and* $\mathcal{D}_2 \prec C_2$.

$$\begin{array}{ccc} \mathcal{D}_1 & \prec & C_1 \\ {\scriptstyle n}\downarrow & & \vdots\!\!\downarrow \\ \mathcal{D}_2 & \prec & C_2 \end{array}$$

Proof. By induction on (d, n) ordered by the standard lexicographic ordering, where d is the depth of the sequence $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$. We perform case analysis on c.

– $c = \texttt{skip}$. In this case $\mathcal{D}_1 = \langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, \texttt{skip} \rangle$ and $C_1 = \langle M_1, \texttt{skip} \rangle$. By definition of the reduction relation over $\mathcal{W}^{deps}$, it must be the case that $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$ is of the form $\mathcal{D}_1 \rightarrowtail \mathcal{D}_2$ (i.e. $n = 1$) with the derivation of this judgement ending in an application of Skip:

$$\frac{}{\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, \texttt{skip} \rangle \rightarrowtail \langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, \mathit{stop} \rangle}\ \textsc{Skip}$$

For this base case ($(d, n) = (1, 1)$), it suffices to take $C_2 = \langle M_1, \mathit{stop} \rangle$.

– $c = \texttt{let}\, x = e\, \texttt{in}\, c_1$. In this case $\mathcal{D}_1 = \langle \kappa_1, \delta_1, \pi_1, \mu_1, \texttt{let}\, x = e\, \texttt{in}\, c_1 \rangle$ and C_1 is

$$\langle M_1, \texttt{let}\, x = e\, \texttt{in}\, x_L := \mathit{lev}(\boldsymbol{x}); x_P := \mathit{dep}(\boldsymbol{x}); \mathsf{trans}(c_1) \rangle$$

where $\boldsymbol{x}$ is a string of comma separated variables returned by $\mathsf{vars}(e)$. By definition of the reduction relation over $\mathcal{W}^{deps}$, it must be the case that $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$ is of the form $\mathcal{D}_1 \rightarrowtail \mathcal{D}_2$ (i.e. $n = 1$) with the derivation of this judgement ending in an application of Let:

$$\frac{\begin{array}{c}\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1[y \mapsto \hat{\mu}(e)], c_1[x := y] \rangle \overset{n_1}{\rightarrowtail} \\ \langle \mathcal{K}_2, \kappa_2, \delta_2, \pi_1, \mu_2[y \mapsto v], \mathit{stop} \rangle\end{array}}{\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, \texttt{let}\, x = e\, \texttt{in}\, c_1 \rangle \rightarrowtail \langle \mathcal{K}_2, \kappa_2, \delta_2, \pi_1, \mu_2 \setminus y, \mathit{stop} \rangle}\ \textsc{Let}$$

where y is some fresh variable. Thus $(d, n) = (d, 1)$. Note that:

$$\begin{array}{ll} & \langle M_1, \texttt{let}\, x = e\, \texttt{in}\, x_L := \mathit{lev}(\boldsymbol{x}); x_P := \mathit{dep}(\boldsymbol{x}); \mathsf{trans}(c_1) \rangle \\ \rightarrow & \langle M_1[z \mapsto \hat{M}_1(e)], z_L := \mathit{lev}(\boldsymbol{x}); z_P := \mathit{dep}(\boldsymbol{x}); \mathsf{trans}(c_1)[x := z] \rangle \\ \rightarrow\rightarrow & \langle M_2, \mathsf{trans}(c_1)[x := z] \rangle \end{array}$$

where $M_2 = M_1[z \mapsto \hat{M}_1(e)][z_L \mapsto lev(\boldsymbol{x})][z_P \mapsto dep(\boldsymbol{x})|M_1(pc)]$. Define

- $\mathcal{D}_3 \stackrel{def}{=} \langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1[z \mapsto \hat{\mu}(e)], c_1[x := z] \rangle$, where we have renamed y to z which is assumed fresh;
- $C_3 \stackrel{def}{=} \langle M_2, \mathsf{trans}(c_1)[x := z] \rangle = \langle M_2, \mathsf{trans}(c_1[x := z]) \rangle$

and note that

- $\mathcal{D}_3 \stackrel{n_1}{\rightarrowtail} \langle \mathcal{K}_2 \kappa_2, \delta_2, \pi_1, \mu_2[z \mapsto v], stop \rangle$ follows from $\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1[y \mapsto \hat{\mu}(e)], c_1[x := y] \rangle \stackrel{n_1}{\rightarrowtail} \langle \mathcal{K}_2, \kappa_2, \delta_2, \pi_1, \mu_2[y \mapsto v], stop \rangle$;
- $\mathsf{depth}(\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1[y \mapsto \hat{\mu}(e)], c_1[x := y] \rangle \stackrel{n_1}{\rightarrowtail} \langle \mathcal{K}_2, \kappa_2, \delta_2, \pi_1, \mu_2[y \mapsto v], stop \rangle) = (d-1, n_1) < (d, 1) = \mathsf{depth}(\mathcal{D}_1 \rightarrowtail \mathcal{D}_2)$.

Thus we resort to the IH and deduce the existence of C_4 s.t. $\langle \mathcal{K}_2, \kappa_2, \delta_2, \pi_1, \mu_2[z \mapsto v], stop \rangle \prec C_4$ and $C_3 \twoheadrightarrow C_4$. Note that $C_4 = \langle M_4, stop \rangle$ for some memory M_4. Therefore, we set the desired $\mathcal{W}$-configuration to be $\langle M_4 \setminus z, stop \rangle$.

– $c = c_1; c_2$. In this case $\mathcal{D}_1 = \langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, c_1; c_2 \rangle$ and $C_1 = \langle M_1, \mathsf{trans}(c_1); \mathsf{trans}(c_2) \rangle$. By definition of the reduction relation over $\mathcal{W}^{deps}$, it must be the case that $\mathcal{D}_1 \stackrel{n}{\rightarrowtail} \mathcal{D}_2$ is of the form $\mathcal{D}_1 \rightarrowtail \mathcal{D}_3 \stackrel{n-1}{\rightarrowtail} \mathcal{D}_2$ with the derivation of the judgement $\mathcal{D}_1 \rightarrowtail \mathcal{D}_3$ ending in an application of SEQ:

$$\frac{\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, c_1 \rangle \stackrel{n_1}{\rightarrowtail} \langle \mathcal{K}_3, \kappa_3, \delta_3, \pi_1, \mu_3, stop \rangle}{\mathcal{D}_1 = \langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, c_1; c_2 \rangle \rightarrowtail \langle \mathcal{K}_3, \kappa_3, \delta_3, \pi_1, \mu_3, c_2 \rangle = \mathcal{D}_3} \text{SEQ}$$

Since

- $\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, c_1 \rangle \prec \langle M_1, \mathsf{trans}(c_1) \rangle$ follows from $\mathcal{D}_1 \prec C_1$;
- $\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, c_1 \rangle \stackrel{n_1}{\rightarrowtail} \langle \mathcal{K}_3, \kappa_3, \delta_3, \pi_1, \mu_3, stop \rangle$; and
- $\mathsf{depth}(\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, c_1 \rangle \stackrel{n_1}{\rightarrowtail} \langle \mathcal{K}_3, \kappa_3, \delta_3, \pi_1, \mu_3, stop \rangle) = (d_1, n_1)$ for some d_1 and $(d_1, n_1) < (d, n) = \mathsf{depth}(\mathcal{D}_1 \stackrel{n}{\rightarrowtail} \mathcal{D}_2)$, since $d_1 + 1$ is used to compute d.

We resort to the IH and assert the existence of C_3 s.t.

$$\langle M_1, \mathsf{trans}(c_1) \rangle \twoheadrightarrow C_3 \tag{1}$$

and

$$\langle \mathcal{K}_3, \kappa_3, \delta_3, \pi_1, \mu_3, stop \rangle \prec C_3 \tag{2}$$

From (2) and $trans(stop) = stop$, $C_3 = \langle M_3, stop \rangle$ for some memory M_3. Therefore,

$$\langle \mathcal{K}_3, \kappa_3, \delta_3, \pi_1, \mu_3, c_2 \rangle \prec \langle M_3, \mathsf{trans}(c_2) \rangle \tag{3}$$

Also,

- $\mathcal{D}_3 = \langle \mathcal{K}_3, \kappa_3, \delta_3, \pi_1, \mu_3, c_2 \rangle \stackrel{n-1}{\rightarrowtail} \langle \mathcal{K}_2, \kappa_2, \delta_2, \pi_1, \mu_2, stop \rangle$; and
- $\mathsf{depth}(\mathcal{D}_3 \stackrel{n-1}{\rightarrowtail} \langle \mathcal{K}_2, \kappa_2, \delta_2, \pi_1, \mu_2, stop \rangle) = (d_2, n-1)$ for some d_2. Since d_2 is used to compute d (hence $d_2 \leq d$), $(d_2, n-1) < (d, n)$.

Once again we resort to the IH to assert the existence of C_2 s.t.

$$\langle M_2, \mathsf{trans}(c_2)\rangle \twoheadrightarrow C_2 \tag{4}$$

and

$$\langle \mathcal{K}_2, \kappa_2, \delta_2, \pi_1, \mu_2, stop\rangle \prec C_2 \tag{5}$$

We conclude from (1) and (4) and Lem. 1(1).

– $c = x := e$. In this case $\mathcal{D}_1 = \langle \kappa_1, \delta_1, \pi_1, \mu_1, x := e\rangle$ and $C_1 = \langle M_1, \mathsf{trans}(x := e)\rangle$. Here $\mathsf{trans}(x := e)$ will be abbreviated c'. Note that c' is the code

$$x_L := lev(\boldsymbol{x}); x_P := dep(\boldsymbol{x})|pc; x := e$$

where $\mathsf{vars}(e) = \boldsymbol{x}$. By definition of the reduction relation over $\mathcal{W}^{deps}$, it must be the case that $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$ is of the form $\mathcal{D}_1 \rightarrowtail \mathcal{D}_2$ (i.e. $n = 1$). Moreover, the derivation of this judgement ends in an application of Assign:

$$\frac{\hat{\mu_1}(e) = \langle v, P, L\rangle}{\begin{array}{c}\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, x := e\rangle \rightarrowtail \\ \langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_2 = \mu_1[x \mapsto \langle v, P \cup \pi, L\rangle], stop\rangle = \mathcal{D}_2\end{array}}\ \textsc{Assign}$$

and $(d, n) = (d, 1)$. Note that

$$\begin{array}{ll} & \langle M_1, \mathsf{trans}(x := e)\rangle \\ = & \langle M_1, x_L := ; x_P := dep(\boldsymbol{x})|pc; x := e\rangle \\ \rightarrow\rightarrow & \langle M_1[x_L \mapsto lev(\boldsymbol{x})][x_P \mapsto dep(\boldsymbol{x})|M_1(pc)], x := e\rangle \\ \rightarrow & \langle M_1[x_L \mapsto lev(\boldsymbol{x})][x_P \mapsto dep(\boldsymbol{x})|M_1(pc)][x \mapsto \hat{M_1}(e)], stop\rangle \end{array}$$

Let M_2 stand for $M_1[x_L \mapsto lev(\boldsymbol{x})][x_P \mapsto dep(\boldsymbol{x})|M_1(pc)][x \mapsto \hat{M_1}(e)]$. We are left to verify that $\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_2, stop\rangle = \mathcal{D}_2 \prec \langle M_2, stop\rangle$. This implies verifying the following items:

1. $M_2(pc) = \pi_1$. This follows from $\mathcal{D}_1 \prec C_1$ and $M_2(pc) = M_1(pc)$.
2. $M_2(k) = \kappa_1$ and $M_2(d) = \delta_1$. Both follow from $\mathcal{D}_1 \prec C_1$ and $M_2(k) = M_1(k)$ and $M_2(d) = M_1(d)$.
3. $\mu_2(x) = \langle M_2(x), M_2(x_P), M_2(x_L)\rangle$, for all $x \in dom(\mu_2)$. For $M_2(x) = v$ this follows from Rem. 1. For $M_2(x_P) = P$ we resort to Lem. 2. Finally, for $M_2(x_L) = L$ we resort to Lem. 2.

– $c = \mathtt{while}_p\, e\, \mathtt{do}\, c_1$. In this case $\mathcal{D}_1 = \langle \kappa_1, \delta_1, \pi_1, \mu_1, \mathtt{while}_p\, e\, \mathtt{do}\, c_1\rangle$ and $C_1 = \langle M_1, \mathsf{trans}(\mathtt{while}_p\, e\, \mathtt{do}\, c_1)\rangle$. By definition of the reduction relation over $\mathcal{W}^{deps}$, two cases are possible.

1. Either $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$ is of the form $\mathcal{D}_1 \rightarrowtail \mathcal{D}_2$ (i.e. $n = 1$) with the derivation of the judgement $\mathcal{D}_1 \rightarrowtail \mathcal{D}_2$ ending in an application of While-F:

$$\frac{\begin{array}{c}\hat{\mu_1}(e) = \langle 0, P, L\rangle \\ \kappa_2 = \kappa_1 \uplus \{p \mapsto \pi_1 \cup P\} \quad \delta_2 = \delta_1 \uplus \{p \mapsto L\}\end{array}}{\begin{array}{c}\mathcal{D}_1 = \langle \kappa_1, \delta_1, \pi_1, \mu_1, \texttt{while}_p\, e\, \texttt{do}\, c_1\rangle \rightarrowtail \\ \langle \kappa_2, \delta_2, \pi_1, \mu_1, stop\rangle = \mathcal{D}_2\end{array}} \text{WHILE-F}$$

2. Or $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$ is of the form $\mathcal{D}_1 \rightarrowtail \mathcal{D}_3 \overset{n-1}{\rightarrowtail} \mathcal{D}_2$ with the derivation of the judgement $\mathcal{D}_1 \rightarrowtail \mathcal{D}_3$ ending in an application of WHILE-T:

$$\frac{\begin{array}{c}\hat{\mu_1}(e) = \langle i, P, L\rangle \quad i \neq 0 \quad \pi_1' = \pi_1 \cup P \\ \kappa_1' = \kappa_1 \uplus \{p \mapsto \pi_1 \cup P\} \quad \delta_1' = \delta_1 \uplus \{p \mapsto L\} \\ \langle \kappa_1', \delta_1', \pi_1', \mu_1, c_1\rangle \overset{n_1}{\rightarrowtail} \langle \kappa_3, \delta_3, \pi_1', \mu_3, stop\rangle\end{array}}{\begin{array}{c}\mathcal{D}_1 = \langle \kappa_1, \delta_1, \pi_1, \mu_1, \texttt{while}_p\, e\, \texttt{do}\, c_1\rangle \rightarrowtail \\ \langle \kappa_3, \delta_3, \pi_1, \mu_3, \texttt{while}_p\, e\, \texttt{do}\, c_1\rangle = \mathcal{D}_3\end{array}} \text{WHILE-T}$$

We consider each case in turn. First, recall that $\mathsf{trans}(c) =$

```
k[p] := k[p]|dep(x)|pc;
d[p] := d[p]|lev(x);
while e do
 (let pc' = pc in
  pc := pc|{p};
  trans(c1);
  pc := pc';
  k[p] := k[p]|dep(x)|pc;
  d[p] := d[p]|lev(x);
```

We address the first case. Note that $C_1 \twoheadrightarrow C_3 = \langle M_3, \texttt{while}\, e\, \texttt{do}\, c_3\rangle$, where c_3 is the above command without the first two lines. Since for all $x \in vars(e)$, $M_1(x) = M_3(x)$ and $\mathcal{D}_1 \prec C_1$, we deduce that $\hat{M}_3(e) = 0$. Thus reduction in $\mathcal{W}$ continues as follows:

$$C_3 = \langle M_3, \texttt{while}\, e\, \texttt{do}\, c_3\rangle \rightarrow \langle M_3, stop\rangle = C_4$$

We are left to verify that $\mathcal{D}_2 \prec C_4$:

1. $stop = \mathsf{trans}(stop)$. This holds by definition of trans.
2. $M_3(pc) = \pi_1$. This follows from $M_3(pc) = M_1(pc) =_{\mathcal{D}_1 \prec C_1} \pi_1$.
3. $M_3(k) = \kappa_1 \uplus \{p \mapsto \pi_1 \cup P\}$ (the case $M_3(d) = \delta_1 \uplus \{p \mapsto L\}$ is similar to this one and hence omitted). Note $M_3(k) = M_2(k) = \kappa_3$ where κ_3 is defined as follows:
$$\kappa_3(q) \overset{def}{=} \begin{cases} M_1(q), & \text{if } q \neq p \\ M_1(k_p) \cup M_1(pc) \cup dep(\boldsymbol{x}), & \text{if } q = p \end{cases}$$
Note that $M_1(pc) = \pi_1$ follows from $\mathcal{D}_1 \prec C_1$. Also from $\mathcal{D}_1 \prec C_1$ and Lem. 2, we obtain $dep(\boldsymbol{x}) = P$. Therefore $\kappa_3 = \kappa_1 \uplus \{p \mapsto \pi_1 \cup P\}$.
4. $\mu_1(x) = \langle M_3(x), M_3(x_P), M_3(x_L)\rangle$, for all $x \in dom(\mu_1)$. Since for all $x \in dom(\mu_1)$, $M_1(x) = M_3(x)$, $M_1(x_P) = M_3(x_P)$ and $M_1(x_L) = M_3(x_L)$ and $\mathcal{D}_1 \prec C_1$, we deduce $\mu_1(x) = \langle M_3(x), M_3(x_P), M_3(x_L)\rangle$, for all $x \in dom(\mu_1)$.

We now address the second case. Note that:

$$
\begin{array}{ll}
& C_1 \\
\twoheadrightarrow & C_3 = \langle M_3, \texttt{while}\, e\, \texttt{do}\, c_3 \rangle \qquad (\star_1) \\
\rightarrow & \langle M_3, c_3; \texttt{while}\, e\, \texttt{do}\, c_3 \rangle \\
= & \langle M_3, \\
& \quad (\texttt{let}\, pc' = pc\, \texttt{in} \\
& \quad\; pc := pc|\{p\}; \\
& \quad\; \mathsf{trans}(c_1); \\
& \quad\; pc := pc'; \\
& \quad\; k[p] := k[p]|dep(\boldsymbol{x})|pc; \\
& \quad\; d[p] := d[p]|lev(\boldsymbol{x}); \\
& \quad \texttt{while}\, e\, \texttt{do}\, c_3 \rangle \\
\rightarrow & \langle M_3[pc' \mapsto M_3(pc)], \\
& \quad\; pc := pc|\{p\}; \\
& \quad\; \mathsf{trans}(c_1); \\
& \quad\; pc := pc'; \\
& \quad\; k[p] := k[p]|dep(\boldsymbol{x})|pc; \\
& \quad\; d[p] := d[p]|lev(\boldsymbol{x}); \\
& \quad \texttt{while}\, e\, \texttt{do}\, c_3 \rangle \\
\rightarrow & \langle M_3[pc' \mapsto M_3(pc)][pc \mapsto M_3(pc)|\{p\}], \\
& \quad\; \mathsf{trans}(c_1); \\
& \quad\; pc := pc'; \\
& \quad\; k[p] := k[p]|dep(\boldsymbol{x})|pc; \\
& \quad\; d[p] := d[p]|lev(\boldsymbol{x}); \\
& \quad \texttt{while}\, e\, \texttt{do}\, c_3 \rangle \\
\twoheadrightarrow & \langle M_4, \qquad (\star_2) \\
& \quad\; pc := pc'; \\
& \quad\; k[p] := k[p]|dep(\boldsymbol{x})|pc; \\
& \quad\; d[p] := d[p]|lev(\boldsymbol{x}); \\
& \quad \texttt{while}\, e\, \texttt{do}\, c_3 \rangle \\
\twoheadrightarrow & \langle M_4[pc \mapsto M_3(pc)], \\
& \quad\; k[p] := k[p]|dep(\boldsymbol{x})|pc; \\
& \quad\; d[p] := d[p]|lev(\boldsymbol{x}); \\
& \quad \texttt{while}\, e\, \texttt{do}\, c_3 \rangle \\
= & \langle M_4[pc \mapsto M_3(pc)], \mathsf{trans}(\texttt{while}\, e\, \texttt{do}\, c_1) \rangle \; (\star_3)
\end{array}
$$

where

- $(\star_1)$: $\hat{M}_3(e) = \hat{M}_1(e) = 1$ holds since for all $x \in vars(e)$, $M_3(x) = M_1(x)$ and $\mathcal{D}_1 \prec C_1$:
- $(\star_2)$. For this step we resort to the IH given that $\langle \kappa'_1, \delta'_1, \pi'_1, \mu_1, c_1 \rangle \overset{n_1}{\rightarrowtail} \langle \kappa_3, \delta_3, \pi', \mu_3, stop \rangle$ and $\langle \kappa'_1, \delta'_1, \pi'_1, \mu_1, c_1 \rangle \prec \langle M_3[pc' \mapsto M_3(pc)][pc \mapsto M_3(pc)|\{p\}], \mathsf{trans}(c_1) \rangle$. Thus there exists that there exists M_4 s.t. $\langle M_3[pc' \mapsto M_3(pc)][pc \mapsto M_3(pc)|\{p\}], \mathsf{trans}(c_1) \rangle \twoheadrightarrow \langle M_4, stop \rangle$. From this we obtain the required computation:

$$
\langle M_3[pc \mapsto M_3(pc)|\{p\}],\ \begin{array}{l} \mathsf{trans}(c_1); \\ pc := pc'; \\ k[p] := k[p]|dep(\boldsymbol{x})|pc; \\ d[p] := d[p]|lev(\boldsymbol{x}); \\ \texttt{while}\, e\, \texttt{do}\, c_3 \rangle \end{array} \twoheadrightarrow \langle M_4,\ \begin{array}{l} pc := pc'; \\ k[p] := k[p]|dep(\boldsymbol{x})|pc; \\ d[p] := d[p]|lev(\boldsymbol{x}); \\ \texttt{while}\, e\, \texttt{do}\, c_3 \rangle \end{array}
$$

- ($\star_3$): follows from Def. 3.

Let $\mathcal{C}_5 \stackrel{def}{=} \langle M_4[pc \mapsto M_3(pc)], \mathsf{trans}(\texttt{while}\, e\, \texttt{do}\, c_1)\rangle$. We now verify that $\mathcal{D}_3 \prec \mathcal{C}_5$. In other words, that

$$\mathcal{D}_3 = \langle \kappa_3, \delta_3, \pi_1, \mu_3, \texttt{while}_p\, e\, \texttt{do}\, c_1\rangle \prec \langle M_4[pc \mapsto M_3(pc)], \mathsf{trans}(\texttt{while}\, e\, \texttt{do}\, c_1)\rangle$$

This implies verifying,

1. $M_4[pc \mapsto M_3(pc)](pc) = \pi_1$. $M_4[pc \mapsto M_3(pc)](pc) = M_3(pc) = M_1(pc) = \pi_1$. The last equality follows from $\mathcal{D}_1 \prec \mathcal{C}_1$.
2. $M_4[pc \mapsto M_3(pc)](k) = \kappa_3$.
3. $M_4[pc \mapsto M_3(pc)](d) = \delta_3$.
4. $\mu_3(x) = \langle M_4[pc \mapsto M_3(pc)](x), M_4[pc \mapsto M_3(pc)](x_P), M_4[pc \mapsto M_3(pc)](x_L)\rangle$, for all $x \in dom(\mu_3)$.

Finally, note that $\mathcal{D}_3 \prec \mathcal{C}_5$; $\mathcal{D}_3 \overset{n-1}{\rightarrowtail} \mathcal{D}_2$; and $(\texttt{while}_p\, e\, \texttt{do}\, c_1, n-1) < (\texttt{while}_p\, e\, \texttt{do}\, c_1, n)$. Thus we resort to the IH and conclude the case.

- $c = \texttt{if}_p\, e\, \texttt{then}\, c_1\, \texttt{else}\, c_2$. Similar to the case for while.
- $c = \texttt{eval}(e)$. In this case $\mathcal{D}_1 = \langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, \texttt{eval}(e)\rangle$ and $\mathcal{C}_1 = \langle M_1, \mathsf{trans}(\texttt{eval}(e))\rangle$. Here $\mathsf{trans}(\texttt{eval}(\mathrm{e}))$, abbreviated c', is the following code:

```
let pc' = pc in
  pc := pc | {p};
  k_p := k_p | dep(vars("e")) | pc';
  d_p := d_p | lev(vars("e"));
  let h = hash(e) in  k := k | K_h;  eval(trans(e));  K_h := K_h | depsIn(k, e);
  pc := pc'
```

By definition of the reduction relation over $\mathcal{W}^{deps}$, it must be the case that $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$ is of the form $\mathcal{D}_1 \rightarrowtail \mathcal{D}_2$ (i.e. $n = 1$). Moreover, the derivation of this judgement ends in an application of EVAL:

$$\frac{\begin{array}{c}\hat{\mu_1}(e) = \langle s, P, L\rangle \quad h = \mathsf{hash}(s) \quad \pi_1' = \pi_1 \uplus \{p\} \\ \kappa_1' = \kappa_1 \uplus \mathcal{K}_1(h) \uplus \{p \mapsto \pi_1 \cup P\} \quad \delta_1' = \delta_1 \uplus \{p \mapsto L\} \\ \langle \mathcal{K}_1, \kappa_1', \delta_1', \pi_1', \mu_1, \mathsf{parse}(s)\rangle \overset{m}{\rightarrowtail} \langle \mathcal{K}_1', \kappa_2, \delta_2, \pi_1', \mu_2, stop\rangle\end{array}}{\begin{array}{c}\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, \texttt{eval}_p(e)\rangle \rightarrowtail \\ \langle \mathcal{K}_1'[h \mapsto \mathcal{K}_1'(h) \uplus \kappa_2|_{\mathsf{parse}(s)}], \kappa_2, \delta_2, \pi_1, \mu_2, stop\rangle\end{array}}\ \textsc{Eval}$$

and $(d, n) = (d_1 + 1, 1)$, where d_1 is the depth of $\langle \mathcal{K}_1, \kappa_1', \delta_1', \pi_1', \mu_1, \mathsf{parse}(s)\rangle \overset{m}{\rightarrowtail} \langle \mathcal{K}_1', \kappa_2, \delta_2, \pi_1', \mu_2, stop\rangle$. Note that

$$\begin{array}{l}
\mathcal{C}_1 = \langle M_1, c'\rangle \\
\twoheadrightarrow \langle M_5, \texttt{let}\, h = hash(e)\, \texttt{in}\, (k := k \,|\, K_h; \texttt{eval}(trans(e)); K_h := K_h \,|\, depsIn(k, e)); pc := pc'\rangle \\
\rightarrow \langle M_6, k := k|K_h; \texttt{eval}(trans(e)); K_h := K_h|depsIn(k, e); pc := pc'\rangle \\
\rightarrow \langle M_7, \texttt{eval}(trans(e)); K_h := K_h|depsIn(k, e); pc := pc'\rangle \\
\rightarrow \langle M_7, \mathsf{parse}(\hat{M}_7(trans(e))); K_h := K_h|depsIn(k, e); pc := pc'\rangle
\end{array}$$

where

- $M_2 \stackrel{def}{=} M_1[pc' \mapsto M_1(pc)]$;
- $M_3 \stackrel{def}{=} M_2[pc \mapsto M_2(pc) \uplus \{p\}]$;
- $M_4 \stackrel{def}{=} M_3[k_p \mapsto M_3(k_p) \uplus \hat{M}_3(dep(\mathsf{vars}(''e'')))) \uplus M_3(pc')]$;
- $M_5 \stackrel{def}{=} M_4[d_p \mapsto M_3(d_p) \uplus \hat{M}_3(lev(\mathsf{vars}(''e''))))]$;
- $M_6 \stackrel{def}{=} M_5[h \mapsto \hat{M}_5(hash(e))]$
- $M_7 \stackrel{def}{=} M_6[k \mapsto \hat{M}_6(k|K_h)]$

Let $\mathcal{D} \stackrel{def}{=} \langle \mathcal{K}_1, \kappa'_1, \delta'_1, \pi'_1, \mu_1, \mathsf{parse}(s)\rangle$ and $C \stackrel{def}{=} \langle M_7, \mathsf{parse}(\hat{M}_7(trans(e)))\rangle$. In order to apply the I.H. we must first determine whether $\mathcal{D} \prec C$. Among other things (analysed below) we must verify that

$$\mathsf{trans}(\mathsf{parse}(s)) = \mathsf{parse}(\hat{M}_7(trans(e)))$$

We do this now by reasoning as follows:

$$\begin{aligned}
&\mathsf{parse}(\hat{M}_7(trans(e))) \\
&= \mathsf{parse}(\mathcal{I}(M_7(trans(e)))) \text{ (Def. of } \hat{M}_7) \\
&= \mathsf{parse}(\hat{trans}(\mathcal{I}(M_7(e)))) \text{ (Def. 1)} \\
&= \mathsf{parse}(\hat{trans}(\hat{M}_7(e))) \\
&= \mathsf{trans}(\mathsf{parse}(\hat{M}_7(e))) \qquad (Lem.\ 3) \\
&= \mathsf{trans}(\mathsf{parse}(s)) \qquad (\mathcal{D}_1 \prec C_1, Rem.\ 1)
\end{aligned}$$

The remaining items to declare $\mathcal{D} \prec C$ are now considered:

1. $M_7(K) = \mathcal{K}_1$, $M_7(k) = \kappa'_1$, $M_7(d) = \delta'_1$, $M_7(pc) = \pi'_1$. From $\mathcal{D}_1 \prec C_1$ we know that $M_1(K) = \mathcal{K}_1$. Therefore, since $M_1(K) = M_7(K)$, we deduce $M_7(K) = \mathcal{K}_1$. Regarding $M_7(pc) = \pi'_1$ we reason as follows: $M_7(pc) = M_3(pc) = M_2(pc) \uplus \{p\} = M_1(pc) \uplus \{p\} = \pi_1 \uplus \{p\}$ (the last equality is due to $\mathcal{D}_1 \prec C_1$). Regarding, $M_7(k) = \kappa'_1$ we wish to show that $M_7(k) = \kappa'_1 = \kappa_1 \uplus \mathcal{K}_1(h) \uplus \{p \mapsto \pi_1 \cup P\}$. We first reason as follows: $M_7(k) = \hat{M}_6(k|K_h) = M_6(k) \uplus M_6(K_h) = M_6(k) \uplus M_1(K_h) = M_6(k) \uplus \mathcal{K}_1(h)$. We note that
$$M_6(k)(q) \stackrel{def}{=} \begin{cases} M_1(k)(q) & p \neq q \\ M_1(k_p) \uplus \hat{M}_1(dep(\mathsf{vars}(''e''))) \uplus M_1(pc) & p = q \end{cases}$$
and resort to the fact that $\mathcal{D}_1 \prec C_1$, thus concluding the case. The item $M_7(d) = \delta'_1$ is proved along similar lines.
2. $\mu_1(x) = \langle M_7(x), M_7(x_P), M_7(x_L)\rangle$, for all $x \in dom(\mu_1)$. From $\mathcal{D}_1 \prec C_1$ we know that $\mu_1(x) = \langle M_1(x), M_1(x_P), M_1(x_L)\rangle$, for all $x \in dom(\mu_1)$. Since M_7 and M_1 differ only in the values assigned to h and pc' (which is fresh) and k, k_p, d_p, pc (which are not in the domain of μ_1), we conclude the case.

Furthermore,
$\mathsf{depth}(\langle \mathcal{K}_1, \kappa'_1, \delta'_1, \pi'_1, \mu_1, \mathsf{parse}(s)\rangle \stackrel{m}{\rightarrowtail} \langle \mathcal{K}'_1, \kappa_2, \delta_2, \pi'_1, \mu_2, stop\rangle) = (d_1, m) < (d, 1)$ since $d = d_1 + 1$. Therefore, the I.H. yields $C_4 = \langle M_8, stop\rangle$ s.t.

$$\langle \mathcal{K}'_1, \kappa_2, \delta_2, \pi_1, \mu_2, stop\rangle \prec C_4 \tag{6}$$

and

$$\langle M_8, \mathsf{parse}(\hat{M}_8(trans(e)))\rangle \twoheadrightarrow C_4 \tag{7}$$

From (7) we have

$$\begin{aligned}
&\langle M_7, \mathsf{parse}(\hat{M}_7(trans(e))); K_h := K_h|depsIn(k,e); pc := pc'\rangle \\
\twoheadrightarrow\ &\langle M_8, K_h := K_h|depsIn(k,e); pc := pc'\rangle \\
\rightarrow\ &\langle M_8[K_h \mapsto \hat{M}_8(K_h|depsIn(k,e))], pc := pc'\rangle \\
\rightarrow\ &C_5 \stackrel{def}{=} \langle M_8[K_h \mapsto \hat{M}_8(K_h|depsIn(k,e))][pc \mapsto M_8(pc')], stop\rangle
\end{aligned}$$

In order to conclude we set $C_2 \stackrel{def}{=} C_5$ and verify that indeed

$$\begin{aligned}
&\langle \mathcal{K}_1'[h \mapsto \mathcal{K}_1'(h) \uplus \kappa_2|_{\mathsf{parse}(s)}], \kappa_2, \delta_2, \pi_1, \mu_2, stop\rangle \prec \\
&\langle M_8[K_h \mapsto \hat{M}_8(K_h|depsIn(k,e))][pc \mapsto M_8(pc')], stop\rangle = C_5
\end{aligned}$$

If M_9 abbreviates $M_8[K_h \mapsto \hat{M}_8(K_h|depsIn(k,e))][pc \mapsto M_8(pc')]$, then this means we must verify

1. $M_9(K) = \mathcal{K}_1'[h \mapsto \mathcal{K}_1'(h) \uplus \kappa_2|_{\mathsf{parse}(s)}]$, $M_9(k) = \kappa_2$, $M_9(d) = \delta_2$, $M_9(pc) = \pi_1$. From (6) we know that $M_8(k) = \kappa_2$, $M_8(d) = \delta_2$ and $M_8(pc) = \pi_1$. Since $M_8(k) = M_9(k)$, $M_8(d) = M_9(d)$ and $M_8(pc) = M_9(pc)$, all but the first item hold immediately. We now address the first item. From (6) we know that $M_4(K) = \mathcal{K}_1'$. Also, $M_9(K) = \hat{M}_8(K_h|depsIn(k,e)) = \hat{M}_8(K_h) \uplus \hat{M}_8(depsIn(k,e)) = \mathcal{K}_1'(h) \uplus \hat{M}_8(depsIn(k,e))$. We conclude that $\hat{M}_8(depsIn(k,e)) = \kappa_2|_{\mathsf{parse}(s)}$ since this follows from $\hat{depsIn}(\kappa, s) = \kappa|_{\mathsf{parse}(s)}$ which holds by assumption on $depsIn$.
2. $\mu_2(x) = \langle M_9(x), M_9(x_P), M_9(x_L)\rangle$, for all $x \in dom(\mu_2)$. From (6) we know that $\mu_2(x) = \langle M_8(x), M_8(x_P), M_8(x_L)\rangle$, for all $x \in dom(\mu_2)$. Since M_9 differs from M_8 only in the value of K and K is never in the domain of the memory component of the configuration, we conclude.