Information Flow Monitor Inlining

with appendix February 15, 2010

Andrey Chudnov David A. Naumann
Stevens Institute of Technology

Abstract

In recent years it has been shown that dynamic monitoring can be used to soundly enforce information flow policies. For programs distributed in source or bytecode form, the use of JIT compilation makes it difficult to implement monitoring by modifying the language runtime system. An inliner avoids this problem and also serves to provide monitoring for more than one runtime. We show how to inline an information flow monitor, specifically a flow sensitive one previously proved to enforce termination insensitive noninterference. We prove that the inlined version is observationally equivalent to the original.

1. Introduction

Many security requirements involve information flow. Many practical security measures involve runtime mechanisms —but it was long thought that information flow requires static enforcement mechanisms because it is not a trace property [McL94], [CS08] and runtime monitoring enforces trace properties [Sch00]. Hamlen et al [HMS06b] show that any property that can be statically analyzed can also be checked by a runtime monitor with access to the program source. Le Guernic et al. [GBJS06] confirm this for an information flow monitor that performs some static analysis. They suggest that their security automaton could be implemented by program transformation; subsequent work has explored dynamic information flow monitoring but not by transformation. In this paper, we argue that inlining is the only way to go, and we show how to prove an inlining transformation correct.

Security policy assigns categories ("levels", related by a partial order $\sqsubseteq$ [Den76]) to input and output channels with the intention that information may flow from an input of level l to an output of level l' only if $l \sqsubseteq l'$. The notion of "information" is formalized by non-interference [Coh78], [GM82]: in multiple runs of the program, variation in an input at level l is correlated with variation of an output at level l' only if $l \sqsubseteq l'$.

Monitoring of the flow of information via data flows is called taint tracking; it has been implemented for machine code [ZJS+09], bytecode [NSCT08], and interpreted source languages such as Javascript [DG09], [NJK+07]. Taint tracking is quite satisfactory in some circumstances (but see [KHHJ08]). It is inherently incomplete because information easily flows via combinations of control and data flows (usually called "implicit" and "explicit" information flow).

Runtime monitoring (that takes into account both implicit and explict flows) has the potential to be more permissive than purely static enforcement, because it can allow secure executions even if the program also has insecure ones which must not be allowed to run to completion. And the determination of whether a run is secure is made on the basis of precise runtime information as opposed to conservative approximations used in static analysis.

One particularly important scenario in which information flow control is needed is web applications, especially in the browser which interleaves execution of many programs from sources accorded widely varying degrees of trust. The work in this paper is intended to contribute towards the tracking of implicit and explicit information flows in popular web languages such as Javascript. Such languages are dauntingly complex and unattractive for static analysis. Worse yet, the ubiquitous use of dynamic code generation (i.e. *eval*) is a serious impediment to static analysis (though it is not impossible [CMJL09]). In theory, runtime monitoring is better suited to handle *eval* [AS09].

To implement a runtime monitor for an interpreted language like Javascript or Java bytecodes is in principle easier than at the hardware level: complete mediation can be achieved by modifications of the language runtime system because it is involved with every control and data flow event. We argue that such VM monitoring is impractical in the browser setting whereas inlining of a monitor appears to be practical. Our main contributions are the design of an inliner and the proof of its correctness relative to a monitor specified in VM style.

For Javascript there are quite a few VMs in wide

deployment, e.g. SpiderMonkey and TraceMonkey [GES+09] by Mozilla, V8 [Goo] by Google, SquirrelFish by Apple. Internally these engines are very different, so the implementation of a monitor for one engine would not be useful for the others. Moreover, rapid development has rendered prototype monitors obsolete before they were completed.

The most striking reason that VM monitors are impractical is that popular run-times also feature *just-in-time* (JIT) compilation. JIT compilation is a process for generating native machine code at run-time. To understand why it might be a problem for information flow monitoring let's look at the high-level structure of modern run-times like Sun Java HotSpot [Sun], Microsoft .NET Common Language Runtime [Ric06], Mozilla TraceMonkey JavaScript engine [GES+09], Google's V8 JavaScript engine [Goo] and others. In such systems the source code in a high-level language is first compiled to *bytecode* —an instruction set designed for efficient execution by an interpreter (virtual machine). Depending on the language, the program in bytecode form is either distributed to the code consumers (.NET and Java) or is used as an intermediate representation in the JIT compilation process (JavaScript, which is distributed in source form and has no standardized bytecode language). Depending on the specific runtime, bytecode is either interpreted or compiled into native code. This could happen on a method-body-basis upon the first invocation of the method (CLR and V8). In this case eventually all the reachable code would be converted to native code. A more elaborate alternative is to execute the bytecode in the virtual machine and convert only those parts of the code where the interpreter spends most of the time (HotSpot and TraceMonkey).

The key point is that the parts of the program translated to native code are no longer under the step-by-step control of the VM, so it is no longer able to mediate every data and control flow event. A solution is evidently for monitoring to be inlined into the compiled code. It may as well be inlined into the source code, so that one inliner can serve as the front end to many VMs, benefitting from code optimizations of the JIT compiler as well as other performance advances.

The inlining of security monitors dates back more than a decade [ES99], [ES00] but work has focused on access control policies and control flow; the monitors are significantly different from those proposed for information flow. As far as we know, ours is the first work to prove correctness of an inlined information flow monitor. There are two key properties: security, in the sense that allowed runs satisfy termination-insensitive noninterference, and transparency, which means the monitor may abort insecure runs but does not otherwise change the observable behavior of the program. Instead of proving these properties directly, we base our work on a VM monitor which serves as specification and which has been proved to be secure [RS]. It suffices to prove that the observable behaviors of our inlined programs are the same as those run under the VM monitor. This is tricky to get right.

In Sect. 2 we discuss related work and we indicate why we expect our proof technique to be useful for other source languages and monitoring schemes. In Sect. 3 we review the programming language and VM monitor studied by Russo and Sabelfeld [RS], including its security and transparency properties. Sect. 4 presents our inlining transformation; Sect. 5 states the correctness results and sketches the proof. Sect. 6 discusses the prospects for practical inlining. Full details of our proofs can be found in the appendix which is online at http://www.cs.stevens.edu/~naumann/inlining/proofs.pdf

2. Related work

2.1. Information flow monitoring

The problem of monitoring information flow has received wide attention recently, especially in the context of JavaScript security. From the practical side there have been developed at least two modifications of Mozilla's Firefox browser that introduce information flow monitoring in the JavaScript engine [NJK+07], [DG09]. Another information flow monitor implementation was done for Java VM [NSCT08]. Other taint tracking work was cited earlier.

Our interest is in provably sound tracking that handles implicit flows, which has only recently been achieved [VXDS06], [GBJS06], [SST08]. Shroff et al. [SST08] handle implicit flow via function pointers as well as explicit branches, which makes control flow tracking nontrivial. An interesting feature of their work is that control dependencies between program points may be learned over the course of multiple runs (e.g. test runs), eventually yielding soundness without the cost and/or conservativity of static full analysis.

The monitor of LeGuernic et al. [GBJS06] is flow sensitive which means the security level of a memory location may vary from one program point to another; it has been extended to concurrency [Gue07]. Flow sensitivity is essential for low level code (due to reuse of registers and other memory locations). It is also essential for dynamic monitoring of code that isn't equipped with security annotations —whereas, for

static monitoring, label inference and label polymorphism may lessen the need for flow sensitivity.

Russo and Sabelfeld [RS] have recently studied the characteristics of purely dynamic and hybrid (dynamic and static) information flow monitors, in connection with flow sensitivity In the course of their work, they adapt the monitor of [GBJS06], reformulating it in a modular way using labelled transitions and proving a stronger security property that takes intermediate outputs into account. This is the basis for our work, so we discuss it at length in Sect. 3.

The "no sensitive upgrade" rule for dynamic monitoring [Zda02], [AF09] avoids the need for static analysis to determine implicit flows due to branches not taken, but it can reject secure computations. We and others are currently investigating its practical applicability. If it is not found to reject too many useful programs, it would be very desirable because it simplifies runtime monitoring and avoids the high cost and conservativeness of static analysis in the presence of aliasing and eval. Absent definitive evidence about practicality, we believe it is important to explore the feasibility of inlining of monitors that rely on static analysis, for which transparency and other properties are more challenging.

2.2. Monitor in-lining

[ZJS+09] implement taint tracking by inlining in binary code, focusing on engineering issues and experimentation but not proving correctness (nor dealing with implicit flow). Taint tracking at the level of operating systems objects also has a long history and recent resurgence, e.g. [ZBWKM06].

There has been extensive work on inlined monitoring for security policies that concern control flow —in particular, calls on sensitive APIs— as opposed to control flow as an information channel. Being pioneered by Erlingsson and Schneider for Java in [ES99], it was later applied to other languages and run-times. Yu et al. described a similar monitor for JavaScript in [YCIS07]. For assurance that the inlined program satisfies the security property, Hamlen et al. [HMS06a] use a type system to certify in-lined monitors for the .NET run-time. Sridhar and Hamlen [SH10] use model checking to certify in-lined monitors for ActionScript bytecode.

Besides security, it is desirable that monitoring is transparent; this is proved in some recent works including [ADG08], [VP08]. These works are impressive in covering substantial fragments of JVM/CLR bytecode. Dam et al. [DJLP09] prove security and transparency for a language with concurrency, unlike our work and the others we cite on information flow monitoring.

Theoretical studies have formulated monitoring in terms of arbitrary state machines [Sch00]. But the inlining systems cited above for programming languages are for policies about control flow events; monitoring is needed only for invocations of security-sensitive methods. They are not easily adapted to track intermingled implicit and explicit flows of information. For example, although Le Guernic et al. [GBJS06] explicitly formulate their monitor as an automaton, their results are proved from scratch rather than drawing on general results about security automata.

Although they were focused on solving a slightly different problem, Chugh et al. in [CMJL09] describe semantics of an information flow monitor for a significant subset of JavaScript as a set of code rewrite rules, which could be seen as a monitor inlining process.

To leverage the security and transparency results of Russo and Sabelfeld, we need a two-way correspondence between computations of the inlined program and computations of the VM-monitored source program. Steps of the latter correspond to multiple steps of the former and it is a little tricky to express the correspondence and formulate induction hypotheses for the core lemmas. This kind of correspondence resembles atomicity refinements for concurrent programs, but what we need is stronger than one-way refinement. It also resembles correspondence between abstract and concrete hardware processor models for which two-way refinement is needed; but those models are quite different in detail (e.g., the concrete processor state that corresponds to an abstract one might be defined in terms of flushing pipelines).

3. Background

In their recent paper [RS] Russo and Sabelfeld discuss soundness of flow sensitive information flow policy enforcement. Their main result is the impossibility of sound, purely dynamic, flow sensitive monitoring. However, they also define a hybrid, flow sensitive information flow monitor, or rather a family of four variations which differ in how they respond to outputs which would be insecure. They prove security (among other properties) for their VM monitor. We take it as the basis of our work showing how to inline a monitor and leverage those correctness results.

In this section, we present the programming language, monitor family, and security property. We make minor changes to their work which are described in due course; we argue informally that the changes do

$$
\begin{array}{lrcl}
\text{Command events} & \beta & ::= & \alpha \mid t \\
& \alpha & ::= & \epsilon \mid o_l(e, v) \\
& \epsilon & ::= & a(x, e) \mid b(e, c) \mid f \mid s \\
\text{Observations} & \omega & ::= & o_l(v)
\end{array}
$$

Figure 2. Events

not alter the security and transparency properties on which we rely.

3.1. Language semantics

Our language is based on the language used in [RS]. The abstract syntax is presented in figure 1: expressions e include literal values $v \in \mathit{Vals}$ (integers, security levels), arithmetic operations $e \oplus e$, level join $e \sqcup e$, level comparison $e \sqsubseteq e$. We use identifier l for literal levels, which are in a set Levs, with $\mathit{Levs} \subseteq \mathit{Vals}$. Expressions over security levels are used in the information flow tracking code introduced by the inlining transformations. Level expressions are not present in mainstream programming languages nor in the language in [RS]. However, we assume given a finite set Levs equipped with a lattice ordering. Levels and the operations on them can be encoded using integers of sufficient size. We distinguish these expressions merely for clarity where we use them in the transformations. (They may as well be allowed in the source program.)

Inputs are modeled by the initial values of program variables. There is an explicit command, $\mathbf{output}_l(e)$, that immediately outputs the value of e on the "channel" that is visible to observers at level l and above.

The command $\square$ has no transitions; it serves merely to mark terminal configurations. Transitions also generate internal events that are visible only to the VM monitor. These are the elegant way in which Russo and Sabelfeld define their monitor. The syntax of events is displayed in figure 2. The transition events consist of the "skip" event s, variable assignment event $a(x, e)$, branching $b(x, e)$, join of branches f, the "step" event t for the ✓ command, and the event $o_l(e, v)$ for output.

The commands $\succ$ and ✓ are essentially labelled skips. Command $\succ$ generates an internal event that lets the VM monitor synchronize with joint points in control flow; it is inserted into the configuration by the transitions for branching commands.

Command ✓ is never present in the source code, so the VM monitor does not need to handle the t event that it emits. It is used for inlining and is a key ingredient of our correctness proof.

$$
\textsc{Skip}\quad \langle \mathbf{skip},\, m\rangle \xrightarrow{s} \langle \square,\, m\rangle
$$

$$
\textsc{Assign}\quad \frac{m(e) = v}{\langle x := e,\, m\rangle \xrightarrow{a(x,e)} \langle \square,\, m[x \mapsto v]\rangle}
$$

$$
\textsc{Seq1}\quad \frac{\langle c_1,\, m\rangle \xrightarrow{\alpha} \langle \square,\, m'\rangle}{\langle c_1; c_2,\, m\rangle \xrightarrow{\alpha} \langle c_2,\, m'\rangle}
$$

$$
\textsc{Seq2}\quad \frac{\langle c_1,\, m\rangle \xrightarrow{\alpha} \langle c_1',\, m'\rangle \qquad c_1' \neq \square}{\langle c_1; c_2,\, m\rangle \xrightarrow{\alpha} \langle c_1'; c_2,\, m'\rangle}
$$

$$
\textsc{IfThen}\quad \frac{m(e) \neq 0}{\langle \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2,\, m\rangle \xrightarrow{b(e,c_2)} \langle c_1; \succ,\, m\rangle}
$$

$$
\textsc{IfElse}\quad \frac{m(e) = 0}{\langle \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2,\, m\rangle \xrightarrow{b(e,c_1)} \langle c_2; \succ,\, m\rangle}
$$

$$
\textsc{WhileLoop}\quad \frac{m(e) \neq 0}{\langle \mathbf{while}\ e\ \mathbf{do}\ c,\, m\rangle \xrightarrow{b(e,\mathbf{skip})} \langle c; \succ; \mathbf{while}\ e\ \mathbf{do}\ c,\, m\rangle}
$$

$$
\textsc{WhileSkip}\quad \frac{m(e) = 0}{\langle \mathbf{while}\ e\ \mathbf{do}\ c,\, m\rangle \xrightarrow{b(e,c)} \langle \succ,\, m\rangle}
$$

$$
\textsc{Output}\quad \frac{m(e) = v}{\langle \mathbf{output}_l(e),\, m\rangle \xrightarrow{o_l(e,v)} \langle \square,\, m\rangle}
$$

$$
\textsc{End}\quad \langle \succ,\, m\rangle \xrightarrow{f} \langle \square,\, m\rangle \qquad \textsc{Step}\quad \langle ✓,\, m\rangle \xrightarrow{t} \langle \square,\, m\rangle
$$

Figure 3. Language semantics

The semantics of the language is defined in Fig. 3 via a set of labeled small-step transitions over configurations of the form $\langle c,\, m\rangle$, where $m : \mathit{Vars} \rightharpoonup \mathit{Vals}$ is a memory. In the initial configuration the domain of m would be $\mathit{vars}(c)$ (and its domain is invariant under the semantic transitions).

We extend m homomorphically to expressions: for values, $m(v) = v$; for arithmetic operations, $m(e +$

$$\begin{array}{llll} \textit{Expressions} & e & ::= & v \mid x \mid e \oplus e \mid e \sqsubseteq e \qquad\qquad v \in \textit{Vals},\ x \in \textit{Vars},\ \oplus \in \{+, \sqcup, \ldots\} \\ \textit{Commands} & c & ::= & \textbf{skip} \mid x := e \mid c; c \mid \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c \mid \textbf{while}\ e\ \textbf{do}\ c \mid \textbf{output}_l(e) \mid \Box \mid \succ \mid \checkmark \end{array}$$

Figure 1. Abstract syntax.

$e') = m(e) + m(e')$ etc; for level join,

$$m(e \sqcup e') = m(e) \sqcup m(e')$$

where $\sqcup$ is the lattice join; and $m(e \sqsubseteq e')$ is 0 or 1 according to whether $m(e) \sqsubseteq m(e')$ in *Levs*. These would all be total functions if we encoded levels as integers, and the level expressions introduced by inlining will be type-correct —so we refrain from fussing about expression types.

3.2. VM monitor

Our information flow monitor is hybrid and flow sensitive. Hybrid means that it employs static analysis of the conditional branches that were not taken in the current run. Flow sensitive means that security levels of variables could by updated during the run. It is a generalization of a monitor by Russo and Sabelfeld [RS] to a multi-level security lattice.

The monitor runs alongside the program in a lock-step fashion, as follows.

Definition 1 (monitored transitions) The transition relation $\xrightarrow{\alpha}_\gamma$ on monitor configurations is defined by

$$\frac{\langle c,\, m\rangle \xrightarrow{\alpha} \langle c',\, m'\rangle \qquad \langle \Gamma,\, \Delta\rangle \xrightarrow{\alpha}_\gamma \langle \Gamma',\, \Delta'\rangle}{\langle\langle c,\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle \xrightarrow{\gamma} \langle\langle c',\, m'\rangle,\, \langle \Gamma',\, \Delta'\rangle\rangle}$$

At each step of the program the monitor inspects the event α generated by the program, updates its state and, if necessary, generates an output event ω.

The transitions of our VM monitor are presented in figure 4. These labelled transitions relate monitor configurations of the form $\langle \Gamma,\, \Delta\rangle$. Here $\Gamma : \textit{Vars} \rightharpoonup \textit{Levs}$ is a partial function from variable names to security levels, to track the level of each variable's current value. The *control context* Δ is a list of pairs of the form (xs, l) where xs is a set of variables and l is a security level; a pair corresponds to a branch point with l the level of the branch condition and xs the variables that may be written on the branch that wasn't taken in the current run.

We extend labelings Γ homomorphically to expressions: $\Gamma(e)$ is the join of the levels $\Gamma(x)$ of variables x in e.

We are only concerned with monitored executions which begin with $\langle \Gamma_0,\, (\bot :: [\,])\rangle$, where Γ_0 is the initial

$$\textsc{M-skip}\quad \langle \Gamma,\, \Delta\rangle \xrightarrow{s} \langle \Gamma,\, \Delta\rangle$$

$$\textsc{M-assign}\quad \langle \Gamma,\, \Delta\rangle \xrightarrow{a(x,e)} \langle \Gamma[x \mapsto \textit{lev}(\Delta) \sqcup \Gamma(e)],\, \Delta\rangle$$

$$\textsc{M-branch}\quad \frac{l = \Gamma(e) \sqcup \textit{lev}(\Delta)}{\langle \Gamma,\, \Delta\rangle \xrightarrow{b(e,c)} \langle \Gamma,\, (\text{update}_l(c) :: \Delta)\rangle}$$

$$\textsc{M-join}\quad \langle \Gamma,\, ((xs, l) :: \Delta)\rangle \xrightarrow{f} \langle \Gamma \sqcup \textit{mkFun}((xs, l)),\, \Delta\rangle$$

$$\textsc{M-OutputFailstop}\quad \frac{\textit{lev}(\Delta) \sqcup \Gamma(e) \sqsubseteq l}{\langle \Gamma,\, \Delta\rangle \xrightarrow{o_l(e,v)}_{o_l(v')} \langle \Gamma,\, \Delta\rangle}$$

$$\textsc{M-OutputDefault}\quad \frac{\Gamma(e) \sqsubseteq l \implies v' = v \qquad \Gamma(e) \not\sqsubseteq l \implies v' = D}{\langle \Gamma,\, \Delta\rangle \xrightarrow{o_l(e,v)}_{o_l(v')} \langle \Gamma,\, \Delta\rangle}$$

$$\textsc{M-OutputSuppress}\quad \frac{\textit{lev}(\Delta) \sqcup \Gamma(e) \sqsubseteq l \implies \gamma = o_l(v) \qquad \textit{lev}(\Delta) \sqcup \Gamma(e) \not\sqsubseteq l \implies \gamma \text{ is nothing}}{\langle \Gamma,\, \Delta\rangle \xrightarrow{o_l(e,v)}_{\gamma} \langle \Gamma,\, \Delta\rangle}$$

$$\textsc{M-OutputDefault/Suppress}\quad \frac{\textit{lev}(\Delta) \not\sqsubseteq l \implies \gamma \text{ is nothing} \qquad \textit{lev}(\Delta) \sqcup \Gamma(e) \sqsubseteq l \implies \gamma = o_l(v) \qquad \textit{lev}(\Delta) \sqsubseteq l \wedge \Gamma(e) \not\sqsubseteq l \implies \gamma = o_l(D)}{\langle \Gamma,\, \Delta\rangle \xrightarrow{o_l(e,v)}_{\gamma} \langle \Gamma,\, \Delta\rangle}$$

Figure 4. VM monitor transitions

labeling and $\bot$ is the bottom level of the security lattice *Levs*, and where $dom(m) = dom(\Gamma_0)$. (We write $[\,]$ for the empty list and use the ML notation :: for prefixing an element to a list.)

To create a pair to push onto Δ, we use the function $\text{update}_l(c)$ defined by

$$\text{update}_l(c) = (mod(c), l) \tag{1}$$

using the set $mod(c)$ of assignment targets of c (defined in Fig. 5). A pair (xs, l) is converted to an

$$
\begin{array}{lcl}
mod(\textbf{skip}) & = & \emptyset \\
mod(\succ) & = & \emptyset \\
mod(\Box) & = & \emptyset \\
mod(c_1; c_2) & = & mod(c_1) \cup mod(c_2) \\
mod(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2) & = & mod(c_1) \cup mod(c_2) \\
mod(\textbf{while } e \textbf{ do } c) & = & mod(c) \\
mod(\textbf{output}_l(e)) & = & \emptyset \\
mod(x := e) & = & \{x\}
\end{array}
$$

Figure 5. The modifiable variables of a command.

l-labelling of the variables xs by $mkFun()$ defined by

$$mkFun((xs, l)) = \{[x \mapsto l] \mid x \in xs\}$$

Observe the invariant that if $\Delta = ((xs, l) :: \Delta')$ then l is the join of all the levels of the branch conditions that the current command implicitly depends on (i.e. including Δ'). The rule [M-Branch] makes sure of that.

The join $\Gamma \sqcup \Gamma'$ of two labelings has as its domain the union of the domains of Γ and Γ'; it is defined in Fig. 6.

The monitor uses a function lev to get the level of the current control dependence region, defined by

$$
\begin{array}{rcl}
lev([\,]) & = & \bot \\
lev(((xs, l) :: \Delta)) & = & l
\end{array}
$$

The definitions provide a family of four monitors, all the same except for their treatment of output events. A family member is designated by choosing one of the four [M-Output*] rules in Fig. 1. The variations have to do with response to insecure output. The rules also take care of a technical issue: for communication between the monitor and monitored program, we need the internal output event to include the output expression so its levels can be checked. But this should not appear in the actual output, which is captured by events as subscript on the transition arrow (cf. ω in Fig. 2).

The formulation in Russo and Sabelfeld [RS] is slightly more general than ours, in that they specify the property required of $update$ rather than stipulating a particular definition. This caters for more sophisticated features such as aliased objects, for which a practical definition would be imprecise. The property holds for our definitions, which disentangle the $mod()$ function from $mkFun()$. The important thing is that $mod()$ covers possible state changes in the usual sense: for any $m, c, m', c', \vec{\alpha}, x$, if $\langle c,\, m\rangle \stackrel{\vec{\alpha}}{\Longrightarrow} \langle c',\, m'\rangle$ and $x \notin mod(c)$ then $m(x) = m(x')$.

For monitored transitions, we define

$$
\begin{array}{c}
\langle\langle c,\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle \stackrel{\omega}{\Longrightarrow} \langle\langle c',\, m'\rangle,\, \langle \Gamma',\, \Delta'\rangle\rangle \\
\text{iff} \\
\langle\langle c,\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle \longrightarrow^* \stackrel{\omega}{\longrightarrow} \longrightarrow^* \langle\langle c',\, m'\rangle,\, \langle \Gamma',\, \Delta'\rangle\rangle
\end{array}
$$

We write $\langle c,\, m\rangle \stackrel{\vec{\omega}}{\Longrightarrow} \langle c',\, m'\rangle$ for the transitive closure with $\vec{\omega}$ the catenation of labels.

The principal difference between our monitor and that of Russo and Sabelfeld is the fact that our monitor works with a multi-level security lattice. This difference is manifested in the structure of Δ (Γ_s in [RS]), the definition of lev and in the definition of update$_l(c)$, which is defined as a partial function from variables to security levels, rather than a tuple. Aside of that, the semantics of the monitors is the same. We think that these differences are small enough to omit formal arguments for their equivalence.

3.3. Security property

Security policy is comprised of the lattice $(Levs, \sqsubseteq)$, the initial labelling Γ_0 of the program variables, and the fixed assignment of levels to outputs that is embodied in the program syntax.

Our attacker model is the same as in [RS]. We assume that the attacker knows or supplies the monitored programs and has an arbitrary power of reasoning about program semantics. However, the attacker does not have control over the virtual machine, the monitor or the inliner. He cannot observe the values stored in memory directly, nor can he influence the behavior of the system in any way other than supplying the programs that are subject to monitoring/transformation and providing non-secret inputs, which are modeled as the initial values of low variables.

We assume that the attacker can observe all the public outputs; however he is insensitive to power consumption, time or any other covert channels. We believe that the described attacker model fits the threat space of mobile code well. For example, in a typical cross-site scripting scenario the attacker can supply malicious JavaScript code and observe the requests made to a web-site he controls, but he cannot influence the work of the user system directly, nor intercept the communication between the user and the trusted web-site.

The desired security property is *termination insensitive non-interference* (TINI, [RS], [AHSS08]). It is concerned with what may be learned about the initial values of variables that are initially labelled secret, by an observer who sees low outputs and the initial values of low variables, but nothing else. Termination insensitivity is attractive because it admits more programs; it is adequate because, even in the presence of output, ignoring termination does not allow leakage of secrets in polynomial time [AHSS08]. The term "progress insensitivity" is sometimes used, because the formal definition says that what is learned from observing

$$
\begin{array}{rcll}
(\Gamma \sqcup \Gamma')(x) & = & \Gamma(x) \sqcup \Gamma'(x) & \text{if } x \in dom(\Gamma) \cap dom(\Gamma') \\
& = & \Gamma(x) & \text{if } x \in dom(\Gamma) - dom(\Gamma') \\
& = & \Gamma'(x) & \text{if } x \in dom(\Gamma') - dom(\Gamma)
\end{array}
$$

Figure 6. Join of partial labellings.

a specific run is no more than what is learned by knowing that there exists a run of the same length.

A little more formally, a command c satisfies TINI under the monitored semantics if the following holds for attackers at any level l. For any initial memories $m1, m2$ that agree on variables x such that $\Gamma_0(x) \not\sqsubseteq l$, if $\langle c,\, m1\rangle \stackrel{\vec{\omega}}{\Longrightarrow} \langle c',\, m1'\rangle$ then there are $m2'$ and $\vec{\omega}'$ such that $\langle c,\, m2\rangle \stackrel{\vec{\omega}'}{\Longrightarrow} \langle c',\, m2'\rangle$ and either (a) $\vec{\omega}$ and $\vec{\omega}'$ have the same sequences of l-visible events (i.e. outputs on channels of level $\leq l$), or (b) the l-visible events of $\vec{\omega}'$ are a prefix of those of $\vec{\omega}$ and configuration $\langle c',\, m2'\rangle$ is stuck due to the monitor.

Russo and Sabelfeld prove that their monitor is sound: Every command satisfies TINI. Their monitor using monitoring rule [M-OutputFailstop] is also transparent: for any $\langle c,\, m\rangle$, every monitored run produces a prefix of the observable outputs of $\langle c,\, m\rangle$ under the ordinary semantics; it may be a proper prefix in case the monitor stops the run due to security violation. (Similar properties hold using the other responses to violations.)

In section 5 we establish soundness of our in-lining process by proving that for any source program its transformation produces the same sequence of outputs and terminates at the same time as the monitored program starting in equivalent states. Through this property we argue that all programs with the monitor inlined are secure and transparent.

4. Inlining

The in-lining process is governed by transformation judgements in the form $G, S, I \vdash c \blacktriangleright \tilde{c}$ for commands and $S \vdash e \blacktriangleright \tilde{e}$ for expressions. The command rules are displayed in Fig. 8 and the expression rules are in Fig. 9.

Function $S : vars(c) \rightarrow Vars$ maps each variable of c to its *shadow variable* $S(x)$ which will hold the current security level of x. We require that S is injective and its range should be disjoint from $vars(c)$. The finite list G contains names of the variables that store security levels of the conditional guard expressions in the current control dependence region with the $head(G)$ being the security level of the innermost conditional or loop guard. The finite list I holds sets of variables of c that could possibly be updated in the branches-not-taken. We require $|I| = |G|$ and moreover the elements of G are disjoint from $rng(S)$. The transformation rules maintain these conditions.

In the inlined program, the state includes the memory m for the original variables and a memory mst for the variables in G and $rng(S)$. The following definition describes correspondence between mst and a VM-monitor configuration. It uses the standard function that converts a pair of lists into a list of pairs:

$$
\begin{array}{lcl}
zip((a :: as), (b :: bs)) & = & ((a, b) :: zip(as, bs)) \\
zip(as, bs) & = & [\,] \quad \text{if } as = [\,] \text{ or } bs = [\,]
\end{array}
$$

Definition 2 (monitor state coupling) Define $\langle \Gamma,\, \Delta\rangle \cong (G, S, I, mst)$ iff $\Gamma = mst \circ S$ and $\Delta = zip(I, mst \circ G)$.

The transformed program should start in a configuration $\langle \tilde{c},\, m_0 \uplus mst_0\rangle$ where m_0 is the initial user memory and mst_0 is the initial *monitor state*. The monitor state satisfies the invariant: $dom(mst) = rng(S) \cup G$. The initial monitor state mst_0 should be coupled with $\langle \Gamma_0,\, \Delta_0\rangle$.

We prove that if the initial memory and monitor state are disjoint they stay disjoint along the whole execution of the program.

Observe that neither G, nor S, nor I are run-time structures – they are only used during the in-lining process and define the layout of a part of program memory which is used to store the monitor state.

The meaning of $G, S, I \vdash c \blacktriangleright \tilde{c}$ is that under the given G, S and I the command c is transformed to a command $\tilde{c}$. The meaning of the rules for expressions is the same save for they do not use G or I.

For a program c its transformed version $\tilde{c}$ could be obtained by applying the transformation rule

$$(x :: [\,]), init(c), (\emptyset :: [\,]) \vdash c \blacktriangleright \tilde{c}$$

where fresh variable x is for the level of the initial security context, $init(c)$ gives us a total injective mapping from $vars(c)$ to a set of variables disjoint from $vars(c)$, and $\emptyset$ is the branch-not-taken set for the initial context.

The transformed programs include commands $\succ$ and $\checkmark$ that have no observable behavior and would be omitted by an implementation; for us they serve as annotations to facilitate the correctness proof.

The transformation process introduces a number of additional commands that track explicit and implicit information flows during the execution of the program. These commands mimic the operations that a VM-monitor does at each step of the computation.

For example, the [TR-Assign] rule introduces an extra assignment command that updates a shadow variable $S(x)$ to the least upper bound of the level of the expression e (the value of the transformed expression $\tilde{e}$) and the level of the control context stored in the variable $head(G)$ on top of the stack of guard levels G. The definition of monitor state coupling tells us that $S(x)$ corresponds to $\Gamma(x)$ and the VM monitor updates it to the same security level as the in-lined monitor. The rule also includes a ✓ command which reflects that, after the transformed program updates x's shadow and x itself, the state corresponds to the state of the VM-monitored program after one step.

According to the rule for conditional [TR-If] the program rewriter would first find a fresh variable name x. This variable would store the security level of the conditional guard; the command $x := \tilde{e} \sqcup head(G)$ will make sure of that. Next, it should transform each branch c_i, but in a different environment: x should be pushed on top of the guard level stack G to reflect the control dependency on the guard. Also the branch-not-taken effect stack I should be updated with the set of variables that might be assigned in the other branch in order to account for implicit flows. This mimics the operations that are done by the VM-monitor on the $b(e, c)$ event. A final touch is putting the ✓ commands in the proper places. It is, perhaps, strange to see not one but three ✓ commands introduced by the rule, knowing that each roughly corresponds to one step taken by the VM-monitored program. However, observe that two of these commands are in the branches. Which means only one gets to be executed because only one of the branches could be taken. Among the two ✓ commands introduced by this rule the first one serves to mark the end of evaluating the guard expression and taking the branching decision. The trailing ✓ has to do with the $\succ$ command that the conditional is reduced to when the branch is finished executing. Since it was not present in the original code, the transformation could not be applied to it. However, a $\succ$ command in the VM-monitored execution will take a step which should be matched by a t in the trace of transformed program. Thus, the [TR-If] rule introduces an extra ✓ for the [TR-End] rule. Also, in order to make the [TR-end] rule work, the rule instructs to introduce the commands given by $\kappa(mod(c_{i \bmod 2+1}), S, x)$.

The functions $mod(c)$ (figure 5) and $\kappa(i, S, e_L)$ (figure 7) are used to generate code that is an in-lined implementation of the update$_l(c)$ function. The result of κ performs compensation of the security levels of variables for the implicit flows resulting from the fact that the other branch was not taken.

$$\text{TR-VALUE}\quad \frac{}{S \vdash v \blacktriangleright \bot} \qquad \text{TR-VAR}\quad \frac{}{S \vdash x \blacktriangleright S(x)}$$

$$\text{TR-OP}\quad \frac{\forall i \in 1..2.\ S \vdash e_i \blacktriangleright \tilde{e}_i}{S \vdash e_1 \oplus e_2 \blacktriangleright \tilde{e}_1 \sqcup \tilde{e}_2} \qquad \text{TR-ORD}\quad \frac{\forall i \in 1..2.\ S \vdash e_i \blacktriangleright \tilde{e}_i}{S \vdash e_1 \sqsubseteq e_2 \blacktriangleright \tilde{e}_1 \sqcup \tilde{e}_2}$$

Figure 9. Transformation rules for expressions.

Now that we have used [TR-Assign] and [TR-If] as a case study to the key principles and components of the transformation rules, we will only highlight the principal ideas behind the rest.

The rule for **while** is similar to the previous rule. We introduce a new variable x to hold the level of the guard which gets updated at the end of each iteration to capture flow sensitivity. The loop body is transformed taking the control dependency on the guard value into account. At the end of the loop there is a compensation $\kappa(mod(c), S, x)$ for the fact that the loop body is not taken. The extra **skip**, $\succ$ and ✓ commands are a requirement of soundness proofs. All these extra skips could be removed from transformation rules for "production" use since they do not modify the memory, alter the control flow or produce any observable outputs.

The family of rules [TR-Output*] is the direct counterpart of the family of monitor behaviors [M-Output*]. Note that only one of the rules should be used in the actual transformation. As in [RS] we leave the user a choice of enforcement strategy while maintaining same guarantees about the resulting program behavior.

The rules in figure 10 should not be used in the inlining process. They are only used in the proof of Theorem 1 and help build correspondence between the commands in the VM configurations and in the inlined monitor configurations during the execution (see Def. 3 below). Justification of these rules is rather lengthy and is better explained by execution traces presented in the proof for Theorem 1. We encourage interested readers to refer to the online appendix for details.

5. Soundness

We show this by proving that transformed programs are *observationaly equivalent* to programs monitored

$$\kappa(xs, S, y) = \begin{cases} S(x) := y \sqcup S(x); \kappa(xs \setminus x, S, y) & \text{if } x \in xs \\ \textbf{skip} & \text{if } xs = \emptyset \end{cases}$$

Figure 7. Definition of the compensation function, κ, by recursion on the variable set xs

TR-SKIP
$$G, S, I \vdash \textbf{skip} \blacktriangleright \textbf{skip}; \checkmark$$

TR-ASSIGN
$$\frac{S \vdash e \blacktriangleright \tilde{e}}{G, S, I \vdash x := e \blacktriangleright S(x) := \tilde{e} \sqcup head(G); x := e; \checkmark}$$

TR-SEQ
$$\frac{\forall i \in 1..2 \mid G, S, I \vdash c_i \blacktriangleright \tilde{c_i} \qquad c_1 \neq \succ}{G, S, I \vdash c_1; c_2 \blacktriangleright \tilde{c_1}; \tilde{c_2}}$$

TR-IF
$$\frac{S \vdash e \blacktriangleright \tilde{e} \qquad x \notin \mathrm{dom}(S) \cup \mathrm{rng}(S) \wedge x \notin G \qquad (x :: G), S, (mod(c_{i \bmod 2+1}) :: I) \vdash c_i \blacktriangleright \tilde{c_i} \qquad \tilde{c_i}' = \checkmark; \tilde{c_i}; \kappa(mod(c_{i \bmod 2+1}), S, x)}{G, S, I \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \blacktriangleright x := \tilde{e} \sqcup head(G); \textbf{if } e \textbf{ then } \tilde{c_1}' \textbf{ else } \tilde{c_2}'; \checkmark}$$

TR-WHILE
$$\frac{S \vdash e \blacktriangleright \tilde{e} \qquad x \notin \mathrm{dom}(S) \cup \mathrm{rng}(S) \wedge x \notin G \qquad (x :: G), S, (\emptyset :: I) \vdash c \blacktriangleright \tilde{c}}{G, S, I \vdash \textbf{while } e \textbf{ do } c \blacktriangleright x := \tilde{e} \sqcup head(G); \textbf{while } e \textbf{ do } (\checkmark; \tilde{c}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(mod(c), S, x); \succ; \checkmark}$$

TR-OUTPUTFAILSTOP
$$\frac{S \vdash e \blacktriangleright \tilde{e}}{G, S, I \vdash \textbf{output}_l(e) \blacktriangleright \textbf{if } head(G) \sqcup \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else } (\textbf{diverge}); \checkmark}$$

TR-OUTPUTDEFAULT
$$\frac{S \vdash e \blacktriangleright \tilde{e}}{G, S, I \vdash \textbf{output}_l(e) \blacktriangleright \textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D); \checkmark}$$

TR-OUTPUTSUPPRESS
$$\frac{S \vdash e \blacktriangleright \tilde{e}}{G, S, I \vdash \textbf{output}_l(e) \blacktriangleright \textbf{if } (head(G) \sqcup \tilde{e}) \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else skip}; \checkmark}$$

TR-OUTPUTDEFAULT/SUPPRESS
$$\frac{S \vdash e \blacktriangleright \tilde{e}}{G, S, I \vdash \textbf{output}_l(e) \blacktriangleright \textbf{if } head(G) \sqsubseteq l \textbf{ then } (\textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D)) \textbf{ else skip}; \checkmark}$$

Figure 8. Transformation rules for commands

by the VM monitor. Two parallel runs of the monitored and the transformed programs, starting with the same user memory and compatible monitor states, produce the same output traces and end up with the same user memory and compatible monitor states.

We formalize this notion in Theorem 1. The theorem uses a notion of VM and IM configuration coupling which uses monitor state coupling defined in Sect. 4.

Definition 3 (Configuration coupling)
$\langle\langle c, m_1\rangle, \langle \Gamma, \Delta\rangle\rangle \sim \langle \tilde{c}, m_2 \uplus mst\rangle$ if and only if $m_1 = m_2$ and there are G, S, I such that $G, S, I \vdash c \blacktriangleright \tilde{c}$ and $\langle \Gamma, \Delta\rangle \cong (G, S, I, mst)$

For transitions of command configurations, we define $\stackrel{\beta}{\Longrightarrow}$ by

$$\langle \tilde{c}, m \uplus mst\rangle \stackrel{\beta}{\Longrightarrow} \langle \tilde{c'}, m' \uplus mst'\rangle$$
iff
$$\langle \tilde{c}, m \uplus mst\rangle \longrightarrow^* \stackrel{\beta}{\longrightarrow} \longrightarrow^* \langle \tilde{c'}, m' \uplus mst'\rangle$$

We write $\stackrel{\vec{\beta}}{\Longrightarrow}$ for the transitive closure, catenating labels.

TR-STOP
$G, S, I \vdash \square \blacktriangleright \square$

TR-END
$G, S, I \vdash \succ \blacktriangleright \kappa(head(I), S, head(G)); \succ; \checkmark$

TR-ENDSEQ
$$\frac{tail(G), S, tail(I) \vdash c \blacktriangleright \tilde{c}}{G, S, I \vdash \succ; c \blacktriangleright (\kappa(head(I), S, head(G)); \succ; \checkmark); \tilde{c}}$$

TR-WHILELOOP
$$\frac{S \vdash e \blacktriangleright \tilde{e} \qquad (x :: G), S, (\emptyset :: I) \vdash c \blacktriangleright \tilde{c}}{G, S, I \vdash \textbf{while } e \textbf{ do } c \blacktriangleright x := \tilde{e} \sqcup x; \succ; \textbf{while } e \textbf{ do } (\checkmark; \tilde{c}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(mod(c), S, x); \succ; \checkmark}$$

Figure 10. Additional transformation rules used in proofs

The transition semantics in Fig. 3 includes internal events for synchronization with the monitor. To streamline the paper, we use the same semantics for transformed programs, but for these programs we are only concerned with outputs. Moreover, the internal event $o_l(e, v)$ that represents an output contains an expression e which is not supposed to be observable. The following relation connects traces of internal events with traces of monitored executions.

Definition 4 (Output relation) We define $\vec{\beta} \rhd \vec{\omega}$ to hold iff $\vec{\omega}$ is obtained from $\vec{\beta}$ by deleting non-output events and changing each $o_l(e, v)$ to v.

Theorem 1 Observational equivalence.

For all $c, \tilde{c}, m, m', mst, \Gamma, \Delta$, if $\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle \sim \langle \tilde{c},\, m \uplus mst\rangle$ then we have

(a) For all c', Γ', Δ', if $\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle \stackrel{\vec{\omega}}{\Longrightarrow} \langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle$ then there is $\vec{\beta}$ such that $\vec{\beta} \rhd \vec{\omega}$ and $\langle c,\, m \uplus mst\rangle \stackrel{\vec{\beta}}{\Longrightarrow} \langle \tilde{c}',\, m' \uplus mst'\rangle$ and $\langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle \sim \langle \tilde{c}',\, m' \uplus mst'\rangle$.

(b) For all $\tilde{c}'$, mst', if $\langle \tilde{c},\, m \uplus mst\rangle \stackrel{\vec{\beta}}{\Longrightarrow}\!\!\xrightarrow{t} \langle \tilde{c}',\, m' \uplus mst'\rangle$ then there is $\vec{\omega}$ such that $\vec{\beta} \rhd \vec{\omega}$ and $\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle \stackrel{\vec{\omega}}{\Longrightarrow} \langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle$ and $\langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle \sim \langle \tilde{c}',\, m' \uplus mst'\rangle$.

Proof: Here we only state proof ideas. For complete proofs, please, refer to the online appendix.

Proof of clause **(a)** goes by induction on the number of steps of the VM monitored configuration $\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle$. For each step, we go by cases on c and compare one step of the monitored configuration to a sequence of steps of the transformed configuration $\langle \tilde{c},\, m \uplus mst\rangle$ finishing with a t-transition. In each case we make sure that if the step of the monitored configuration produces an output $o_l(v)$ then there is an output event $o_l(e, v)$ in the trace of the transformed configuration. If no output is produced, both traces should agree on that. To find instances G', S' and I' for which the resulting configurations are coupled, we are typically guided by the need to establish monitor state coupling: $\langle\Gamma',\, \Delta'\rangle \cong (G', S', I', mst')$. We are also guided by the form of the resulting commands, since we need to connect them according to the transformation $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$.

We sketch the case for $c = x := e$ as an illustration. First we figure out the transformed command. According to [TR-Assign]: $\tilde{c} = S(x) := \tilde{e} \sqcup g; x := e; \checkmark$ where $S \vdash e \blacktriangleright \tilde{e}$ and $g = head(G)$. Next, we show a step of the VM-monitored execution and note the event produced and the resulting configuration.

$$\begin{array}{ll} & \langle\langle x := e,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle \\ \longrightarrow & \\ & \langle\langle\square,\, m[x \mapsto m(e)]\rangle,\, \langle\Gamma[x \mapsto lev(\Delta) \sqcup \Gamma(e)],\, \Delta\rangle\rangle \\ = & \\ & \langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta\rangle\rangle \qquad \text{define } c', m', \Gamma' \end{array}$$

Next, we show the trace of the transformed command until the first t-event:

$$\begin{array}{ll} & \langle S(x) := \tilde{e} \sqcup g; x := e; \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{\alpha_1} & \qquad \text{where } \alpha_1 = a(S(x), \tilde{e} \sqcup g) \\ & \langle x := e; \checkmark,\, m \uplus mst[S(x) \mapsto mst(\tilde{e}) \sqcup g]\rangle \\ \xrightarrow{\alpha_2} & \qquad \text{where } \alpha_2 = a(x, e) \\ & \langle\checkmark,\, m[x \mapsto m(e)] \uplus mst[S(x) \mapsto mst(\tilde{e}) \sqcup g]\rangle \\ \xrightarrow{t} & \\ & \langle\square,\, m[x \mapsto m(e)] \uplus mst[S(x) \mapsto mst(\tilde{e}) \sqcup g]\rangle \\ = & \\ & \langle \tilde{c}',\, m' \uplus mst'\rangle \qquad \text{define } \tilde{c}', mst' \end{array}$$

By [TR-Stop] we obtain a valid transformation relation between c' and $\tilde{c}'$ in this case: $G', S', I' \vdash \square \blacktriangleright \square$ —for any G', S' and I'. In this case we can take $G' = G$, $S' = S$, $I' = I$ and get $\langle\Gamma',\, \Delta\rangle \cong (G, S, I, mst')$ because $\Gamma' = mst' \circ S$ and $\Delta' = zip(I, mst' \circ G)$.

In another case, $c =$ **while** e **do** c_1, we need to consider two subcases: $m(e) = 0$ and $m(e) \neq 0$. In each of them we consider two subcases for the rule used to find $\tilde{c}$: [TR-While] and [TR-WhileLoop]. The

proof is long, so we only highlight here the trace of $\langle \tilde{c},\ m \uplus mst \rangle$ (Fig. 11) used in the subcase $m(e) \neq 0$ and we assume $G, S, I \vdash c \blacktriangleright \tilde{c}$ according to [TR-While]. And we use the same proof tactic as before.

For clause **(b)** of the Theorem, the proof goes by induction on the number of t-events in the trace of the inlined monitor trace. For each t-event, we consider the steps

$$\langle \tilde{c},\ m \uplus mst \rangle \stackrel{\alpha}{\Longrightarrow} \stackrel{t}{\longrightarrow} \langle \tilde{c}',\ m' \uplus mst' \rangle$$

leading to it (by unrolling the semantics of $\tilde{c}$). Going by cases on c, and using the assumption

$$\langle\langle c,\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle \sim \langle \tilde{c},\ m \uplus mst\rangle$$

the semantics gives us c', Γ' and Δ' for the step

$$\langle\langle c,\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle \stackrel{\alpha'}{\longrightarrow} \langle\langle c',\ m'\rangle,\ \langle \Gamma',\ \Delta'\rangle\rangle$$

and it remains to check that $\alpha' = \alpha$ and

$$\langle\langle c',\ m'\rangle,\ \langle \Gamma',\ \Delta'\rangle\rangle \sim \langle \tilde{c}',\ m' \uplus mst'\rangle$$

Strictly speaking, this is an inner induction on the structure of c, because the transition relation is defined inductively in the case of sequence.

Note that for both clauses **(a)** and **(b)** the proofs take advantage of determinacy of the transition relations. □

The condition used in Theorem 1 resembles a familiar notion of *bisimulation*. The difference is in the fact that the executions do not match each other's steps, as in classic bisimulation. In fact, it is impossible for a transformation of a non-trivial program to match the steps of the VM-monitored program precisely because the transformed program is simply larger due to additional commands that update the monitor state and enforce the policy. For a given output trace, would necessarily make more steps that its VM-monitored counterpart.

A direct consequence of our theorem and the soundness results of [RS] is the following.

Corollary 1 Every run of the transformed program satisfies termination-insensitive non-interference with respect to the given policy.

In addition, transparency of our inlined monitors is a direct consequence of transparency for the VM monitor. That should be easily proved, owing to the elegant form of the monitor and its linking with the monitored program; however, we did not find a formal result in [RS].

6. Discussion

Our long term goal is provably secure information flow control for Javascript. For reasons discussed in the introduction, it needs to be done by an inlined monitor. Inlining was suggested by Le Guernic et al. [GBJS06] but we have not found any results on provably correct inlining for an information flow monitors. As a first step, we investigated a very simple programming language and found that already it is tricky to carry out a detailed soundness argument.

For a real world language like Javascript, merely defining the semantics is a major undertaking. For purposes of specification, it is desirable to use a VM monitor which provides modularity and relative ease of proving transparency. Our hope is that the technical devices we have used in our inlining and its correctness proof will facilitate development of provably secure inlined monitors for richer languages.

We have chosen this particular language and monitor [RS] for two reasons: it is flow-sensitive and it is provably sound. We believe that flow sensitivity is important for practical applications. The key reason is the demand for compatibility with legacy software; we don't require secure software to be rewritten in a new language or augmented with security annotations which are required by flow-insensitive approaches [HS06]. However, as discussed in [RS], these benefits come with strings attached: the requirement of static analysis of the branches-not-taken. This static analysis is represented by the $mod()$ function. Although the definition of this function is quite simple for the given language, it is quite hard to give adequate definitions for practical dynamic languages, as discussed below.

Dynamically allocated objects, such as in JavaScript, can be handled in the following way. For each field we create a "shadow" field that stores its security level. These fields are maintained in the same way as the "shadow" variables described in this work. The fact that fields and their "shadows" are stored in the same object eliminates problems with aliasing. Special caution should be taken if fields could be added at run-time and if the user can enumerate all the fields of an object. Possible solutions include renaming the "shadow" fields if a field with the same name is added and rewriting the enumeration operations in such a way that they do not access the "shadow" fields. The problem with precise and flow-sensitive treatment of objects lies in the fact that it is generally impossible to get a precise set of all the field locations modified by the program in the presence of aliasing. This requires approximations which could be quite precise, but are

$$
\begin{array}{ll}
 & \langle x := \tilde{e} \sqcup head(G); \textbf{while } e \textbf{ do } (\checkmark; \tilde{c_1}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(mod(c_1), S, x); \succ; \checkmark,\ m \uplus mst\rangle \\
\xrightarrow{\alpha_1} & \quad \text{where } \alpha_1 = a(x, \tilde{e} \sqcup head(G)) \text{ and we use } m(e) \neq 0 \\
 & \langle \textbf{while } e \textbf{ do } (\checkmark; \tilde{c_1}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(mod(c_1), S, x); \succ; \checkmark, \\
 & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))]\rangle \\
\xrightarrow{\alpha_2} & \quad \text{where } \alpha_2 = b(e, \textbf{skip}) \\
 & \langle \checkmark; \tilde{c_1}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x; \succ; \textbf{while } e \textbf{ do } (\checkmark; \tilde{c_1}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(mod(c_1), S, x); \succ; \checkmark, \\
 & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))]\rangle \\
\xrightarrow{t} & \\
 & \langle \tilde{c_1}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x; \succ; \textbf{while } e \textbf{ do } (\checkmark; \tilde{c_1}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(mod(c_1), S, x); \succ; \checkmark, \\
 & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))]\rangle \\
= & \\
 & \langle \tilde{c}',\ m \uplus mst'\rangle \qquad \text{define } \tilde{c}', mst'
\end{array}
$$

Figure 11. Trace of a transformation of **while** e **do** c_1

computationally expensive. It is a big question to what degree can aliasing information be approximated without causing too many false positives in the existing software.

Dynamic code evaluation is another tricky but necessary feature. It is curious that dynamic information flow monitoring was in no small part motivated by it [AS09]. At first glance it does not seem a hard task to handle. We can rewrite any occurrence of **eval()** in such a way that its argument is first transformed according to our inlining algorithm. We can also insert guards that make sure that the evaluated code does not access the sensitive monitor state. But as with objects, the specification of $mod()$ is tricky. In general, the variables that might be updated by an **eval()** is the set of all the variables in the current scope. Clearly, such judgment will cause a lot of false positives. More precise alternatives would require analysis of the current run-time context, e.g. the set of strings that might be passed to an **eval()** as arguments, but this is an expensive analysis to perform.

We can see that our approach faces the same problems and trade-offs as the VM monitor. We believe that if there is a VM monitor that could handle these features, we can design an inlined version with the same soundness guarantees.

Austin and Flanagan in [AF09] take a completely different approach to these problems. They propose to get rid of the requirement to do static analysis by trading some amount of flow-sensitivity. In short, their approach, "no sensitive upgrade", prohibits assignments to low variables in high context. Clearly, this could be implemented by inlining using our approach. In short, we will need to modify the rules for conditionals and loops and exclude the compensation for the branches-not-taken. We can also exclude the list I from the transformation environment. Finally, a change should be made to the assignment rule that would act as a security violation, halting execution if the left hand side has a level which is lower that the current control context. While Austin and Flanagan have proved that this approach is sound, it is not clear to us whether it will produce low false positive rate on the existing applications. It is possible that an approach combining both the static analysis and no-sensitive-upgrade would be the most viable. We are currently investigating these questions through experimentation.

References

[ADG08] Irem Aktug, Mads Dam, and Dilian Gurov. Provably correct runtime monitoring. In *International Symposium on Formal Methods*, volume 5014 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2008.

[AF09] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 113–124, New York, NY, USA, 2009. ACM.

[AHSS08] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS '08: Proceedings of the 13th European Symposium on Research in Computer Security*, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag.

[AS09] Aslan Askarov and Andrei Sabelfeld. Tight Enforcement of Information-Release Policies for Dynamic Languages. In *proceedings of*

22nd IEEE Computer Security Foundations Symposium. IEEE Computer Society Press, 2009.

[CMJL09] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, page XXX, New York, NY, USA, 2009. ACM.

[Coh78] Ellis S. Cohen. Information transmission in sequential programs. In Anita K. Jones Richard A. DeMillo, David P. Dobkin and Richard J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[CS08] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *IEEE Computer Security Foundations Symposium*, pages 51–65. IEEE Computer Society, 2008.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[DG09] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in javascript-based browser extensions. In *ACSAC'09: Proceedings of the 25th Annual Computer Security Applications Conference*, December 2009.

[DJLP09] Mads Dam, Bart Jacobs, Andreas Lundblad, and Frank Piessens. Security monitor inlining for multithreaded java. In *Genoa: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, pages 546–569, 2009.

[ES99] Ulfar Erlingsson and Fred B. Schneider. Sasi enforcement of security policies: A retrospective. In *In Proceedings of the New Security Paradigms Workshop*, pages 87–95. ACM Press, 1999.

[ES00] Ulfar Erlingsson and Fred B. Schneider. Irm enforcement of java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

[GBJS06] Gurvan Le Guernic, Anindya Banerjee, Thomas P. Jensen, and David A. Schmidt. Automata-based confidentiality monitoring. In Mitsu Okada and Ichiro Satoh, editors, *ASIAN*, volume 4435 of *Lecture Notes in Computer Science*, pages 75–89. Springer, 2006.

[GES+09] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI*, pages 465–478, 2009.

[GM82] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[Goo] Google. V8 project page. http://code.google.com/p/v8/.

[Gue07] Gurvan Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *IEEE Computer Security Foundations Symposium*, pages 218–232. IEEE Computer Society, 2007.

[HMS06a] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Certified in-lined reference monitoring on .net. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 7–16, New York, NY, USA, 2006. ACM.

[HMS06b] Kevin W. Hamlen, J. Gregory Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, 2006.

[HS06] Sebastian Hunt and David Sands. On flow-sensitive security types. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 79–90, New York, NY, USA, 2006. ACM.

[KHHJ08] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *ICISS*, volume 5352 of *LNCS*, pages 56–70, 2008.

[McL94] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *IEEE Symposium on Security and Privacy*, pages 79–93, 1994.

[NJK+07] Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *In Proceeding of the Network and Distributed System Security Symposium (NDSS'07*, 2007.

[NSCT08] Srijith K. Nair, Patrick N.D. Simpson, Bruno Crispo, and Andrew S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science*, 197(1):3–16, February 2008.

[Ric06] Jeffrey Richter. *CLR Via C#*. Microsoft Press, 2006.

[RS] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. Static Flow-Sensitive Security Analysis. Manuscript, January 2010.

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[SH10] Meera Sridhar and Kevin W. Hamlen. Model-Checking In-Lined Reference Monitors. In *Proceedings of 1th International Conference on Verification, Model Checking, and Abstract Interpretation*, 17 January 2010.

[SST08] Paritosh Shroff, Scott F. Smith, and Mark Thober. Securing information flow via dynamic capture of dependencies. *J. Comput. Secur.*, 16(5):637–688, 2008. Preliminary version appeared in CSF 2007.

[Sun] Sun Microsystems. The Java HotSpot Performance Engine Architecture. http://java.sun.com/products/hotspot/whitepaper.html.

[VP08] Dries Vanoverberghe and Frank Piessens. A caller-side inline reference monitor for an object-oriented intermediate language. In *Formal Methods for Open Object-Based Distributed Systems*, volume 5051 of *Lecture Notes in Computer Science*, pages 240–258. Springer, 2008.

[VXDS06] V. N. Venkatakrishnan, Wei Xu, Daniel C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In *Information and Communications Security (ICICS)*, volume 4307 of *Lecture Notes in Computer Science*, pages 332–351. Springer, 2006.

[YCIS07] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 237–249, New York, NY, USA, 2007. ACM.

[ZBWKM06] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th SOSP*, pages 263–278, 2006.

[Zda02] Stephan A. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, 2002.

[ZJS+09] Yu Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. Privacy scope: A precise information flow tracking system for finding application leaks. Technical Report UCB/EECS-2009-145, EECS Department, University of California, Berkeley, Oct 2009.

Appendix A.
Proofs

Where appropriate we treat Δ as a function from natural numbers, indicating positions, to security levels, indexing from 1.

Definition 5 (Multi-step with at most one output)

$$\langle \tilde{c},\ m \uplus mst\rangle \stackrel{\alpha}{\Longrightarrow} \langle \tilde{c}',\ m' \uplus mst'\rangle$$
$$\text{iff}$$
$$\langle \tilde{c},\ m \uplus mst\rangle \stackrel{\epsilon}{\longrightarrow}^{*} \stackrel{\alpha}{\longrightarrow} \stackrel{\epsilon}{\longrightarrow}^{*} \langle \tilde{c}',\ m' \uplus mst'\rangle$$

The following makes more precise a notion used in Def. 4.

Definition 6 (VM events and event relation) We let σ range over the "events" emitted by the VM monitor in Fig. 4, i.e., those of the form ω and the absence of an event. Then define $\alpha \triangleright \sigma$ iff either σ is nothing and α is in category ϵ (of Fig. 2) or $\alpha = o_l(e, v)$ and $\sigma = o_l(v)$.

Definition 7 (Join of partial functions) Suppose m_1, m_2 are partial functions $A \rightharpoonup L$ where L is a lattice with a join operation $\sqcup$. Then a join of m_1, m_2 is $m_1 \sqcup m_2 = m$ such that

$$m(a) = \begin{cases} m_1(a) \sqcup m_2(a) & a \in dom(m_1) \cap dom(m_2) \\ m_1(a) & a \in dom(m_1) - dom(m_2) \\ m_2(a) & a \in dom(m_2) - dom(m_1) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Lemma 1 For all Γ, Δ, G, S, I, e, $\tilde{e}$, if $\langle \Gamma,\ \Delta\rangle \cong (G, S, I, mst)$ and $S \vdash e \blacktriangleright \tilde{e}$ then $\Gamma(e) = mst(\tilde{e})$

Proof: Structural induction on e:

Case v :

$$\begin{array}{lll} & \Gamma(v) & \\ = & \bot & \text{definition of } \Gamma \\ = & mst(\bot) & \text{definition of } mst \\ = & mst(\tilde{v}) & \text{[TR-Value]} \\ = & mst(\tilde{e}) & \end{array}$$

Case x :

$$\begin{array}{lll} & \Gamma(x) & \\ = & mst(S(x)) & \text{definition of } \cong \\ = & mst(\tilde{x}) & \text{[TR-Var]} \\ = & mst(\tilde{e}) & \end{array}$$

Induction hypothesis : Suppose $\forall i \in (1..2) \mid S \vdash e_i \blacktriangleright \tilde{e}_i \wedge \Gamma(e_i) = mst(\tilde{e}_i)$

Case $e_1 \oplus e_2$:

$$\begin{array}{lll} & \Gamma(e_1 \oplus e_2) & \\ = & \Gamma(e_1) \sqcup \Gamma(e_2) & \text{definition of } \Gamma \\ = & mst(\tilde{e}_1) \sqcup mst(\tilde{e}_2) & \text{induction hypothesis} \\ = & mst(\tilde{e}_1 \sqcup \tilde{e}_2) & \text{definition of } mst \\ = & mst(\tilde{e}) & \text{[TR-Op]} \end{array}$$

End of structural induction.

□

Lemma 2 $\forall m,\ mst,\ S,\ G,\ I,\ cfgm,\ x \mid dom(m) \cap dom(mst) = \emptyset,\ cfgm \cong (G, S, I, mst),\ dom(m) \subseteq dom(S),\ x \in dom(m)$.

a $(m \uplus mst)[x \mapsto v] = m[x \mapsto v] \uplus mst$

b $(m \uplus mst)[S(x) \mapsto v] = m \uplus mst[S(x) \mapsto v]$

Proof: The proof is straightforward. □

Lemma 3 For all $G,\ S,\ I,\ \Gamma,\ \Delta,\ m$ and mst such that $\langle \Gamma, \Delta \rangle \cong (G, S, I, mst)$ (**1**) it holds that $\langle \kappa(head(I), S, head(G)),\ m \uplus mst \rangle \xRightarrow{\epsilon} \langle \square,\ m \uplus mst' \rangle$ where $mst' = mst \sqcup (mkFun(head(\Delta)) \circ S^{-1})$

Proof: We prove by unrolling the execution of the result of $\kappa(head(I), S, head(G))$ and using lemma 2 and the definition $\kappa(i, S, x)$. Induction on size of $head(I)$:

Case $head(I) = \emptyset$:

$$
\begin{array}{lll}
 & \langle \kappa(head(I), S, head(G)),\ m \uplus mst \rangle & \\
= & \langle \textbf{skip},\ m \uplus mst \rangle & \text{by definition of } \kappa \\
\xrightarrow{\epsilon} & \langle \square,\ m \uplus mst \rangle & \text{semantics} \\
= & \langle \square,\ m \uplus mst' \rangle & \text{found } mst' \text{ (\textbf{2})} \\
 & mst' & \\
= & mst & \text{by (2)} \\
= & mst \sqcup \emptyset & \emptyset \text{ is an identity for } \sqcup \\
= & mst \sqcup (mkFun(head(\Delta)) \circ S^{-1}) & \text{by (1)}
\end{array}
$$

Case $head(I) = \{x\} \cup i$:

$$
\begin{array}{ll}
 & \langle \kappa(head(I), S, head(G)),\ m \uplus mst \rangle \\
= & \quad \text{definition of } \kappa \\
 & \langle S(x) \textbf{:=} head(G) \sqcup S(x); \kappa(i, S, head(G)),\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \quad \text{semantics} \\
 & \langle \kappa(i, S, head(G)),\ m \uplus mst[S(x) \mapsto mst(head(G)) \sqcup mst(S(x))] \rangle \\
\xRightarrow{\epsilon} & \quad \text{induction hypothesis} \\
 & \langle \square,\ m \uplus mst[S(x) \mapsto mst(head(G)) \sqcup mst(S(x))] \sqcup ((mkFun(head(\Delta)) \setminus [x \mapsto lev(\Delta)]) \circ S^{-1}) \rangle \\
= & \quad \text{found } mst' \text{ (\textbf{3})} \\
 & \langle \square,\ m \uplus mst' \rangle
\end{array}
$$

$$
\begin{array}{ll}
 & mst' \\
= & \quad (3) \\
 & mst[S(x) \mapsto mst(head(G)) \sqcup mst(S(x))] \sqcup ((mkFun(head(\Delta)) \setminus [x \mapsto lev(\Delta)]) \circ S^{-1}) \\
= & \\
 & mst \sqcup [S(x) \mapsto mst(head(G))] \sqcup ((mkFun(head(\Delta)) \setminus [x \mapsto lev(\Delta)]) \circ S^{-1}) \\
= & \quad \text{induction hypothesis, definition of update}_l(c),\ lev(\Delta) \\
 & mst \sqcup (([y \mapsto mst(head(G))] \sqcup \{[y \mapsto l] \mid s \in fst(head(\Delta)) \setminus \{x\}, l = snd(head(\Delta))\}) \circ S^{-1}) \\
= & \quad \text{definition of } \cong \\
 & mst \sqcup (\{[y \mapsto l] \mid y \in fst(head(\Delta)), l = snd(head(\Delta))\} \circ S^{-1}) \\
= & \quad \text{by (1), definition of } \cong \\
 & mst \sqcup (mkFun(head(\Delta)) \circ S^{-1})
\end{array}
$$

End of induction.

□

Lemma 4 For all c, $\tilde{c}$, m, mst, Γ, Δ, c', m', Γ' and Δ' such that $\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle \sim \langle\tilde{c},\, m \uplus mst\rangle$: if $\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle \xrightarrow{\sigma} \langle\langle c',\, m'\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle$ then there exist such $\tilde{c}', mst'$ and α that $\langle\tilde{c},\, m \uplus mst\rangle \overset{\alpha}{\Longrightarrow}\xrightarrow{t} \langle\tilde{c}',\, m' \uplus mst'\rangle$, and $\langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle \sim \langle\tilde{c}',\, m' \uplus mst'\rangle$, and $\alpha \triangleright \sigma$.

Proof:

Rule induction on $\longrightarrow$:

Case $c = \textbf{skip}$: We show one step of $\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle$ in order to determine c', m', σ, Γ' and Δ.

$$\begin{array}{rll} \longrightarrow & \langle\langle \textbf{skip},\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle & \sigma \text{ is empty} \\ = & \langle\langle \square,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle & \text{found } c',\, m',\, \Gamma',\, \Delta' \\ & \langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle & \end{array}$$

The transformed command $\tilde{c} = \textbf{skip;}\checkmark$ according to [TR-Skip].
To prove that there exist such $\tilde{c}', mst'$

$$\langle\tilde{c},\, m \uplus mst\rangle \overset{\alpha}{\Longrightarrow}\xrightarrow{t} \langle\tilde{c}',\, m' \uplus mst'\rangle$$

and

$$\langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle \sim \langle\tilde{c}',\, m' \uplus mst'\rangle$$

such that $\alpha \triangleright \sigma$ where σ is empty we unroll the execution of $\langle\tilde{c},\, m \uplus mst\rangle$, applying language semantic rules for $c\textbf{;}\, c$, **skip** and $\checkmark$:

$$\begin{array}{rl} & \langle \textbf{skip;}\checkmark,\, m \uplus mst\rangle \\ \xrightarrow{\epsilon} & \quad \alpha = \epsilon \text{ which agrees with } \alpha \triangleright \sigma. \\ & \langle\checkmark,\, m \uplus mst\rangle \\ \xrightarrow{t} & \\ & \langle\square,\, m \uplus mst\rangle \\ = & \quad \text{found } \tilde{c}',\, mst' \\ & \langle\tilde{c}',\, m' \uplus mst'\rangle \end{array}$$

Next, we need to prove that $\langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle \sim \langle\tilde{c}',\, m' \uplus mst'\rangle$.
To do that we need to prove that there exist such G', S', I' such that $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ and $\langle\Gamma',\, \Delta'\rangle \cong (G', S', I', mst')$.
It is obvious that $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ according to [TRi-stop] independent of the values of G', S', I'.
We show that $\langle\Gamma',\, \Delta'\rangle \cong (G', S', I', mst')$ by substituting G' with G, S' with S, I' with I, mst' with mst, Γ' with Γ and Δ' with Δ and by the hypothesis of the lemma.

Case $c = \succ$: We show a step of the VM-monitored configuration $\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle$:

$$\begin{array}{rll} \longrightarrow & \langle\langle \succ,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle & \\ & \langle\langle \square,\, m\rangle,\, \langle\Gamma \sqcup mkFun(head(\Delta)),\, tail(\Delta)\rangle\rangle & \sigma \text{ is empty} \\ = & \langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle & \text{found } c',\, m',\, \Gamma' \textbf{ (1)},\, \Delta' \textbf{ (2)} \end{array}$$

$\tilde{c} = \kappa(head(I), S, head(G))\textbf{;}\, \succ\textbf{;}\checkmark$ according to [TR-End].
First, we unroll the execution of $\langle\tilde{c},\, m \uplus mst\rangle$:

$$\begin{array}{rll} \overset{\epsilon}{\Longrightarrow} & \langle\kappa(head(I), S, head(G))\textbf{;}\, \succ\textbf{;}\checkmark,\, m \uplus mst\rangle & \\ & \langle\succ\textbf{;}\checkmark,\, m \uplus (mst \sqcup (mkFun(head(\Delta)) \circ S^{-1}))\rangle & \text{lemma 3, semantics} \\ \xrightarrow{\epsilon} & \langle\checkmark,\, m \uplus (mst \sqcup (mkFun(head(\Delta)) \circ S^{-1}))\rangle & \\ \xrightarrow{t} & \langle\square,\, m \uplus (mst \sqcup (mkFun(head(\Delta)) \circ S^{-1}))\rangle & \\ = & \langle\tilde{c}',\, m' \uplus mst'\rangle & \text{found } \tilde{c}',\, mst' \textbf{ (3)} \end{array}$$

Next, we need to prove that $\langle\langle c',\ m'\rangle,\ \langle\Gamma',\ \Delta'\rangle\rangle \sim \langle\tilde{c}',\ m' \uplus mst'\rangle$.
To do that we need to prove that there exist such G', S', I' such that $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ and $\langle\Gamma',\ \Delta'\rangle \cong (G', S', I', mst')$. It is obvious that $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ according to [TR-Stop] independent of the values of G', S', I'.
Finally, we show that for $S' = S$ (**4**), $G' = tail(G)$ (**5**) and $I' = tail(I)$ (**6**) $\langle\Gamma',\ \Delta'\rangle \cong (G', S', I', mst')$:

1) we show that $\Gamma' = mst' \circ S'$:

$$
\begin{array}{lll}
 & \Gamma' & \\
= & \Gamma \sqcup mkFun(head(\Delta)) & \text{by (1)} \\
= & (mst \circ S) \sqcup mkFun(head(\Delta)) & \text{by the hypothesis of the lemma} \\
= & (mst \circ S) \sqcup ((mkFun(head(\Delta)) \circ S^{-1}) \circ S) & S^{-1} \circ S = Id,\ S \text{ is injective} \\
= & (mst \sqcup (mkFun(head(\Delta)) \circ S^{-1})) \circ S & \text{distribute composition over union} \\
= & mst' \circ S' & \text{by (3) and (4)}
\end{array}
$$

2) we show that $\Delta' = zip(I', mst' \circ G')$:

$$
\begin{array}{lll}
 & \Delta' & \\
= & tail(\Delta) & \text{(2)} \\
= & tail(zip(I, mst \circ G)) & \text{hypothesis of the lemma} \\
= & tail(((head(I), head(mst \circ G)) :: zip(tail(I))), tail(mst \circ G)) & \text{definition of } zip \\
= & zip(tail(I), tail(mst \circ G)) & \text{definition of } tail \\
= & zip(tail(I), tail((mst \sqcup (mkFun(head(\Delta)) \circ S^{-1})) \circ G)) & dom(S^{-1}) \cap rng(G) = \emptyset \\
= & zip(tail(I), (mst \sqcup (mkFun(head(\Delta)) \circ S^{-1})) \circ tail(G)) & \\
= & zip(I', mst' \circ G') & \text{(6), (5), (3)}
\end{array}
$$

Case $c = \square$: The original configuration could not make progress according to language semantics. This does not fulfill one of our assumptions. This case is, thus, unrealizable.

Case $c = \checkmark$: This case is unrealizable because $\checkmark$ is not part of the user language and does not appear in any intermediate configuration of user programs. It could only be introduced by the instrumented inlining procedure.

Case $c = x \texttt{ := } e$: We show a step of the VM-monitored execution:

$$
\begin{array}{lll}
 & \langle\langle x \texttt{ := } e,\ m\rangle,\ \langle\Gamma,\ \Delta\rangle\rangle & \\
\longrightarrow & \langle\langle\square,\ m[x \mapsto m(e)]\rangle,\ \langle\Gamma[x \mapsto lev(\Delta) \sqcup \Gamma(e)],\ \Delta\rangle\rangle & \\
= & \langle\langle c',\ m'\rangle,\ \langle\Gamma',\ \Delta'\rangle\rangle & \text{found } c',\ m',\ \Gamma' \text{ (7) and } \Delta' = \Delta \text{ (8)}
\end{array}
$$

$\tilde{c} = S(x) \texttt{ := } \tilde{e} \sqcup g\texttt{; } x \texttt{ := } e\texttt{; } \checkmark \wedge S \vdash e \blacktriangleright \tilde{e}$ according to [TR-assign].
First we unroll the execution of $\langle\tilde{c},\ m \uplus mst\rangle$, applying language semantic rules for $c\texttt{; } c$, $x \texttt{ := } e$ and $\checkmark$ appropriately:

$$
\begin{array}{lll}
 & \langle S(x) \texttt{ := } \tilde{e} \sqcup g\texttt{; } x \texttt{ := } e\texttt{; } \checkmark,\ m \uplus mst\rangle & \\
\xrightarrow{\epsilon} & \langle x \texttt{ := } e\texttt{; } \checkmark,\ m \uplus mst[S(x) \mapsto mst(\tilde{e}) \sqcup g]\rangle & \text{semantics, where } g = head(G), \text{ Lemma 2} \\
\xrightarrow{\epsilon} & \langle\checkmark,\ m[x \mapsto m(e)] \uplus mst[S(x) \mapsto mst(\tilde{e}) \sqcup g]\rangle & \text{semantics, Lemma 2} \\
\xrightarrow{t} & \langle\square,\ m[x \mapsto m(e)] \uplus mst[S(x) \mapsto mst(\tilde{e}) \sqcup g]\rangle & \\
= & \langle\tilde{c}',\ m' \uplus mst'\rangle & \text{found } \tilde{c}',\ mst'
\end{array}
$$

By looking at the last configuration in the execution we can see that:

- $\tilde{c}' = \square$
- $mst' = mst[S(x) \mapsto mst(\tilde{e}) \sqcup head(G)]$ (**9**)

For any G', S', I' we obtain a valid instance of [TR-Stop]: $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$.
Next, we show that for $G' = G$ (**10**), $S' = S$ (**11**), $I' = I$ (**12**) we get $\langle \Gamma',\ \Delta' \rangle \cong (G', S', I', mst')$. We need to prove that $\Gamma' = mst' \circ S'$ and $\Delta' = zip(I', mst' \circ G')$.

- we prove that $\Gamma' = mst' \circ S'$, i.e. $\forall z \in dom(\Gamma').\Gamma'(z) = mst'(S(z))$: Case distinction on z:
  Case $z = x$:

$$\begin{array}{ll} & \Gamma'(z) \\ = & \Gamma[x \mapsto lev(\Delta) \sqcup \Gamma(e)](x) \\ = & \Gamma(e) \sqcup lev(\Delta) \\ = & mst(\tilde{e}) \sqcup lev(\Delta) \\ = & mst[S(x) \mapsto mst(\tilde{e}) \sqcup head(G)](S(x)) \\ = & mst'(S(z)) \end{array} \quad \begin{array}{l} (10) \\ \text{definition of mapping update} \\ \text{Lemma 1, assumption} \\ \text{definition of mapping update} \\ (9) \end{array}$$

  Case $z \neq x$:

$$\begin{array}{ll} & \Gamma'(z) \\ = & \Gamma[x \mapsto \Gamma(e) \sqcup lev(\Delta)](z) \\ = & \Gamma(z) \sqcup lev(\Delta) \\ = & mst(z) \sqcup head(G) \\ = & mst[S(x) \mapsto mst(\tilde{e}) \sqcup head(G)](S(z)) \\ = & mst'(S(z)) \end{array} \quad \begin{array}{l} \text{definition of mapping update} \\ \text{Lemma 1, assumption} \\ \text{definition of mapping update} \\ (9) \end{array}$$

  End of case distinction.

- we prove that $\Delta' = zip(I', mst' \circ G')$:

$$\begin{array}{ll} & \Delta' \\ = & \Delta \\ = & zip(I, mst \circ G) \\ = & zip(I, (mst \sqcup [S(x) \mapsto head(G)]) \circ G) \\ = & zip(I, mst[S(x) \mapsto mst(\tilde{e}) \sqcup head(G)] \circ G) \\ = & zip(I', mst' \circ G') \end{array} \quad \begin{array}{l} (8) \\ \\ dom(S) \cap rng(G) = \emptyset \\ \\ (9),\ (10),\ (12) \end{array}$$

Case $c =$ **if** e **then** c_1 **else** c_2 : Without limiting the generality, we can assume that $m(e) \neq 0$. The proof for the case when $m(e) = 0$ is similar.
We show the step of the VM-monitored configuration:

$$\begin{array}{ll} & \langle\langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2,\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle \\ \longrightarrow & \\ = & \langle\langle c_1\textbf{;}\succ,\ m\rangle,\ \langle \Gamma,\ (\Gamma(e) \sqcup lev(\Delta) :: \Delta)\rangle\rangle \\ & \langle\langle c',\ m'\rangle,\ \langle \Gamma',\ \Delta'\rangle\rangle \end{array} \quad \text{found } c',\ m',\ \Gamma' \textbf{ (13)},\ \Delta' \textbf{ (14)}$$

$\tilde{c} = x \textbf{:=} \tilde{e} \sqcup head(G)\textbf{; if } e \textbf{ then } \tilde{c}'_1 \textbf{ else } \tilde{c}'_2\textbf{;} \checkmark$ and $S \vdash e \blacktriangleright \tilde{e}$ where $x \notin \text{dom}(S) \cup \text{rng}(S) \wedge x \notin G$ and $(x :: G), S, (mod(c_{i \bmod 2+1}) :: I) \vdash c_i \blacktriangleright \tilde{c}_i$ (**15**) and $\tilde{c}'_i = \checkmark\textbf{;}\, \tilde{c}_i\textbf{;}\, \kappa(mod(c_{(i \bmod 2)+1}), S, x)$ according to [TR-if].

First, we unroll the execution of $\langle \tilde{c},\ m \uplus mst \rangle$:

$$
\begin{array}{ll}
 & \langle x := \tilde{e} \sqcup head(G); \textbf{if } e \textbf{ then } \tilde{c}_1' \textbf{ else } \tilde{c}_2'; \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \textbf{if } e \textbf{ then } \tilde{c}_1' \textbf{ else } \tilde{c}_2'; \checkmark,\ m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
\xrightarrow{\epsilon} & \quad \text{semantics, assumption } m(e) \neq 0, \text{ substitution for } \tilde{c}_1' \\
 & \langle \checkmark; \tilde{c}_1; \kappa(mod(c_2), S, x); \succ; \checkmark,\ m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
\xrightarrow{t} & \\
 & \langle \tilde{c}_1; \kappa(mod(c_2), S, x); \succ; \checkmark,\ m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
= & \quad \text{found } \tilde{c}',\ mst' \textbf{ (16)} \\
 & \langle \tilde{c}',\ m' \uplus mst' \rangle
\end{array}
$$

Next, we show that $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ for $G' = (x :: G)$ **(17)**, $S' = S$ **(18)** and $I' = (mod(c_2) :: I)$ **(19)**.

$$
\text{TR-SEQ}\ \frac{\dfrac{\text{(15)}}{(x :: G), S, (mod(c_2) :: I) \vdash c_1 \blacktriangleright \tilde{c}_1} \quad (x :: G), S, (mod(c_2) :: I) \vdash \succ \blacktriangleright \kappa(mod(c_2), S, x); \succ; \checkmark\ \text{TR-END}}{G' = (x :: G), S, (mod(c_2) :: I) \vdash c_1; \succ \blacktriangleright \tilde{c}_1; \kappa(mod(c_2), S, x); \succ; \checkmark}
$$

Last, we show that $\langle \Gamma',\ \Delta' \rangle \cong (G', S', I', mst')$ for the given $G',\ S',\ I'$.

1) we prove that $\Gamma' = mst' \circ S'$

$$
\begin{array}{ll}
 & \Gamma' \\
= & \quad \text{by (13)} \\
 & \Gamma \\
= & \quad \text{by the hypothesis of the lemma} \\
 & mst \circ S \\
= & \quad \text{since } x \notin dom(S) \\
 & mst \circ S \sqcup [x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \circ S \\
= & \quad \text{distribute function composition over union} \\
 & mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \circ S \\
= & \quad \text{by (16),(18)} \\
 & mst' \circ S'
\end{array}
$$

2) we prove that $\Delta' = zip(I', mst' \circ G')$:

$$
\begin{array}{ll}
 & \Delta' \\
= & \quad (14) \\
 & (\text{update}_{\Gamma(e) \sqcup lev(\Delta)}(c_2) :: \Delta) \\
= & \quad \text{definition of update}_l(c) \\
 & ((mod(c_2), \Gamma(e) \sqcup lev(\Delta)) :: \Delta) \\
= & \quad \text{definition of } zip\text{, the hypothesis of the lemma, definition of } lev \\
 & zip((mod(c_2) :: I), (\Gamma(e) \sqcup lev(\Delta)) :: mst \circ G)) \\
= & \quad \text{lemma 1} \\
 & zip((mod(c_2) :: I), (mst(\tilde{e}) \sqcup mst(head(G)) :: mst \circ G)) \\
= & \\
 & zip((mod(c_2) :: I), (mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))](x) :: mst \circ G)) \\
= & \quad \text{distribute composition over list construction, } x \notin rng(G) \\
 & zip((mod(c_2) :: I), mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \circ (x :: G)) \\
= & \quad (19), (16), (17) \\
 & zip(I', mst' \circ G')
\end{array}
$$

Case $c = \textbf{while } e \textbf{ do } c_1$: Case distinction on $m(e)$:

Case $m(e) \neq 0$: We unroll the execution of $\langle\langle c,\ m\rangle,\ \langle\Gamma,\ \Delta\rangle\rangle$:

$$
\begin{array}{lll}
 & \langle\langle \textbf{while } e \textbf{ do } c_1,\ m\rangle,\ \langle\Gamma,\ \Delta\rangle\rangle & \\
\longrightarrow & \langle\langle c_1 \textbf{;} \succ \textbf{; while } e \textbf{ do } c_1,\ m\rangle,\ \langle\Gamma,\ (\text{update}_{\Gamma(e) \sqcup lev(\Delta)}(\textbf{skip}) :: \Delta)\rangle\rangle & \text{semantics, } m(e) \neq 0 \\
= & \langle\langle c',\ m'\rangle,\ \langle\Gamma',\ \Delta'\rangle\rangle & \text{found } c',\ m',\ \Gamma' \textbf{ (20)} \text{ and } \Delta' \textbf{ (21)}
\end{array}
$$

Case distinction on $\tilde{c}$:

Case $x \textbf{ := } \tilde{e} \sqcup head(G) \textbf{; while } e \textbf{ do } (\checkmark \textbf{; } \tilde{c_1} \textbf{; skip; } \succ \textbf{; } \checkmark \textbf{; } x \textbf{ := } \tilde{e} \sqcup x) \textbf{; } \checkmark \textbf{; } \kappa(c_1, S, x) \textbf{; } \succ \textbf{; } \checkmark$:
according to [TR-While] where $S \vdash e \blacktriangleright \tilde{e}$, and $(x :: G), S, (\emptyset :: I) \vdash c_1 \blacktriangleright \tilde{c_1}$ and $x \notin \text{dom}(S) \cup \text{rng}(S) \wedge x \notin G$.
We unroll the execution of $\langle \tilde{c},\ m \uplus mst\rangle$:

$$
\begin{array}{ll}
 & \langle x \textbf{ := } \tilde{e} \sqcup head(G) \textbf{; while } e \textbf{ do } (\checkmark \textbf{; } \tilde{c_1} \textbf{; skip; } \succ \textbf{; } \checkmark \textbf{; } x \textbf{ := } \tilde{e} \sqcup x) \textbf{; } \checkmark \textbf{; } \kappa(mod(c_1), S, x) \textbf{; } \succ \textbf{; } \checkmark,\ m \uplus mst\rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \textbf{while } e \textbf{ do } (\checkmark \textbf{; } \tilde{c_1} \textbf{; skip; } \succ \textbf{; } \checkmark \textbf{; } x \textbf{ := } \tilde{e} \sqcup x) \textbf{; } \checkmark \textbf{; } \kappa(mod(c_1), S, x) \textbf{; } \succ \textbf{; } \checkmark, \\
 & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))]\rangle \\
\xrightarrow{\epsilon} & \quad \text{semantics, } m(e) \neq 0 \\
 & \langle \checkmark \textbf{; } \tilde{c_1} \textbf{; skip; } \succ \textbf{; } \checkmark \textbf{; } x \textbf{ := } \tilde{e} \sqcup x \textbf{; } \succ \textbf{; while } e \textbf{ do } (\checkmark \textbf{; } \tilde{c_1} \textbf{; skip; } \succ \textbf{; } \checkmark \textbf{; } x \textbf{ := } \tilde{e} \sqcup x) \textbf{; } \checkmark \textbf{; } \kappa(mod(c_1), S, x) \textbf{; } \succ \textbf{; } \checkmark, \\
 & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))]\rangle \\
\xrightarrow{t} & \\
 & \langle \tilde{c_1} \textbf{; skip; } \succ \textbf{; } \checkmark \textbf{; } x \textbf{ := } \tilde{e} \sqcup x \textbf{; } \succ \textbf{; while } e \textbf{ do } (\checkmark \textbf{; } \tilde{c_1} \textbf{; skip; } \succ \textbf{; } \checkmark \textbf{; } x \textbf{ := } \tilde{e} \sqcup x) \textbf{; } \checkmark \textbf{; } \kappa(mod(c_1), S, x) \textbf{; } \succ \textbf{; } \checkmark, \\
 & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))]\rangle \\
= & \quad \text{found } \tilde{c}',\ mst' \\
 & \langle \tilde{c}',\ m \uplus mst'\rangle
\end{array}
$$

Case $x \textbf{ := } \tilde{e} \sqcup x \textbf{; } \succ \textbf{; while } e \textbf{ do } (\checkmark \textbf{; } \tilde{c_1} \textbf{; skip; } \succ \textbf{; } \checkmark \textbf{; } x \textbf{ := } \tilde{e} \sqcup x) \textbf{; } \checkmark \textbf{; } \kappa(c_1, S, x) \textbf{; } \succ \textbf{; } \checkmark$:
according to [TR-WhileLoop], where $S \vdash e \blacktriangleright \tilde{e}$, and $G, S, I \vdash c_1 \blacktriangleright \tilde{c_1}$ **(22)**, and $head(G) = x$ and $head(I) = \emptyset$.

We unroll the execution of $\langle \tilde{c},\ m \uplus mst \rangle$:

$$
\begin{array}{ll}
 & \langle x := \tilde{e} \sqcup head(G);\ \succ;\ \textbf{while}\ e\ \textbf{do}\ (\checkmark;\ \tilde{c_1};\ \textbf{skip};\ \succ;\ \checkmark;\ x := \tilde{e} \sqcup head(G));\ \checkmark;\ \kappa(mod(c_1), S, x);\ \succ;\ \checkmark,\\
 & m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \succ;\ \textbf{while}\ e\ \textbf{do}\ (\checkmark;\ \tilde{c_1};\ \textbf{skip};\ \succ;\ \checkmark;\ x := \tilde{e} \sqcup head(G));\ \checkmark;\ \kappa(mod(c_1), S, x);\ \succ;\ \checkmark,\\
 & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \textbf{while}\ e\ \textbf{do}\ (\checkmark;\ \tilde{c_1};\ \textbf{skip};\ \succ;\ \checkmark;\ x := \tilde{e} \sqcup head(G));\ \checkmark;\ \kappa(mod(c_1), S, x);\ \succ;\ \checkmark,\\
 & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
\xrightarrow{\epsilon} & \quad \text{semantics, } m(e) \neq 0 \\
 & \langle \begin{array}{l} \checkmark;\ \tilde{c_1};\ \textbf{skip};\ \succ;\ \checkmark;\ x := \tilde{e} \sqcup head(G);\ \succ;\\ \textbf{while}\ e\ \textbf{do}\ (\checkmark;\ \tilde{c_1};\ \textbf{skip};\ \succ;\ \checkmark;\ x := \tilde{e} \sqcup head(G));\ \checkmark;\ \kappa(mod(c_1), S, x);\ \succ;\ \checkmark \end{array}\ ',\\
 & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
\xrightarrow{t} & \\
 & \langle \begin{array}{l} \tilde{c_1};\ \textbf{skip};\ \succ;\ \checkmark;\ x := \tilde{e} \sqcup head(G);\ \succ;\\ \textbf{while}\ e\ \textbf{do}\ (\checkmark;\ \tilde{c_1};\ \textbf{skip};\ \succ;\ \checkmark;\ x := \tilde{e} \sqcup head(G));\ \checkmark;\ \kappa(mod(c_1), S, x);\ \succ;\ \checkmark \end{array}\ ',\\
 & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
= & \quad \text{found } \tilde{c}',\ mst'\ \textbf{(23)} \\
 & \langle \tilde{c}',\ m \uplus mst' \rangle
\end{array}
$$

End of case distinction.

Next, we show that $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ is a valid instance of [TR-Seq] for $G = (x :: G)$, **(24)**, $S' = S$ **(25)** and $I' = (\emptyset :: I)$ **(26)**.

$$
\text{TR-Seq}\ \frac{\text{TR-EndSeq}\ \dfrac{\text{TR-WhileLoop}\ \dfrac{\dfrac{(22)}{(x :: G), S, (\emptyset :: I) \vdash c_1 \blacktriangleright \tilde{c_1}}}{\begin{array}{l} G, S, I \vdash \textbf{while}\ e\ \textbf{do}\ c_1 \blacktriangleright \\ synseq x := \tilde{e} \sqcup head(G) \succ;\ \textbf{while}\ e\ \textbf{do}\ (\checkmark;\ \tilde{c_1});\\ \checkmark;\ \kappa(mod(c_1), S, x);\ \succ;\ \checkmark \end{array}}}{\begin{array}{l} (x :: G), S, (\emptyset :: I) \vdash \succ;\ \textbf{while}\ e\ \textbf{do}\ c_1 \blacktriangleright \\ \textbf{skip};\ \succ;\ \checkmark;\ x := \tilde{e} \sqcup head(G);\ \succ;\ \textbf{while}\ e\ \textbf{do}\ (\checkmark;\ \tilde{c_1});\ \checkmark;\ \kappa(mod(c_1), S, x);\ \succ;\ \checkmark \end{array}}}{\begin{array}{l} (x :: G), S, (\emptyset :: I) \vdash c_1;\ \succ;\ \textbf{while}\ e\ \textbf{do}\ c_1 \blacktriangleright \\ \tilde{c_1};\ \textbf{skip};\ \succ;\ \checkmark;\ x := \tilde{e} \sqcup head(G);\ \succ;\ \textbf{while}\ e\ \textbf{do}\ (\checkmark;\ \tilde{c_1};\ \textbf{skip};\ \succ;\ \checkmark;\ x := \tilde{e} \sqcup head(G));\ \checkmark;\\ \kappa(mod(c_1), S, x);\ \succ;\ \checkmark \end{array}}
$$

Last, we show that $\langle \Gamma',\ \Delta' \rangle \cong (G', S', I', mst')$:

1) we prove that $\Gamma' = mst' \circ S'$

$$
\begin{array}{ll}
 & \Gamma' \\
= & \quad (20) \\
 & \Gamma \\
= & \quad \text{the hypothesis of the lemma} \\
 & mst(S(z)) \\
= & \quad \text{by definition of mapping update and because } x \notin dom(S) \\
 & (mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))])(S(z)) \\
= & \quad (23) \\
 & mst'(S(z))
\end{array}
$$

2) we prove that $\Delta' = zip(I', mst' \circ G')$:

$$
\begin{array}{ll}
& \Delta' \\
= & \quad \text{(21)} \\
& (\text{update}_{\Gamma(e) \sqcup lev(\Delta)}(\textbf{skip}) :: \Delta) \\
= & \quad \text{definition of } lev \\
& ((mod(\textbf{skip}), \Gamma(e) \sqcup head(\Delta)) :: \Delta) \\
= & \quad \text{lemma 1, the hypothesis of the lemma, definition of } mod() \\
& ((\emptyset, mst(\tilde{e}) \sqcup mst(head(G))) :: zip(I, mst \circ G)) \\
= & \quad \text{definition of } zip \\
& zip((\emptyset :: I), (mst(\tilde{e}) \sqcup mst(head(G)) :: mst \circ G)) \\
= & \quad x \notin rng(G) \\
& zip((\emptyset :: I), (mst[x \mapsto mst(\tilde{e}) mst(head(G))](x) :: mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \circ G)) \\
= & \quad \text{distribute composition over list contructor} \\
& zip((\emptyset :: I), mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \circ (x :: G)) \\
= & \quad \text{(24), (26), (23)} \\
& zip(I', mst' \circ G')
\end{array}
$$

Case $m(e) = 0$: We unroll the execution of $\langle\langle c,\ m\rangle,\ \langle\Gamma,\ \Delta\rangle\rangle$:

$$
\begin{array}{lll}
\longrightarrow & \langle\langle \textbf{while } e \textbf{ do } c_1,\ m\rangle,\ \langle\Gamma,\ \Delta\rangle\rangle & \text{semantics, } m(e) = 0 \\
= & \langle\langle \succ,\ m\rangle,\ \langle\Gamma,\ (\text{update}_{\Gamma(e) \sqcup lev(\Delta)}(c_1) :: \Delta)\rangle\rangle & \text{found } c',\ m',\ \Gamma' \textbf{ (27)},\ \Delta' \textbf{ (28)} \\
& \langle\langle c',\ m'\rangle,\ \langle\Gamma',\ \Delta'\rangle\rangle &
\end{array}
$$

Case distinction on $\tilde{c}$:

Case $x := \tilde{e} \sqcup head(G)$; **while** e **do** $(\checkmark; \tilde{c_1}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(c_1, S, x); \succ; \checkmark$:
according to [TR-While] where $S \vdash e \blacktriangleright \tilde{e}$, and $(x :: G), S, (\emptyset :: I) \vdash c_1 \blacktriangleright \tilde{c_1}$ **(29)**, and $x \notin \text{dom}(S) \cup \text{rng}(S) \wedge x \notin G$.
We unroll the execution of $\langle \tilde{c},\ m \uplus mst\rangle$:

$$
\begin{array}{ll}
& \langle x := \tilde{e} \sqcup head(G); \textbf{while } e \textbf{ do } (\checkmark; \tilde{c_1}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(c_1, S, x); \succ; \checkmark, \\
& m \uplus mst\rangle \\
\xrightarrow{\epsilon} & \\
& \langle \textbf{while } e \textbf{ do } (\checkmark; \tilde{c_1}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(c_1, S, x); \succ; \checkmark, \\
& m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))]\rangle \\
\xrightarrow{\epsilon} & \quad \text{semantics, } m(e) = 0 \\
& \langle \succ; \checkmark; \kappa(c_1, S, x); \succ; \checkmark,\ m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))]\rangle \\
\xrightarrow{\epsilon} & \\
& \langle \checkmark; \kappa(c_1, S, x); \succ; \checkmark,\ m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))]\rangle \\
\xrightarrow{t} & \\
& \langle \kappa(c_1, S, x); \succ; \checkmark,\ m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))]\rangle \\
= & \quad \text{found } \tilde{c}',\ mst' \textbf{ (30)} \\
& \langle \tilde{c}',\ m' \uplus mst'\rangle
\end{array}
$$

Case $x := \tilde{e} \sqcup x; \succ;$ **while** e **do** $(\checkmark; \tilde{c_1}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(c_1, S, x); \succ; \checkmark$: according to [TR-WhileLoop], where $S \vdash e \blacktriangleright \tilde{e}$, and $G, S, I \vdash c_1 \blacktriangleright \tilde{c_1}$ and $head(G) = x$ and $head(I) = \emptyset$.

We unroll the execution of $\langle \tilde{c},\ m \uplus mst \rangle$:

$$
\begin{array}{ll}
 & \langle x := \tilde{e} \sqcup x;\ \succ;\ \textbf{while}\ e\ \textbf{do}\ (\checkmark;\ \tilde{c_1};\ \textbf{skip};\ \succ;\ \checkmark;\ x := \tilde{e} \sqcup x);\ \checkmark;\ \kappa(c_1, S, x);\ \succ;\ \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \succ;\ \textbf{while}\ e\ \textbf{do}\ (\checkmark;\ \tilde{c_1};\ \textbf{skip};\ \succ;\ \checkmark;\ x := \tilde{e} \sqcup x);\ \checkmark;\ \kappa(c_1, S, x);\ \succ;\ \checkmark, \\
 & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \textbf{while}\ e\ \textbf{do}\ (\checkmark;\ \tilde{c_1};\ \textbf{skip};\ \succ;\ \checkmark;\ x := \tilde{e} \sqcup x);\ \checkmark;\ \kappa(c_1, S, x);\ \succ;\ \checkmark, \\
 & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
\xrightarrow{\epsilon} & \quad \text{semantics, } m(e) = 0 \\
 & \langle \succ;\ \checkmark;\ \kappa(c_1, S, x);\ \succ;\ \checkmark,\ m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \checkmark;\ \kappa(c_1, S, x);\ \succ;\ \checkmark,\ m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
\xrightarrow{t} & \\
 & \langle \kappa(c_1, S, x);\ \succ;\ \checkmark,\ m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
= & \quad \text{found } \tilde{c}' \text{ and } mst' \\
 & \langle \tilde{c},\ m' \uplus mst' \rangle
\end{array}
$$

End of case distinction.

We show that $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ for $G' = (x :: G)$ (**31**), $S' = S$ (**32**), $I' = (mod(c_1) :: I)$ (**33**) is a valid instance of [TR-End]:

$$(x :: G), S, (mod(c_1) :: I) \vdash \succ \blacktriangleright \kappa(mod(c_1), S, x);\ \succ;\ \checkmark$$

Last, we show that $\langle \Gamma',\ \Delta' \rangle \cong (G', S', I', mst')$:

1) we prove that $\Gamma' = mst' \circ S'$

$$
\begin{array}{ll}
 & \Gamma' \\
= & \quad (27) \\
 & \Gamma \\
= & \quad \text{the hypothesis of the lemma} \\
 & mst(S(z)) \\
= & \quad \text{by definition of mapping update and because } x \notin dom(S), (32) \\
 & mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \circ S \\
= & \quad \text{obtained earlier} \\
 & mst' \circ S'
\end{array}
$$

2) we prove that $\Delta' = zip(I', mst' \circ G')$:

$$
\begin{array}{ll}
 & \Delta' \\
= & \quad \text{(28)} \\
 & (\text{update}_{\Gamma(e)\sqcup lev(\Delta)}(c_1) :: \Delta) \\
= & \quad \text{definitions of update}_l(c) \text{ and } lev \\
 & ((mod(c_1), \Gamma(e) \sqcup head(\Delta)) :: \Delta) \\
= & \quad \text{lemma 1, the hypothesis of the lemma} \\
 & zip((mod(c_1) :: I), (mst(\tilde{e}) \sqcup mst(head(G)) :: mst \circ G)) \\
= & \quad x \notin rng(G) \\
 & zip((mod(c_1) :: I), (mst(\tilde{e}) \sqcup mst(head(G)) :: mst \circ G \sqcup [x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \circ G)) \\
= & \quad \text{distribute composition over union} \\
 & zip((mod(c_1) :: I), (mst(\tilde{e}) \sqcup mst(head(G)) :: mst \sqcup [x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \circ G)) \\
= & \quad \text{apply mapping lookup, rewrite update as union} \\
 & zip((mod(c_1) :: I), (mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))](x) :: mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \circ G)) \\
= & \quad \text{distribute composition over list construction} \\
 & zip((mod(c_1) :: I), mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \circ (x :: G)) \\
= & \quad \text{(33), (30), (31)} \\
 & zip(I', mst' \circ G')
\end{array}
$$

End of case distinction.

Case $c = \textbf{output}_l(e)$: Case distinction on the type of enforcement:

Case OutputFailstop : We unroll the execution of $\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle$ to find c', m', Γ', Δ':

$$
\begin{array}{lll}
 & \langle\langle \textbf{output}_l(e),\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle & \\
\xrightarrow{o_l(v)} & & \\
= & \langle\langle \Box,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle & \\
 & \langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle & \text{found } c',\, m',\, \Gamma',\, \Delta'
\end{array}
$$

The step would only be possible if $lev(\Delta) \sqcup \Gamma(e) \sqsubseteq l$ (**34**).
According to [TR-OutputFailstop] $\tilde{c} = \textbf{if } head(G) \sqcup \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else } (\textbf{diverge});\checkmark$ where $S \vdash e \blacktriangleright \tilde{e}$.
We unroll the execution of $\langle \tilde{c},\, m \uplus mst\rangle$:

$$
\begin{array}{ll}
 & \langle \textbf{if } head(G) \sqcup \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else } (\textbf{diverge});\checkmark,\, m \uplus mst\rangle \\
\xrightarrow{\epsilon} & \quad \text{(34), } \Gamma(e) = mst(\tilde{e}) \text{ – lemma 1, } mst(head(G)) = lev(\Delta) \\
 & \langle \textbf{output}_l(e);\, \succ;\, \checkmark,\, m \uplus mst\rangle \\
\xrightarrow{o_l(e,v)} & \\
 & \langle \succ;\, \checkmark,\, m \uplus mst\rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \checkmark,\, m \uplus mst\rangle \\
\xrightarrow{t} & \\
 & \langle \Box,\, m \uplus mst\rangle \\
= & \quad \text{found } \tilde{c}',\, mst' \\
 & \langle \tilde{c}',\, m' \uplus mst'\rangle
\end{array}
$$

Case OutputDefault : According to [TR-OutputDefault] $\tilde{c} = \textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D);\checkmark$ where $S \vdash e \blacktriangleright \tilde{e}$.

Case distinction on $\Gamma(e)$:

Case $\Gamma(e) \sqsubseteq l$: **(35)**

We unroll the execution of $\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle$:

$$\begin{array}{rll} & \langle\langle \textbf{output}_l(e),\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle & \\ \xrightarrow{o_l(v)} & & \\ & \langle\langle \square,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle & \\ = & & \\ & \langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle & \text{found } c',\, m',\, \Gamma',\, \Delta' \end{array}$$

We unroll the execution of $\langle \tilde{c},\, m \uplus mst\rangle$:

$$\begin{array}{rl} & \langle \textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D); \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{\epsilon} & \\ & \quad \text{(35), } \Gamma(e) = mst(\tilde{e}) \text{ – lemma 1} \\ & \langle \textbf{output}_l(e); \succ; \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{o_l(e,v)} & \\ & \langle \succ; \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{\epsilon} & \\ & \langle \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{t} & \\ & \langle \square,\, m \uplus mst\rangle \\ = & \quad \text{found } \tilde{c}',\, mst' \\ & \langle \tilde{c}',\, m' \uplus mst'\rangle \end{array}$$

Case $\Gamma(e) \not\sqsubseteq l$: **(36)**

We unroll the execution of $\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle$:

$$\begin{array}{rll} & \langle\langle \textbf{output}_l(e),\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle & \\ \xrightarrow{o_l(D)} & & \\ & \langle\langle \square,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle & \\ = & & \\ & \langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle & \text{found } c',\, m',\, \Gamma',\, \Delta' \end{array}$$

We unroll the execution of $\langle \tilde{c},\, m \uplus mst\rangle$:

$$\begin{array}{rl} & \langle \textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D); \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{\epsilon} & \\ & \quad \text{(36), } \Gamma(e) = mst(\tilde{e}) \text{ – lemma 1} \\ & \langle \textbf{output}_l(D); \succ; \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{o_l(D,D)} & \\ & \langle \succ; \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{\epsilon} & \\ & \langle \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{t} & \\ & \langle \square,\, m \uplus mst\rangle \\ = & \quad \text{found } \tilde{c}',\, mst' \\ & \langle \tilde{c}',\, m' \uplus mst'\rangle \end{array}$$

End of case distinction.

Case OutputSuppress : According to [TR-OutputSuppress] $\tilde{c} = \textbf{if } (head(G) \sqcup \tilde{e}) \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else skip; } \checkmark$ where $S \vdash e \blacktriangleright \tilde{e}$. Case distinction on $lev(\Delta) \sqcup \Gamma(e)$:

Case $lev(\Delta) \sqcup \Gamma(e) \sqsubseteq l$: **(37)**

We unroll the execution of $\langle\langle c,\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle$:

$$\begin{array}{rll} & \langle\langle \textbf{output}_l(e),\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle & \\ \xrightarrow{o_l(v)} & & \\ & \langle\langle \square,\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle & \\ = & & \\ & \langle\langle c',\, m'\rangle,\, \langle \Gamma',\, \Delta'\rangle\rangle & \text{found } c',\, m',\, \Gamma',\, \Delta' \end{array}$$

We unroll the execution of $\langle \tilde{c},\, m \uplus mst\rangle$:

$$\begin{array}{rl} & \langle \textbf{if } (head(G) \sqcup \tilde{e}) \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else skip;} \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{\epsilon} & \quad (37),\, \Gamma(e) = mst(\tilde{e}) \text{ – lemma 1, } mst(head(G)) = head(\Delta) \\ & \langle \textbf{output}_l(e)\textbf{; } \succ \textbf{; } \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{o_l(e,v)} & \\ & \langle \succ \textbf{; } \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{\epsilon} & \\ & \langle \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{t} & \\ & \langle \square,\, m \uplus mst\rangle \\ = & \quad \text{found } \tilde{c}',\, mst' \\ & \langle \tilde{c}',\, m' \uplus mst'\rangle \end{array}$$

Case $lev(\Delta) \sqcup \Gamma(e) \not\sqsubseteq l$: **(38)** We unroll the execution of $\langle\langle c,\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle$:

$$\begin{array}{rll} & \langle\langle \textbf{output}_l(e),\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle & \\ \longrightarrow & & \\ & \langle\langle \square,\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle & \\ = & & \\ & \langle\langle c',\, m'\rangle,\, \langle \Gamma',\, \Delta'\rangle\rangle & \text{found } c',\, m',\, \Gamma',\, \Delta' \end{array}$$

We unroll the execution of $\langle \tilde{c},\, m \uplus mst\rangle$:

$$\begin{array}{rl} & \langle \textbf{if } (head(G) \sqcup \tilde{e}) \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else skip;} \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{\epsilon} & \quad (38),\, \Gamma(e) = mst(\tilde{e}) \text{ – lemma 1, } mst(head(G)) = head(\Delta) \\ & \langle \textbf{skip; } \succ \textbf{; } \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{\epsilon} & \\ & \langle \succ \textbf{; } \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{\epsilon} & \\ & \langle \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{t} & \\ & \langle \square,\, m \uplus mst\rangle \\ = & \quad \text{found } \tilde{c}',\, mst' \\ & \langle \tilde{c}',\, m' \uplus mst'\rangle \end{array}$$

End of case distinction.

Case OutputDefault/Suppress :

According to [TR-OutputDefault/Suppress] $\tilde{c} = \textbf{if } head(G) \sqsubseteq l \textbf{ then } (\textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D)) \textbf{ else skip;} \checkmark$ where $S \vdash e \blacktriangleright \tilde{e}$.

Case distinction on $lev(\Delta)$:

Case $lev(\Delta) \sqsubseteq l$: **(39)** Case distinction on $\Gamma(e)$:

Case $\Gamma(e) \sqsubseteq l$: (**40**) We unroll the execution of $\langle\langle c,\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle$:

$$\begin{array}{ll} & \langle\langle \textbf{output}_l(e),\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle \\ \xrightarrow{o_l(v)} & \\ = & \langle\langle \square,\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle \\ & \langle\langle c',\, m'\rangle,\, \langle \Gamma',\, \Delta'\rangle\rangle \end{array} \qquad \text{found } c',\, m',\, \Gamma',\, \Delta'$$

We unroll the execution of $\langle \tilde{c},\, m \uplus mst\rangle$:

$$\begin{array}{ll} & \langle \textbf{if } head(G) \sqsubseteq l \textbf{ then } (\textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D)) \textbf{ else skip;} \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{\epsilon} & \quad (39),\, \Gamma(e) = mst(\tilde{e}) - \text{lemma 1},\, mst(head(G)) = head(\Delta) \\ & \langle \textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D) \textbf{;} \succ \textbf{;} \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{\epsilon} & \quad (40),\, \Gamma(e) = mst(\tilde{e}) - \text{lemma 1},\, mst(head(G)) = head(\Delta) \\ & \langle \textbf{output}_l(e) \textbf{;} \succ \textbf{;} \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{o_l(e,v)} & \\ & \langle \succ \textbf{;} \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{\epsilon} & \\ & \langle \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{t} & \\ & \langle \square,\, m \uplus mst\rangle \\ = & \quad \text{found } \tilde{c}',\, mst' \\ & \langle \tilde{c}',\, m' \uplus mst'\rangle \end{array}$$

Case $\Gamma(e) \not\sqsubseteq l$: (**41**) We unroll the execution of $\langle\langle c,\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle$:

$$\begin{array}{ll} & \langle\langle \textbf{output}_l(e),\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle \\ \xrightarrow{o_l(D)} & \\ = & \langle\langle \square,\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle \\ & \langle\langle c',\, m'\rangle,\, \langle \Gamma',\, \Delta'\rangle\rangle \end{array} \qquad \text{found } c',\, m',\, \Gamma',\, \Delta'$$

We unroll the execution of $\langle \tilde{c},\, m \uplus mst\rangle$:

$$\begin{array}{ll} & \langle \textbf{if } head(G) \sqsubseteq l \textbf{ then } (\textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D)) \textbf{ else skip;} \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{\epsilon} & \quad (39),\, \Gamma(e) = mst(\tilde{e}) - \text{lemma 1},\, mst(head(G)) = head(\Delta) \\ & \langle \textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D) \textbf{;} \succ \textbf{;} \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{\epsilon} & \quad (41),\, \Gamma(e) = mst(\tilde{e}) - \text{lemma 1},\, mst(head(G)) = head(\Delta) \\ & \langle \textbf{output}_l(D) \textbf{;} \succ \textbf{;} \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{o_l(D,D)} & \\ & \langle \succ \textbf{;} \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{\epsilon} & \\ & \langle \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{t} & \\ & \langle \square,\, m \uplus mst\rangle \\ = & \quad \text{found } \tilde{c}',\, mst' \\ & \langle \tilde{c}',\, m' \uplus mst'\rangle \end{array}$$

End of case distinction.

Case $lev(\Delta) \not\sqsubseteq l$: (**42**) We unroll the execution of $\langle\langle c,\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle$:

$$
\begin{array}{cll}
 & \langle\langle \textbf{output}_l(e),\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle & \\
\longrightarrow & \langle\langle \square,\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle & \\
= & \langle\langle c',\ m'\rangle,\ \langle \Gamma',\ \Delta'\rangle\rangle & \text{found } c',\ m',\ \Gamma',\ \Delta'
\end{array}
$$

We unroll the execution of $\langle \tilde{c},\ m \uplus mst\rangle$:

$$
\begin{array}{cll}
 & \langle \textbf{if } head(G) \sqsubseteq l \textbf{ then } (\textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D)) \textbf{ else skip;} \checkmark,\ m \uplus mst\rangle & \\
\xrightarrow{\epsilon} & \quad (42),\ \Gamma(e) = mst(\tilde{e}) \text{ – lemma 1, } mst(head(G)) = head(\Delta) & \\
 & \langle \textbf{skip;} \succ \textbf{;} \checkmark,\ m \uplus mst\rangle & \\
\xrightarrow{\epsilon} & \quad (41),\ \Gamma(e) = mst(\tilde{e}) \text{ – lemma 1, } mst(head(G)) = head(\Delta) & \\
 & \langle \succ \textbf{;} \checkmark,\ m \uplus mst\rangle & \\
\xrightarrow{\epsilon} & & \\
 & \langle \checkmark,\ m \uplus mst\rangle & \\
\xrightarrow{t} & & \\
 & \langle \square,\ m \uplus mst\rangle & \\
= & \quad \text{found } \tilde{c}',\ mst' & \\
 & \langle \tilde{c}',\ m' \uplus mst'\rangle &
\end{array}
$$

End of case distinction.

End of case distinction.

We show that for each of the cases above $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ is a valid instance of [TR-Stop] for $G' = G$, $S' = S$ and $I' = I$:

$$G, S, I \vdash \square \blacktriangleright \square$$

We show that $\langle \Gamma',\ \Delta'\rangle \cong (G', S', I', mst')$: by substituting $\Gamma' = \Gamma$, $\Delta' = \Delta$, $G' = G$, $S' = S$ and $I' = I$ and from the hypothesis of the lemma:

$$\langle \Gamma,\ \Delta\rangle \cong (G, S, I, mst)$$

Case $c = \succ\textbf{;}\ c_2$:

- $\tilde{c} = (\kappa(head(I), S, head(G))\textbf{;} \succ \textbf{;} \checkmark)\textbf{;}\ \tilde{c}_2$ where $tail(G), S, tail(I) \vdash c_2 \blacktriangleright \tilde{c}_2$ **(43)**
- $c' = c_2$ and $m' = m$ according to language semantics
- $\Gamma' = \Gamma \sqcup \{[x \mapsto l] \mid x \in xs\}$ and $\Delta' = tail(\Delta)$, where $\Delta = ((xs, l) :: \Delta')$

We unroll the execution of $\tilde{c}$ and use the results obtained in the base case for $\succ$:

$$
\begin{array}{cll}
 & \langle \tilde{c},\ m \uplus mst\rangle & \\
= & & \\
 & \langle (\kappa(head(I), S, head(G))\textbf{;} \succ \textbf{;} \checkmark)\textbf{;}\ \tilde{c}_2,\ m \uplus mst\rangle & \\
\xrightarrow{\epsilon}\xrightarrow{t} & & \text{see base case } \succ\text{, semantics for } c_1\textbf{;}\ c_2 \\
 & \langle \tilde{c}_2,\ m \uplus (mst \sqcup (head(I) \circ S^{-1}))\rangle & \\
= & & \text{found } \tilde{c}',\ mst' \\
 & \langle \tilde{c}',\ m' \uplus mst'\rangle &
\end{array}
$$

We show that $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ for $G' = tail(G)$, $S' = S$ and $I' = tail(I)$:

$$\frac{(43)}{tail(G), S, tail(I) \vdash c_2 \blacktriangleright \tilde{c}_2}$$

The proof of $\langle \Gamma',\ \Delta'\rangle \cong (G', S', I', mst')$ follows the correspondent proof in base case $\succ$.

Induction hypothesis : Suppose $G, S, I \vdash c_1 \blacktriangleright \tilde{c}_1 \wedge \langle \tilde{c},\ m \uplus mst \rangle \overset{\epsilon}{\Longrightarrow} \overset{t}{\longrightarrow} \langle \tilde{c}',\ m' \uplus mst' \rangle \wedge G', S', I' \vdash c' \blacktriangleright \tilde{c}' \wedge \langle \Gamma',\ \Delta' \rangle \cong (G', S', I', mst')$

Case $c = c_1; c_2$ **when** $c_1 \neq \succ$:

- $\tilde{c} = \tilde{c_1}; \tilde{c_2}$ where $G, S, I \vdash c_1 \blacktriangleright \tilde{c_1}$ and $G, S, I \vdash c_2 \blacktriangleright \tilde{c_2}$ (**44**)
- suppose $\langle \langle c_1,\ m \rangle,\ \langle \Gamma,\ \Delta \rangle \rangle \xrightarrow{\sigma} \langle \langle c_1',\ m'' \rangle,\ config \Gamma'' \Delta'' \rangle$, then $m' = m''$, $\Gamma' = \Gamma''$ and $\Delta' = \Delta''$ according to semantics of $c_1; c_2$ and monitor semantics.

Case distinction on c_1':

Case $c_1' = \square$:

- $c' = c_2$ according to semantics for sequence

Let's unroll the execution of $\langle \tilde{c},\ m \uplus mst \rangle$:

$$\begin{array}{lll} & \langle \tilde{c_1}; \tilde{c_2},\ m \uplus mst \rangle & \\ \overset{\alpha}{\Longrightarrow} \overset{t}{\longrightarrow} & & \text{semantics, induction hypothesis} \\ & \langle \tilde{c_2},\ m' \uplus mst'' \rangle & \\ = & & \text{presenting } \tilde{c}',\ mst' \\ & \langle \tilde{c}',\ m' \uplus mst' \rangle & \end{array}$$

With the help of the induction hypothesis and the execution trace above, we can easily see that substituting $G' = G$, $S' = S$ and $I' = I$ in $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ gives us a valid transformation rule which is a part of our assumptions:

$$\frac{(44)}{G, S, I \vdash c_2 \blacktriangleright \tilde{c_2}}$$

And from the induction hypothesis we get $cfgm' \cong (G', S', I', mst')$:

Case $c_2' \neq \square$:

- $c' = c_1'; c_2$ according to semantics of sequence

Let's unroll the execution of $\langle \tilde{c},\ m \uplus mst \rangle$:

$$\begin{array}{lll} & \langle \tilde{c_1}; \tilde{c_2},\ m \uplus mst \rangle & \\ \overset{\alpha}{\Longrightarrow} \overset{t}{\longrightarrow} & & \text{semantics, induction hypothesis} \\ & \langle \tilde{c_1}'; \tilde{c_2},\ m' \uplus mst'' \rangle & \\ = & & \text{presenting } \tilde{c}',\ mst' \\ & \langle \tilde{c}',\ m' \uplus mst' \rangle & \end{array}$$

Note that the induction hypothesis implies that $\exists G'', S'', I'' \mid G'', S'', I'' \vdash c_1' \blacktriangleright \tilde{c_1}'$ (**45**), $cfgm' \cong (G'', S'', I'', mst')$

From the induction hypothesis and by substituting $G' = G''$, $S' = S''$ and $I' = I''$ into $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ we get a valid instance of [TR-Seq]:

$$\frac{(44) \qquad (45)}{G', S', I' \vdash c_1'; c_2 \blacktriangleright \tilde{c_1}'; \tilde{c_2}}$$

Last, from the induction hypothesis and by substituting $G' = G''$, $S' = S''$ and $I' = I''$ we obtain $cfgm' \cong (G', S', I', mst')$.

End of case distinction.

End of rule induction.

□

Lemma 5 For all $c, c',\ \tilde{c}, \tilde{c}',\ G,\ S,\ I,\ m, mst, m', mst',\ \vec{\beta}$ such that $G, S, I \vdash c \blacktriangleright \tilde{c}'$ if $\langle \tilde{c},\ m \uplus mst \rangle \overset{\vec{\beta}}{\Longrightarrow} \langle \square,\ m' \uplus mst' \rangle$ then there exist $\vec{\beta}'$ such that $\langle \tilde{c},\ m \uplus mst \rangle \overset{\vec{\beta}'}{\Longrightarrow} \overset{t}{\longrightarrow} \langle \square,\ m' \uplus mst' \rangle$

Proof: By structural induction on c and observing the trace of $\langle \tilde{c},\ m \uplus mst \rangle$ in each case. This would essentially resemble proofs of lemmas 4 and 6 so we save the space and don't specify it here.

Informally, this theorem holds because, according to the transformation rules, every command that reduces to a $\square$ is transformed to a sequence of commands that ends with a t-transition. $\square$

Lemma 6 For all c, $\tilde{c}$, m, mst, Γ, Δ, G, S, I, c', m' such that $\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle \sim \langle\tilde{c},\, m \uplus mst\rangle$ if $\langle\tilde{c},\, m \uplus mst\rangle \overset{\alpha}{\Longrightarrow}\overset{t}{\longrightarrow} \langle\tilde{c}',\, m' \uplus mst'\rangle$ then there exist c', σ, Γ' and Δ' such that $\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle \xrightarrow{\sigma} \langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle$, and $\langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle \sim \langle\tilde{c}',\, m' \uplus mst'\rangle$, and $\alpha \triangleright \sigma$

Proof: The proof of lemma 6 largely resembles the proof for lemma 4. However, for lemma 4 we are considering several steps leading up to a t; so the argument in this lemma is not rule induction on $\longrightarrow$ but rather a structural induction on c. In each case of c, we consider several steps of $\tilde{c}$.

Structural induction on c:

Case $c = \textbf{skip}$:

The transformed command looks like $\tilde{c} = \textbf{skip;}\checkmark$ according to [TR-Skip].

First, we unroll the execution of execution of $\langle\tilde{c},\, m \uplus mst\rangle$ in order to find $\tilde{c}'$, m', mst':

$$\begin{array}{cll} \xrightarrow{\epsilon} & \langle \textbf{skip;}\checkmark,\, m \uplus mst\rangle & \\ & & \text{semantics} \\ \xrightarrow{t} & \langle\checkmark,\, m \uplus mst\rangle & \\ & & \text{semantics} \\ = & \langle\Box,\, m \uplus mst\rangle & \\ & \langle\tilde{c}',\, m' \uplus mst'\rangle & \text{found } \tilde{c}',\, m' \text{ and } mst' \end{array}$$

Next, we show how one step of $\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle$ could match the trace of the transformed command:

$$\begin{array}{cll} \longrightarrow & \langle\langle \textbf{skip},\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle & \\ = & \langle\langle\Box,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle & \\ & \langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle & \text{found } c',\, \Gamma',\, \Delta' \end{array}$$

Finally, we need to prove that $\langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle \sim \langle\tilde{c}',\, m' \uplus mst'\rangle$.

To do that we need to prove that there exist such G', S', I' such that $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ and $\langle\Gamma',\, \Delta'\rangle \cong (G', S', I', mst')$.

It is obvious that $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ according to [TRi-stop] independent of the values of G', S', I'.

We show that $\langle\Gamma',\, \Delta'\rangle \cong (G', S', I', mst')$ by substituting G' with G, S' with S, I' with I, mst' with mst, Γ' with Γ and Δ' with Δ and by the hypothesis of the lemma.

Case $c = \succ$:

According to [TR-End] $\tilde{c} = \kappa(head(I), S, head(G))\textbf{;}\ \succ\textbf{;}\checkmark$.

First, we unroll the execution of $\langle\tilde{c},\, m \uplus mst\rangle$ until the first $\checkmark$ event to find $\tilde{c}'$, m' and mst':

$$\begin{array}{cll} \overset{\epsilon}{\Longrightarrow} & \langle\kappa(head(I), S, head(G))\textbf{;}\ \succ\textbf{;}\checkmark,\, m \uplus mst\rangle & \\ & \langle\succ\textbf{;}\checkmark,\, m \uplus (mst \sqcup (mkFun(head(\Delta)) \circ S^{-1}))\rangle & \text{lemma 3, semantics} \\ \xrightarrow{\epsilon} & \langle\checkmark,\, m \uplus (mst \sqcup (mkFun(head(\Delta)) \circ S^{-1}))\rangle & \\ \xrightarrow{t} & \langle\Box,\, m \uplus (mst \sqcup (mkFun(head(\Delta)) \circ S^{-1}))\rangle & \\ = & \langle\tilde{c}',\, m' \uplus mst'\rangle & \text{found } \tilde{c}',\, m',\, mst' \textbf{ (1)} \end{array}$$

Next, we show a step of the VM-monitored configuration $\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle$:

$$\begin{array}{cll} \longrightarrow & \langle\langle\succ,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle & \\ = & \langle\langle\Box,\, m\rangle,\, \langle\Gamma \sqcup mkFun(head(\Delta)),\, tail(\Delta)\rangle\rangle & \\ & \langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle & \text{found } c',\, \Gamma' \textbf{ (2)},\, \Delta' \textbf{ (3)} \end{array}$$

We need to prove that $\langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle \sim \langle\tilde{c}',\, m' \uplus mst'\rangle$.

To do that we need to prove that there exist such G', S', I' such that $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ and $\langle\Gamma',\, \Delta'\rangle \cong (G', S', I', mst')$. It is obvious that $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ according to [TR-Stop] independent of the values of G', S', I'.

Finally, we show that for $S' = S$ **(4)**, $G' = tail(G)$ **(5)** and $I' = tail(I)$ **(6)** $\langle\Gamma',\, \Delta'\rangle \cong (G', S', I', mst')$:

1) we show that $\Gamma' = mst' \circ S'$:

$$\begin{array}{ll} & \Gamma' \\ = & \Gamma \sqcup mkFun(head(\Delta)) \quad \text{by (2)} \\ = & (mst \circ S) \sqcup mkFun(head(\Delta)) \quad \text{by the hypothesis of the lemma} \\ = & (mst \circ S) \sqcup ((mkFun(head(\Delta)) \circ S^{-1}) \circ S) \quad S^{-1} \circ S = Id,\ S \text{ is injective} \\ = & (mst \sqcup (mkFun(head(\Delta)) \circ S^{-1})) \circ S \quad \text{distribute composition over union} \\ = & mst' \circ S' \quad \text{by (1) and (4)} \end{array}$$

2) we show that $\Delta' = zip(I', mst' \circ G')$:

$$\begin{array}{ll} & \Delta' \\ = & tail(\Delta) \quad (3) \\ = & tail(zip(I, mst \circ G)) \quad \text{the hypothesis of the lemma} \\ = & tail(((head(I), head(mst \circ G)) :: zip(tail(I))), tail(mst \circ G)) \quad \text{definition of } zip \\ = & zip(tail(I), tail(mst \circ G)) \quad \text{definition of } tail \\ = & zip(tail(I), tail((mst \sqcup (mkFun(head(\Delta)) \circ S^{-1})) \circ G)) \quad dom(S^{-1}) \cap rng(G) = \emptyset \\ = & zip(tail(I), (mst \sqcup (mkFun(head(\Delta)) \circ S^{-1})) \circ tail(G)) \\ = & zip(I', mst' \circ G') \quad (6), (5), (1) \end{array}$$

Case $c = \square$:

According to [TR-Stop] $\tilde{c} = \square$.

A configuration $\langle \square,\ m \uplus mst \rangle$ could not make progress according to language semantics. This does not fulfill our assumptions. This case is, thus, unrealizable.

Case $c = \checkmark$:

This case is unrealizable because $\checkmark$ is not part of the user language and there is no way to obtain $\tilde{c}$ from it according to transformation rules.

Case $c = x \,\mathbf{:=}\, e$:

According to [TR-Assign], $\tilde{c} = S(x) \,\mathbf{:=}\, \tilde{e} \sqcup head(G)\mathbf{;}\ x \,\mathbf{:=}\, e\mathbf{;}\ \checkmark$ where $S \vdash e \blacktriangleright \tilde{e}$.

First we unroll the execution of $\langle \tilde{c},\ m \uplus mst \rangle$, applying language semantic rules for $c\mathbf{;}\, c$, $x \,\mathbf{:=}\, e$ and $\checkmark$ appropriately:

$$\begin{array}{ll} & \langle S(x) \,\mathbf{:=}\, \tilde{e} \sqcup head(G)\mathbf{;}\ x \,\mathbf{:=}\, e\mathbf{;}\ \checkmark,\ m \uplus mst \rangle \\ \xrightarrow{\epsilon} & \langle x \,\mathbf{:=}\, e\mathbf{;}\ \checkmark,\ m \uplus mst[S(x) \mapsto mst(\tilde{e}) \sqcup head(G)] \rangle \quad \text{Lemma 2} \\ \xrightarrow{\epsilon} & \langle \checkmark,\ m[x \mapsto m(e)] \uplus mst[S(x) \mapsto mst(\tilde{e}) \sqcup head(G)] \rangle \quad \text{semantics, Lemma 2} \\ \xrightarrow{t} & \langle \square,\ m[x \mapsto m(e)] \uplus mst[S(x) \mapsto mst(\tilde{e}) \sqcup head(G)] \rangle \\ = & \langle \tilde{c}',\ m' \uplus mst' \rangle \quad \text{found } \tilde{c}',\ m' \text{ and } mst' \end{array}$$

We show a step of the VM-monitored configuration $\langle\langle c,\ m \rangle,\ \langle \Gamma,\ \Delta \rangle\rangle$:

$$\begin{array}{ll} & \langle\langle x \,\mathbf{:=}\, e,\ m \rangle,\ \langle \Gamma,\ \Delta \rangle\rangle \\ \longrightarrow & \langle\langle \square,\ m[x \mapsto m(e)] \rangle,\ \langle \Gamma[x \mapsto lev(\Delta) \sqcup \Gamma(e)],\ \Delta \rangle\rangle \\ = & \langle\langle c',\ m' \rangle,\ \langle \Gamma',\ \Delta' \rangle\rangle \quad \text{found } c',\ \Gamma' \textbf{ (7)} \text{ and } \Delta' = \Delta \textbf{ (8)} \end{array}$$

We prove that $\langle\langle c',\ m' \rangle,\ \langle \Gamma',\ \Delta' \rangle\rangle \sim \langle \tilde{c}',\ m' \uplus mst' \rangle$.

$G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ is valid for any G', S', I': we obtain a valid instance of [TR-Stop]

$$G', S', I' \vdash \square \blacktriangleright \square$$

Next, we show that for $G' = G$ (**9**), $S' = S$ (**10**), $I' = I$ (**11**) we get $\langle \Gamma',\ \Delta' \rangle \cong (G', S', I', mst')$. We need to prove that $\Gamma' = mst' \circ S'$ and $|\Delta'| = |G'| = |I'|$ and $\forall n \in 1..|\Delta'| \mid \Delta'(i) = \{[x \mapsto l] \mid x \in I'(n), l = G'(n)\}$

- we prove that $\Gamma' = mst' \circ S'$, i.e. $\forall z \in dom(\Gamma').\Gamma'(z) = mst'(S(z))$: Case distinction on z:
  Case $z = x$:

$$\begin{array}{ll} & \Gamma'(z) \\ = & \Gamma[x \mapsto lev(\Delta) \sqcup \Gamma(e)](x) \\ = & \Gamma(e) \sqcup lev(\Delta) \\ = & mst(\tilde{e}) \sqcup lev(\Delta) \\ = & mst[S(x) \mapsto mst(\tilde{e}) \sqcup head(G)](S(x)) \\ = & mst'(S(z)) \end{array} \quad \begin{array}{l} (9) \\ \text{definition of mapping update} \\ \text{Lemma 1, assumption} \\ \text{definition of mapping update} \\ \text{(mst'assign??)} \end{array}$$

  Case $z \neq x$:

$$\begin{array}{ll} & \Gamma'(z) \\ = & \Gamma[x \mapsto \Gamma(e) \sqcup lev(\Delta)](z) \\ = & \Gamma(z) \sqcup lev(\Delta) \\ = & mst(z) \sqcup head(G) \\ = & mst[S(x) \mapsto mst(\tilde{e}) \sqcup head(G)](S(z)) \\ = & mst'(S(z)) \end{array} \quad \begin{array}{l} \\ \text{definition of mapping update} \\ \text{Lemma 1, assumption} \\ \text{definition of mapping update} \\ \text{(mst'assign??)} \end{array}$$

  End of case distinction.

- we prove that $\Delta' = zip(I', mst' \circ G')$:

$$\begin{array}{ll} & \Delta' \\ = & \Delta \\ = & zip(I, mst \circ G) \\ = & zip(I, (mst \sqcup [S(x) \mapsto head(G)]) \circ G) \\ = & zip(I, mst[S(x) \mapsto mst(\tilde{e}) \sqcup head(G)] \circ G) \\ = & zip(I', mst' \circ G') \end{array} \quad \begin{array}{l} (8) \\ \\ dom(S) \cap rng(G) = \emptyset \\ \\ \text{(mst'assign??), (9), (11)} \end{array}$$

Case $c =$ **if** e **then** c_1 **else** c_2 :
Without limiting the generality, we can assume that $m(e) \neq 0$. The proof for the case when $m(e) = 0$ is symmetric.
According to [TR-If]

$$\tilde{c} = x \textbf{ := } \tilde{e} \sqcup head(G)\textbf{; if } e \textbf{ then } \tilde{c}_1' \textbf{ else } \tilde{c}_2'\textbf{; } \checkmark$$

where $S \vdash e \blacktriangleright \tilde{e}$ and $x \notin \text{dom}(S) \cup \text{rng}(S) \wedge x \notin G$ and

$$(x :: G), S, (mod(c_{i \bmod 2+1}) :: I) \vdash c_i \blacktriangleright \tilde{c}_i$$

(**12**) and

$$\tilde{c}_i' = \checkmark\textbf{; } \tilde{c}_i\textbf{; } \kappa(mod(c_{(i \bmod 2)+1}), S, x)$$

First, we unroll the execution of $\langle \tilde{c},\ m \uplus mst \rangle$:

$$
\begin{array}{ll}
 & \langle x := \tilde{e} \sqcup head(G); \textbf{if } e \textbf{ then } \tilde{c}_1' \textbf{ else } \tilde{c}_2'; \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \textbf{if } e \textbf{ then } \tilde{c}_1' \textbf{ else } \tilde{c}_2'; \checkmark,\ m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
\xrightarrow{\epsilon} & \quad \text{semantics, assumption } m(e) \neq 0, \text{ substitution for } \tilde{c}_1' \\
 & \langle \checkmark; \tilde{c}_1; \kappa(mod(c_2), S, x); \succ; \checkmark,\ m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
\xrightarrow{t} & \\
 & \langle \tilde{c}_1; \kappa(mod(c_2), S, x); \succ; \checkmark,\ m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
= & \quad \text{found } \tilde{c}',\ m' \text{ and } mst' \textbf{ (13)} \\
 & \langle \tilde{c}',\ m' \uplus mst' \rangle
\end{array}
$$

We show the step of the VM-monitored configuration that matches the multi-step trace of the transformed command:

$$
\begin{array}{ll}
 & \langle\langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2,\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle \\
\longrightarrow & \\
 & \langle\langle c_1; \succ,\ m\rangle,\ \langle \Gamma,\ (\Gamma(e) \sqcup lev(\Delta) :: \Delta)\rangle\rangle \\
= & \\
 & \langle\langle c',\ m'\rangle,\ \langle \Gamma',\ \Delta'\rangle\rangle
\end{array}
\quad \text{found } c',\ m',\ \Gamma' \textbf{ (14)},\ \Delta' \textbf{ (15)}
$$

Next, we show that $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ for $G' = (x :: G)$ **(16)**, $S' = S$ **(17)** and $I' = (mod(c_2) :: I)$ **(18)**.

$$
\text{TR-SEQ}\ \frac{\dfrac{(12)}{(x :: G), S, (mod(c_2) :: I) \vdash c_1 \blacktriangleright \tilde{c}_1} \quad (x :: G), S, (mod(c_2) :: I) \vdash \succ \blacktriangleright \kappa(mod(c_2), S, x); \succ; \checkmark\ \text{TR-END}}{G' = (x :: G), S, (mod(c_2) :: I) \vdash c_1; \succ \blacktriangleright \tilde{c}_1; \kappa(mod(c_2), S, x); \succ; \checkmark}
$$

Last, we show that $\langle \Gamma',\ \Delta' \rangle \cong (G', S', I', mst')$ for the given G', S', I'.

1) we prove that $\Gamma' = mst' \circ S'$

$$
\begin{array}{ll}
 & \Gamma' \\
= & \quad \text{by (14)} \\
 & \Gamma \\
= & \quad \text{the hypothesis of the lemma} \\
 & mst \circ S \\
= & \quad \text{since } x \notin dom(S) \\
 & mst \circ S \sqcup [x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \circ S \\
= & \quad \text{distribute function composition over union} \\
 & mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \circ S \\
= & \quad \text{by (13),(17)} \\
 & mst' \circ S'
\end{array}
$$

2) we prove that $\Delta' = zip(I', mst' \circ G')$:

$$
\begin{array}{ll}
 & \Delta' \\
= & \quad (15) \\
 & (\text{update}_{\Gamma(e) \sqcup lev(\Delta)}(c_2) :: \Delta) \\
= & \quad \text{definition of update}_l(c) \\
 & ((mod(c_2), \Gamma(e) \sqcup lev(\Delta)) :: \Delta) \\
= & \quad \text{definition of } zip\text{, the hypothesis of the lemma, definition of } lev \\
 & zip((mod(c_2) :: I), (\Gamma(e) \sqcup lev(\Delta)) :: mst \circ G)) \\
= & \quad \text{lemma 1} \\
 & zip((mod(c_2) :: I), (mst(\tilde{e}) \sqcup mst(head(G)) :: mst \circ G)) \\
= & \\
 & zip((mod(c_2) :: I), (mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))](x) :: mst \circ G)) \\
= & \quad \text{distribute composition over list construction, } x \notin rng(G) \\
 & zip((mod(c_2) :: I), mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \circ (x :: G)) \\
= & \quad (18), (13), (16) \\
 & zip(I', mst' \circ G')
\end{array}
$$

Case $c =$ **while** e **do** c_1 :

There are two transformed commands that might correspond to c.

Case distinction on $\tilde{c}$:

Case $x :=$ **;while** e **do** $(\checkmark; \tilde{c_1}; \text{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(c_1, S, x); \succ; \checkmark$:

according to [TR-While] where $S \vdash e \blacktriangleright \tilde{e}$, and $(x :: G), S, (\emptyset :: I) \vdash c_1 \blacktriangleright \tilde{c_1}$ (**19**) and $x \notin \text{dom}(S) \cup \text{rng}(S) \wedge x \notin G$.

Case distinction on $m(e)$:

Case $m(e) \neq 0$:

First, we unroll the execution of $\langle \tilde{c},\ m \uplus mst \rangle$ in order to find $\tilde{c}'$, m' and mst' keeping in mind that $m(e) \neq 0$:

$$
\begin{array}{ll}
 & \langle x := \tilde{e} \sqcup head(G); \text{while } e \text{ do } (\checkmark; \tilde{c_1}; \text{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(mod(c_1), S, x); \succ; \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \text{while } e \text{ do } (\checkmark; \tilde{c_1}; \text{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(mod(c_1), S, x); \succ; \checkmark, \\
 & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
\xrightarrow{\epsilon} & \quad \text{semantics, } m(e) \neq 0 \\
 & \langle \checkmark; \tilde{c_1}; \text{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x; \succ; \text{while } e \text{ do } (\checkmark; \tilde{c_1}; \text{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(mod(c_1), S, x); \succ; \checkmark, \\
 & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
\xrightarrow{t} & \\
 & \langle \tilde{c_1}; \text{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x; \succ; \text{while } e \text{ do } (\checkmark; \tilde{c_1}; \text{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(mod(c_1), S, x); \succ; \checkmark, \\
 & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
= & \quad \text{found } \tilde{c}',\ mst' \text{ (\textbf{20})} \\
 & \langle \tilde{c}',\ m \uplus mst' \rangle
\end{array}
$$

Next we show a step of $\langle\langle c,\ m \rangle,\ \langle \Gamma,\ \Delta \rangle\rangle$ that matches the trace presented above:

$$
\begin{array}{lll}
 & \langle\langle \text{while } e \text{ do } c_1,\ m \rangle,\ \langle \Gamma,\ \Delta \rangle\rangle & \\
\longrightarrow & \langle\langle c_1; \succ; \text{while } e \text{ do } c_1,\ m \rangle,\ \langle \Gamma,\ (\text{update}_{\Gamma(e) \sqcup lev(\Delta)}(\text{skip}) :: \Delta) \rangle\rangle & \text{semantics, } m(e) \neq 0 \\
= & \langle\langle c',\ m' \rangle,\ \langle \Gamma',\ \Delta' \rangle\rangle & \text{found } c', \Gamma' \text{ (\textbf{21}) and } \Delta' \text{ (\textbf{22})}
\end{array}
$$

Next, we show that $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ is a valid instance of [TR-Seq] for $G = (x :: G)$, (**23**), $S' = S$ (**24**) and $I' = (\emptyset :: I)$ (**25**).

$$\text{TR-Seq}\ \frac{\text{TR-EndSeq}\ \dfrac{\text{TR-WhileLoop}\ \dfrac{\dfrac{(19)}{(x :: G), S, (\emptyset :: I) \vdash c_1 \blacktriangleright \tilde{c_1}}}{\begin{array}{l} G, S, I \vdash \textbf{while}\ e\ \textbf{do}\ c_1 \blacktriangleright \\ x := \tilde{e} \sqcup head(G); \succ; \\ \textbf{while}\ e\ \textbf{do}\ (\checkmark; \tilde{c_1}); \checkmark; \kappa(mod(c_1), S, x); \succ; \checkmark \end{array}}}{\begin{array}{l} \kappa(\emptyset, S, x) = \textbf{skip} \\ (x :: G), S, (\emptyset :: I) \vdash \succ; \textbf{while}\ e\ \textbf{do}\ c_1 \blacktriangleright \\ \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup head(G); \succ; \textbf{while}\ e\ \textbf{do}\ (\checkmark; \tilde{c_1}); \checkmark; \\ \kappa(mod(c_1), S, x); \succ; \checkmark \end{array}}}{\begin{array}{l} (x :: G), S, (\emptyset :: I) \vdash c_1; \succ; \textbf{while}\ e\ \textbf{do}\ c_1 \blacktriangleright \\ \tilde{c_1}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup head(G); \succ; \textbf{while}\ e\ \textbf{do}\ (\checkmark; \tilde{c_1}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup head(G)); \\ \checkmark; \kappa(mod(c_1), S, x); \succ; \checkmark \end{array}}$$

We show that $\langle \Gamma', \Delta' \rangle \cong (G', S', I', mst')$:

1) we prove that $\Gamma' = mst' \circ S'$

$$\begin{array}{ll} & \Gamma' \\ = & \quad (21) \\ & \Gamma \\ = & \quad \text{the hypothesis of the lemma} \\ & mst(S(z)) \\ = & \quad \text{by definition of mapping update and because } x \notin dom(S) \\ & (mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))])(S(z)) \\ = & \quad (20) \\ & mst'(S(z)) \end{array}$$

2) we prove that $\Delta' = zip(I', mst' \circ G')$:

$$\begin{array}{ll} & \Delta' \\ = & \quad (22) \\ & (\text{update}_{\Gamma(e) \sqcup lev(\Delta)}(\textbf{skip}) :: \Delta) \\ = & \quad \text{definition of } lev \\ & ((mod(\textbf{skip}), \Gamma(e) \sqcup head(\Delta)) :: \Delta) \\ = & \quad \text{lemma 1, the hypothesis of the lemma, definition of } mod() \\ & ((\emptyset, mst(\tilde{e}) \sqcup mst(head(G))) :: zip(I, mst \circ G)) \\ = & \quad \text{definition of } zip \\ & zip((\emptyset :: I), (mst(\tilde{e}) \sqcup mst(head(G)) :: mst \circ G)) \\ = & \quad x \notin rng(G) \\ & zip((\emptyset :: I), (mst[x \mapsto mst(\tilde{e}) mst(head(G))](x) :: mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \circ G)) \\ = & \quad \text{distribute composition over list contructor} \\ & zip((\emptyset :: I), mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \circ (x :: G)) \\ = & \quad (23), (25), (20) \\ & zip(I', mst' \circ G') \end{array}$$

Case $m(e) = 0$:

We unroll the execution of $\langle \tilde{c},\ m \uplus mst \rangle$:

$$
\begin{array}{ll}
 & \langle x := \tilde{e} \sqcup head(G); \textbf{while } e \textbf{ do } (\checkmark; \tilde{c_1}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(c_1, S, x); \succ; \checkmark, \\
 & m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \textbf{while } e \textbf{ do } (\checkmark; \tilde{c_1}; \textbf{skip}; \succ; \checkmark; x := \tilde{e} \sqcup x); \checkmark; \kappa(c_1, S, x); \succ; \checkmark, \\
 & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
\xrightarrow{\epsilon} & \quad \text{semantics, } m(e) = 0 \\
 & \langle \succ; \checkmark; \kappa(c_1, S, x); \succ; \checkmark,\ m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \checkmark; \kappa(c_1, S, x); \succ; \checkmark,\ m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
\xrightarrow{t} & \\
 & \langle \kappa(c_1, S, x); \succ; \checkmark,\ m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
= & \quad \text{found } \tilde{c}',\ mst' \textbf{ (26)} \\
 & \langle \tilde{c}',\ m' \uplus mst' \rangle
\end{array}
$$

Next, we show a step of $\langle\langle c,\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle$:

$$
\begin{array}{lll}
 & \langle\langle \textbf{while } e \textbf{ do } c_1,\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle & \\
\longrightarrow & \langle\langle \succ,\ m\rangle,\ \langle \Gamma,\ (\text{update}_{\Gamma(e) \sqcup lev(\Delta)}(c_1) :: \Delta)\rangle\rangle & \text{semantics, } m(e) = 0 \\
= & \langle\langle c',\ m'\rangle,\ \langle \Gamma',\ \Delta'\rangle\rangle & \text{found } c',\ m',\ \Gamma' \textbf{ (27)},\ \Delta' \textbf{ (28)}
\end{array}
$$

Finally, we show the equivalence of configurations:
We show that $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ for $G' = (x :: G)$ **(29)**, $S' = S$ **(30)**, $I' = (mod(c_1) :: I)$ **(31)** is a valid instance of [TR-End]:

$$(x :: G), S, (mod(c_1) :: I) \vdash \succ \blacktriangleright \kappa(mod(c_1), S, x); \succ; \checkmark$$

Last, we show that $\langle \Gamma',\ \Delta'\rangle \cong (G', S', I', mst')$:

1) we prove that $\Gamma' = mst' \circ S'$

$$
\begin{array}{ll}
 & \Gamma' \\
= & \quad (27) \\
 & \Gamma \\
= & \quad \text{the hypothesis of the lemma} \\
 & mst(S(z)) \\
= & \quad \text{by definition of mapping update and because } x \notin dom(S), (30) \\
 & mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \circ S \\
= & \quad \text{obtained earlier} \\
 & mst' \circ S'
\end{array}
$$

2) we prove that $\Delta' = zip(I', mst' \circ G')$:

$$
\begin{array}{ll}
 & \Delta' \\
= & \quad \text{(28)} \\
 & (\text{update}_{\Gamma(e) \sqcup lev(\Delta)}(c_1) :: \Delta) \\
= & \quad \text{definitions of update}_l(c) \text{ and } lev \\
 & ((mod(c_1), \Gamma(e) \sqcup head(\Delta)) :: \Delta) \\
= & \quad \text{lemma 1, the hypothesis of the lemma} \\
 & zip((mod(c_1) :: I), (mst(\tilde{e}) \sqcup mst(head(G)) :: mst \circ G)) \\
= & \quad x \notin rng(G) \\
 & zip((mod(c_1) :: I), (mst(\tilde{e}) \sqcup mst(head(G)) :: mst \circ G \sqcup [x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \circ G)) \\
= & \quad \text{distribute composition over union} \\
 & zip((mod(c_1) :: I), (mst(\tilde{e}) \sqcup mst(head(G)) :: mst \sqcup [x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \circ G)) \\
= & \quad \text{apply mapping lookup, rewrite update as union} \\
 & zip((mod(c_1) :: I), (mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))](x) :: mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \circ G)) \\
= & \quad \text{distribute composition over list construction} \\
 & zip((mod(c_1) :: I), mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \circ (x :: G)) \\
= & \quad \text{(31), (26), (29)} \\
 & zip(I', mst' \circ G')
\end{array}
$$

End of case distinction.

Case $x := \tilde{e} \sqcup x;\ \succ;$ **while** e **do** $(\checkmark;\ \tilde{c_1};$ **skip;** $\succ;\ \checkmark;\ x := \tilde{e} \sqcup x);\ \checkmark;\ \kappa(c_1, S, x);\ \succ;\ \checkmark$:
according to [TR-WhileLoop], where $S \vdash e \blacktriangleright \tilde{e}$, and $G, S, I \vdash c_1 \blacktriangleright \tilde{c_1}$, and $head(G) = x$ and $head(I) = \emptyset$.
Case distinction on $m(e)$:

Case $m(e) \neq 0$: We unroll the execution of $\langle \tilde{c},\ m \uplus mst \rangle$:

$$
\begin{array}{ll}
 & \langle x := \tilde{e} \sqcup head(G);\ \succ;\ \textbf{while } e \textbf{ do } (\checkmark;\ \tilde{c_1};\ \textbf{skip;}\ \succ;\ \checkmark;\ x := \tilde{e} \sqcup head(G));\ \checkmark;\ \kappa(mod(c_1), S, x);\ \succ;\ \checkmark, \\
 & m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \succ;\ \textbf{while } e \textbf{ do } (\checkmark;\ \tilde{c_1};\ \textbf{skip;}\ \succ;\ \checkmark;\ x := \tilde{e} \sqcup head(G));\ \checkmark;\ \kappa(mod(c_1), S, x);\ \succ;\ \checkmark, \\
 & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \textbf{while } e \textbf{ do } (\checkmark;\ \tilde{c_1};\ \textbf{skip;}\ \succ;\ \checkmark;\ x := \tilde{e} \sqcup head(G));\ \checkmark;\ \kappa(mod(c_1), S, x);\ \succ;\ \checkmark, \\
 & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
\xrightarrow{\epsilon} & \quad \text{semantics, } m(e) \neq 0 \\
 & \langle \begin{array}{l} \checkmark;\ \tilde{c_1};\ \textbf{skip;}\ \succ;\ \checkmark;\ x := \tilde{e} \sqcup head(G);\ \succ; \\ \textbf{while } e \textbf{ do } (\checkmark;\ \tilde{c_1};\ \textbf{skip;}\ \succ;\ \checkmark;\ x := \tilde{e} \sqcup head(G));\ \checkmark;\ \kappa(mod(c_1), S, x);\ \succ;\ \checkmark \end{array}\ , \\
 & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
\xrightarrow{t} & \\
 & \langle \begin{array}{l} \tilde{c_1};\ \textbf{skip;}\ \succ;\ \checkmark;\ x := \tilde{e} \sqcup head(G);\ \succ; \\ \textbf{while } e \textbf{ do } (\checkmark;\ \tilde{c_1};\ \textbf{skip;}\ \succ;\ \checkmark;\ x := \tilde{e} \sqcup head(G));\ \checkmark;\ \kappa(mod(c_1), S, x);\ \succ;\ \checkmark \end{array}\ , \\
 & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))] \rangle \\
= & \quad \text{found } \tilde{c}',\ mst' \\
 & \langle \tilde{c}',\ m \uplus mst' \rangle
\end{array}
$$

Next we show a step of $\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle$ that matches the trace presented above:

$$\begin{array}{ll} \longrightarrow & \langle\langle \textbf{while } e \textbf{ do } c_1,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle \\ & \langle\langle c_1\textbf{;} \succ\textbf{; while } e \textbf{ do } c_1,\, m\rangle,\, \langle\Gamma,\, (\text{update}_{\Gamma(e)\sqcup lev(\Delta)}(\textbf{skip}) :: \Delta)\rangle\rangle & \text{semantics, } m(e) \neq 0 \\ = & \langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle & \text{found } c',\, \Gamma' \text{ and } \Delta' \end{array}$$

The argument for why $\langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle \sim \langle\tilde{c}',\, m' \uplus mst'\rangle$ is the same as in previous case when $m(e) \neq 0$.

Case $m(e) = 0$:

We unroll the execution of $\langle\tilde{c},\, m \uplus mst\rangle$:

$$\begin{array}{ll} & \langle x \textbf{:=} \tilde{e} \sqcup x\textbf{; } \succ\textbf{; while } e \textbf{ do } (\checkmark\textbf{; } \tilde{c_1}\textbf{; skip; } \succ\textbf{; } \checkmark\textbf{; } x \textbf{:=} \tilde{e} \sqcup x)\textbf{; } \checkmark\textbf{; } \kappa(c_1, S, x)\textbf{; } \succ\textbf{; } \checkmark,\, m \uplus mst\rangle \\ \xrightarrow{\epsilon} & \\ & \langle \succ\textbf{; while } e \textbf{ do } (\checkmark\textbf{; } \tilde{c_1}\textbf{; skip; } \succ\textbf{; } \checkmark\textbf{; } x \textbf{:=} \tilde{e} \sqcup x)\textbf{; } \checkmark\textbf{; } \kappa(c_1, S, x)\textbf{; } \succ\textbf{; } \checkmark, \\ & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))]\rangle \\ \xrightarrow{\epsilon} & \\ & \langle \textbf{while } e \textbf{ do } (\checkmark\textbf{; } \tilde{c_1}\textbf{; skip; } \succ\textbf{; } \checkmark\textbf{; } x \textbf{:=} \tilde{e} \sqcup x)\textbf{; } \checkmark\textbf{; } \kappa(c_1, S, x)\textbf{; } \succ\textbf{; } \checkmark, \\ & m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))]\rangle \\ \xrightarrow{\epsilon} & \text{semantics, } m(e) = 0 \\ & \langle \succ\textbf{; } \checkmark\textbf{; } \kappa(c_1, S, x)\textbf{; } \succ\textbf{; } \checkmark,\, m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))]\rangle \\ \xrightarrow{\epsilon} & \\ & \langle \checkmark\textbf{; } \kappa(c_1, S, x)\textbf{; } \succ\textbf{; } \checkmark,\, m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))]\rangle \\ \xrightarrow{t} & \\ & \langle \kappa(c_1, S, x)\textbf{; } \succ\textbf{; } \checkmark,\, m \uplus mst[x \mapsto mst(\tilde{e}) \sqcup mst(head(G))]\rangle \\ = & \text{found } \tilde{c}' \text{ and } mst' \\ & \langle \tilde{c},\, m' \uplus mst'\rangle \end{array}$$

We show a step of $\langle\langle c,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle$ that matches the trace of $\langle\tilde{c},\, m \uplus mst\rangle$:

$$\begin{array}{ll} \longrightarrow & \langle\langle \textbf{while } e \textbf{ do } c_1,\, m\rangle,\, \langle\Gamma,\, \Delta\rangle\rangle \\ & \langle\langle \succ,\, m\rangle,\, \langle\Gamma,\, (\text{update}_{\Gamma(e)\sqcup lev(\Delta)}(c_1) :: \Delta)\rangle\rangle \\ = & \langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle \end{array} \quad \text{found } c',\, \Gamma',\, \Delta'$$

The argument for why $\langle\langle c',\, m'\rangle,\, \langle\Gamma',\, \Delta'\rangle\rangle \sim \langle\tilde{c}',\, m' \uplus mst'\rangle$ is the same as for the first case when $m(e) = 0$.

End of case distinction.

End of case distinction.

Case $c = \textbf{output}_l(e)$: Case distinction on the type of enforcement:

Case OutputFailstop : According to [TR-OutputFailstop] $\tilde{c} = \textbf{if } head(G) \sqcup \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else } (\textbf{diverge})\textbf{; } \checkmark$ where $S \vdash e \blacktriangleright \tilde{e}$.

We unroll the execution of $\langle\tilde{c},\, m \uplus mst\rangle$ to find $\tilde{c}'$, m' and mst'.

Case distinction on $mst(head(G) \sqcup \tilde{e})$:

Case $mst(head(G) \sqcup \tilde{e}) \sqsubseteq l$:

$$
\begin{aligned}
& \langle \textbf{if } head(G) \sqcup \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else } (\textbf{diverge}); \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} \quad & \\
& \langle \textbf{output}_l(e); \succ; \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{o_l(e,v)} \quad & \\
& \langle \succ; \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} \quad & \\
& \langle \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{t} \quad & \\
& \langle \square,\ m \uplus mst \rangle \\
= \quad & \text{found } \tilde{c}',\ m' \text{ and } mst' \\
& \langle \tilde{c}',\ m' \uplus mst' \rangle
\end{aligned}
$$

We show a step of $\langle\langle c,\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle$ that matches the trace of the inlined configuration:

$$
\begin{aligned}
& \langle\langle \textbf{output}_l(e),\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle & \\
\xrightarrow{o_l(v)} \quad & & \\
& \langle\langle \square,\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle & \\
= \quad & \langle\langle c',\ m'\rangle,\ \langle \Gamma',\ \Delta'\rangle\rangle & \text{found } c',\ \Gamma',\ \Delta'
\end{aligned}
$$

Case $mst(head(G) \sqcup \tilde{e}) \not\sqsubseteq l$:

$$
\begin{aligned}
& \langle \textbf{if } head(G) \sqcup \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else } (\textbf{diverge}); \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} \quad & \text{substitution for } \textbf{diverge} \\
& \langle \textbf{while } 1 \textbf{ do skip}; \succ; \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} \quad & \\
& \langle \textbf{skip}; \succ; \textbf{while } 1 \textbf{ do skip}; \succ; \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} \quad & \\
& \langle \succ; \textbf{while } 1 \textbf{ do skip}; \succ; \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} \quad & \\
& \langle \textbf{while } 1 \textbf{ do skip}; \succ; \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon}{}^{3} \quad & \\
& \langle \textbf{while } 1 \textbf{ do skip}; \succ; \checkmark,\ m \uplus mst \rangle
\end{aligned}
$$

This trace diverges, never generating a t event. Thus, this case does not fulfill our assumption.
End of case distinction.

Case OutputDefault :
According to [TR-OutputDefault] $\tilde{c} = \textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D); \checkmark$ where $S \vdash e \blacktriangleright \tilde{e}$.
Case distinction on $\Gamma(e)$:
Case $\Gamma(e) \sqsubseteq l$: **(32)**

We unroll the execution of $\langle \tilde{c},\, m \uplus mst \rangle$:

$$
\begin{array}{ll}
 & \langle \textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D); \checkmark,\, m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \quad \text{(32), } \Gamma(e) = mst(\tilde{e}) \text{ – lemma 1} \\
 & \langle \textbf{output}_l(e); \succ; \checkmark,\, m \uplus mst \rangle \\
\xrightarrow{o_l(e,v)} & \\
 & \langle \succ; \checkmark,\, m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \checkmark,\, m \uplus mst \rangle \\
\xrightarrow{t} & \\
 & \langle \square,\, m \uplus mst \rangle \\
= & \quad \text{found } \tilde{c}',\, mst' \\
 & \langle \tilde{c}',\, m' \uplus mst' \rangle
\end{array}
$$

We show that a step $\langle\langle c,\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle$ produces a matching output event:

$$
\begin{array}{lll}
 & \langle\langle \textbf{output}_l(e),\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle & \\
\xrightarrow{o_l(v)} & & \\
 & \langle\langle \square,\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle & \\
= & \langle\langle c',\, m'\rangle,\, \langle \Gamma',\, \Delta'\rangle\rangle & \text{found } c',\, m',\, \Gamma',\, \Delta'
\end{array}
$$

Case $\Gamma(e) \not\sqsubseteq l$: **(33)**

We unroll the execution of $\langle \tilde{c},\, m \uplus mst \rangle$ in order to find whether an output event is generated, and also $\tilde{c}',\, m',\, mst'$:

$$
\begin{array}{ll}
 & \langle \textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D); \checkmark,\, m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \quad \text{(33), } \Gamma(e) = mst(\tilde{e}) \text{ – lemma 1} \\
 & \langle \textbf{output}_l(D); \succ; \checkmark,\, m \uplus mst \rangle \\
\xrightarrow{o_l(D,D)} & \quad \text{observe the event} \\
 & \langle \succ; \checkmark,\, m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \checkmark,\, m \uplus mst \rangle \\
\xrightarrow{t} & \\
 & \langle \square,\, m \uplus mst \rangle \\
= & \quad \text{found } \tilde{c}',\, m',\, mst' \\
 & \langle \tilde{c}',\, m' \uplus mst' \rangle
\end{array}
$$

We show that the step of $\langle\langle c,\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle$ produces a matching output:

$$
\begin{array}{lll}
 & \langle\langle \textbf{output}_l(e),\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle & \\
\xrightarrow{o_l(D)} & & \\
 & \langle\langle \square,\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle & \\
= & \langle\langle c',\, m'\rangle,\, \langle \Gamma',\, \Delta'\rangle\rangle & \text{found } c',\, \Gamma',\, \Delta'
\end{array}
$$

End of case distinction.

Case OutputSuppress :

According to [TR-OutputSuppress] $\tilde{c} = \textbf{if } (head(G) \sqcup \tilde{e}) \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else skip;} \checkmark$ where $S \vdash e \blacktriangleright \tilde{e}$.

Case distinction on $lev(\Delta) \sqcup \Gamma(e)$:

Case $lev(\Delta) \sqcup \Gamma(e) \sqsubseteq l$: **(34)**

We unroll the execution of $\langle \tilde{c},\ m \uplus mst \rangle$:

$$
\begin{array}{ll}
 & \langle \textbf{if } (head(G) \sqcup \tilde{e}) \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else skip;} \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \quad (34),\ \Gamma(e) = mst(\tilde{e}) \text{ – lemma 1, } mst(head(G)) = head(\Delta) \\
 & \langle \textbf{output}_l(e)\textbf{;} \succ \textbf{;} \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{o_l(e,v)} & \\
 & \langle \succ \textbf{;} \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{t} & \\
 & \langle \square,\ m \uplus mst \rangle \\
= & \quad \text{found } \tilde{c}',\ mst' \\
 & \langle \tilde{c}',\ m' \uplus mst' \rangle
\end{array}
$$

We show a step of $\langle\langle c,\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle$ that matches the output of $\langle \tilde{c},\ m \uplus mst\rangle$:

$$
\begin{array}{lll}
 & \langle\langle \textbf{output}_l(e),\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle & \\
\xrightarrow{o_l(v)} & & \\
 & \langle\langle \square,\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle & \\
= & \langle\langle c',\ m'\rangle,\ \langle \Gamma',\ \Delta'\rangle\rangle & \text{found } c',\ m',\ \Gamma',\ \Delta'
\end{array}
$$

Case $lev(\Delta) \sqcup \Gamma(e) \not\sqsubseteq l$: **(35)**

We unroll the execution of $\langle \tilde{c},\ m \uplus mst\rangle$:

$$
\begin{array}{ll}
 & \langle \textbf{if } (head(G) \sqcup \tilde{e}) \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else skip;} \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \quad (35),\ \Gamma(e) = mst(\tilde{e}) \text{ – lemma 1, } mst(head(G)) = head(\Delta) \\
 & \langle \textbf{skip;} \succ \textbf{;} \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \succ \textbf{;} \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{t} & \\
 & \langle \square,\ m \uplus mst \rangle \\
= & \quad \text{found } \tilde{c}',\ mst' \\
 & \langle \tilde{c}',\ m' \uplus mst' \rangle
\end{array}
$$

We show a step of $\langle\langle c,\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle$ that doesn't produce an output, matching the trace of the inlined monitor:

$$
\begin{array}{lll}
 & \langle\langle \textbf{output}_l(e),\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle & \\
\longrightarrow & & \\
 & \langle\langle \square,\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle & \\
= & \langle\langle c',\ m'\rangle,\ \langle \Gamma',\ \Delta'\rangle\rangle & \text{found } c',\ m',\ \Gamma',\ \Delta'
\end{array}
$$

End of case distinction.

Case OutputDefault/Suppress :

According to [TR-OutputDefault/Suppress] $\tilde{c}$ = **if** $head(G) \sqsubseteq l$ **then** (**if** $\tilde{e} \sqsubseteq l$ **then output**$_l(e)$ **else output**$_l(D)$) **else skip;** ✓ where $S \vdash e \blacktriangleright \tilde{e}$.
Case distinction on $lev(\Delta)$:

Case $lev(\Delta) \sqsubseteq l$: (**36**) Case distinction on $\Gamma(e)$:

Case $\Gamma(e) \sqsubseteq l$: (**37**)

We unroll the execution of $\langle \tilde{c},\ m \uplus mst \rangle$:

$$
\begin{array}{rl}
 & \langle \textbf{if } head(G) \sqsubseteq l \textbf{ then } (\textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D)) \textbf{ else skip;} \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \quad (36),\ \Gamma(e) = mst(\tilde{e}) \text{ – lemma 1, } mst(head(G)) = head(\Delta) \\
 & \langle \textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D) \textbf{; } \succ \textbf{; } \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \quad (37),\ \Gamma(e) = mst(\tilde{e}) \text{ – lemma 1, } mst(head(G)) = head(\Delta) \\
 & \langle \textbf{output}_l(e) \textbf{; } \succ \textbf{; } \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{o_l(e,v)} & \\
 & \langle \succ \textbf{; } \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{t} & \\
 & \langle \square,\ m \uplus mst \rangle \\
= & \quad \text{found } \tilde{c}',\ m \text{ and } mst' \\
 & \langle \tilde{c}',\ m' \uplus mst' \rangle
\end{array}
$$

We show a step of $\langle\langle c,\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle$ that matches the output of the inlined monitor:

$$
\begin{array}{rll}
 & \langle\langle \textbf{output}_l(e),\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle & \\
\xrightarrow{o_l(v)} & & \\
 & \langle\langle \square,\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle & \\
= & & \text{found } c',\ \Gamma',\ \Delta' \\
 & \langle\langle c',\ m'\rangle,\ \langle \Gamma',\ \Delta'\rangle\rangle &
\end{array}
$$

Case $\Gamma(e) \not\sqsubseteq l$: (**38**)

We unroll the execution of $\langle \tilde{c},\ m \uplus mst \rangle$:

$$
\begin{array}{rl}
 & \langle \textbf{if } head(G) \sqsubseteq l \textbf{ then } (\textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D)) \textbf{ else skip;} \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \quad (36),\ \Gamma(e) = mst(\tilde{e}) \text{ – lemma 1, } mst(head(G)) = head(\Delta) \\
 & \langle \textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D) \textbf{; } \succ \textbf{; } \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \quad (38),\ \Gamma(e) = mst(\tilde{e}) \text{ – lemma 1, } mst(head(G)) = head(\Delta) \\
 & \langle \textbf{output}_l(D) \textbf{; } \succ \textbf{; } \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{o_l(D,D)} & \\
 & \langle \succ \textbf{; } \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \checkmark,\ m \uplus mst \rangle \\
\xrightarrow{t} & \\
 & \langle \square,\ m \uplus mst \rangle \\
= & \quad \text{found } \tilde{c}',\ m' \text{ and } mst' \\
 & \langle \tilde{c}',\ m' \uplus mst' \rangle
\end{array}
$$

End of case distinction.

We show a step $\langle\langle c,\ m\rangle,\ \langle \Gamma,\ \Delta\rangle\rangle$ with matching output:

$$
\begin{array}{lll}
 & \langle\langle \textbf{output}_l(e),\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle & \\
\xrightarrow{o_l(D)} & \langle\langle \square,\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle & \\
= & \langle\langle c',\, m'\rangle,\, \langle \Gamma',\, \Delta'\rangle\rangle & \text{found } c',\, \Gamma',\, \Delta'
\end{array}
$$

Case $lev(\Delta) \not\sqsubseteq l$: **(39)**

We unroll the execution of $\langle \tilde{c},\, m \uplus mst\rangle$:

$$
\begin{array}{ll}
 & \langle \textbf{if } head(G) \sqsubseteq l \textbf{ then } (\textbf{if } \tilde{e} \sqsubseteq l \textbf{ then output}_l(e) \textbf{ else output}_l(D)) \textbf{ else skip;} \checkmark,\, m \uplus mst\rangle \\
\xrightarrow{\epsilon} & \quad \text{(39), } \Gamma(e) = mst(\tilde{e}) \text{ – lemma 1, } mst(head(G)) = head(\Delta) \\
 & \langle \textbf{skip;} \succ \textbf{;} \checkmark,\, m \uplus mst\rangle \\
\xrightarrow{\epsilon} & \quad \text{(38), } \Gamma(e) = mst(\tilde{e}) \text{ – lemma 1, } mst(head(G)) = head(\Delta) \\
 & \langle \succ \textbf{;} \checkmark,\, m \uplus mst\rangle \\
\xrightarrow{\epsilon} & \\
 & \langle \checkmark,\, m \uplus mst\rangle \\
\xrightarrow{t} & \\
 & \langle \square,\, m \uplus mst\rangle \\
= & \quad \text{found } \tilde{c}',\, m' \text{ and } mst' \\
 & \langle \tilde{c}',\, m' \uplus mst'\rangle
\end{array}
$$

We show the step of $\langle\langle c,\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle$ that, too, doesn't produce any output.

$$
\begin{array}{lll}
 & \langle\langle \textbf{output}_l(e),\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle & \\
\longrightarrow & \langle\langle \square,\, m\rangle,\, \langle \Gamma,\, \Delta\rangle\rangle & \\
= & \langle\langle c',\, m'\rangle,\, \langle \Gamma',\, \Delta'\rangle\rangle & \text{found } c',\, \Gamma',\, \Delta'
\end{array}
$$

End of case distinction.

End of case distinction.

We show that for each of the cases above $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ is a valid instance of [TR-Stop] for $G' = G$, $S' = S$ and $I' = I$:

$$G, S, I \vdash \square \blacktriangleright \square$$

We show that $\langle \Gamma',\, \Delta'\rangle \cong (G', S', I', mst')$: by substituting $\Gamma' = \Gamma$, $\Delta' = \Delta$, $G' = G$, $S' = S$ and $I' = I$ and from the hypothesis of the lemma:

$$\langle \Gamma,\, \Delta\rangle \cong (G, S, I, mst)$$

Case $c = \succ\textbf{;}\, c_2$:

$$\tilde{c} = (\kappa(head(I), S, head(G))\textbf{;}\ \succ\textbf{;}\ \checkmark)\textbf{;}\ \tilde{c}_2$$

where $tail(G), S, tail(I) \vdash c_2 \blacktriangleright \tilde{c}_2$ **(40)**

We unroll the execution of $\tilde{c}$:

$$
\begin{array}{ll}
 & \langle \tilde{c},\, m \uplus mst\rangle \\
= & \\
\xRightarrow{\epsilon}\xrightarrow{t} & \langle (\kappa(head(I), S, head(G))\textbf{;}\ \succ\textbf{;}\ \checkmark)\textbf{;}\ \tilde{c}_2,\, m \uplus mst\rangle \\
 & \langle \tilde{c_2},\, m \uplus (mst \sqcup (head(I) \circ S^{-1}))\rangle \\
= & \langle \tilde{c}',\, m' \uplus mst'\rangle
\end{array}
\qquad
\begin{array}{l}
\\
\text{see base case } \succ\text{, semantics for } c_1\textbf{;}\, c_2 \\
\text{found } \tilde{c}',\, mst'
\end{array}
$$

Next, we demonstrate one step of $\langle\langle c,\ m\rangle,\ \langle\Gamma,\ \Delta\rangle\rangle$:

$$\begin{array}{ll} & \langle\langle \succ; c_2,\ m\rangle,\ \langle\Gamma,\ \Delta\rangle\rangle \\ \longrightarrow & \\ = & \langle\langle c_2,\ m\rangle,\ \langle\Gamma \sqcup mkFun(head(\Delta)),\ tail(\Delta)\rangle\rangle \\ & \langle\langle c',\ m'\rangle,\ \langle\Gamma',\ \Delta'\rangle\rangle \end{array} \quad \text{found } c',\ \Gamma' \textbf{ (41)},\ \Delta' \textbf{ (42)}$$

We show that $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ for $G' = tail(G)$, $S' = S$ and $I' = tail(I)$:

$$\frac{(40)}{tail(G), S, tail(I) \vdash c_2 \blacktriangleright \tilde{c_2}}$$

The proof of $\langle\Gamma',\ \Delta'\rangle \cong (G', S', I', mst')$ follows the correspondent proof in base case $\succ$.

Case $c = c_1; c_2$ **when** $c_1 \neq \succ$:

$\tilde{c} = \tilde{c_1}; \tilde{c_2}$ where $G, S, I \vdash c_1 \blacktriangleright \tilde{c_1}$ and $G, S, I \vdash c_2 \blacktriangleright \tilde{c_2}$ **(43)**

First, we unroll the execution of $\langle\tilde{c},\ m \uplus mst\rangle$ until the first t event.

$$\begin{array}{ll} & \langle\tilde{c_1}; \tilde{c_2},\ m \uplus mst\rangle \\ \overset{\alpha}{\Longrightarrow}\overset{t}{\longrightarrow} & \quad \text{found } \tilde{c}',\ m',\ mst' \\ & \langle\tilde{c}',\ m' \uplus mst''\rangle \end{array}$$

Observe, that according to the semantics of a sequence of commands and lemma 5, c' is either $c_1'; c_2$ or c_2, but not some reduction of c_2: c_2'. This is because, in order to reduce $c_1; c_2$ to c_2, c_1 needs to take a sequence of steps and end in a configuration containing a $\square$. Lemma 5 says that this could only happen if there is at least one t in the trace. Because the trace $\overset{\alpha}{\Longrightarrow}\overset{t}{\longrightarrow}$ contains exactly one t, c_2 is the farthest $\checkmark c_1\, c_2$ can go.

Case distinction on $\tilde{c}'$:

Case $\tilde{c_1}'; \tilde{c_2}$:

According to semantic rules for $c_1; c_2$ this could only happen if

$$\langle\tilde{c_1},\ m \uplus mst\rangle \overset{\alpha}{\Longrightarrow}\overset{t}{\longrightarrow} \langle\tilde{c_1}',\ m' \uplus mst'\rangle$$

Applying the induction hypothesis and semantic rules for $c_1; c_2$ we show a step of $\langle\langle c,\ m\rangle,\ \langle\Gamma,\ \Delta\rangle\rangle$:

$$\begin{array}{ll} & \langle\langle c_1; c_2,\ m\rangle,\ \langle\Gamma,\ \Delta\rangle\rangle \\ \overset{\sigma}{\longrightarrow} & \\ & \langle\langle c_1'; c_2,\ m'\rangle,\ \langle\Gamma',\ \Delta'\rangle\rangle \end{array} \quad \text{where } \exists G'', S'', I'' \mid G'', S'', I'' \vdash c_1' \blacktriangleright \tilde{c_1}'$$

From the induction hypothesis and by substituting $G' = G''$, $S' = S''$ and $I' = I''$ into $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ we get a valid instance of [TR-Seq]:

$$\frac{}{G', S', I' \vdash c_1'; c_2 \blacktriangleright \tilde{c_1}'; \tilde{c_2}}$$

Last, from the induction hypothesis and by substituting $G' = G''$, $S' = S''$ and $I' = I''$ we obtain $\langle\Gamma',\ \Delta'\rangle \cong (G', S', I', mst')$.

Case $\tilde{c_2}$:

According to semantic rules for $c_1; c_2$ this could only happen if

$$\langle\tilde{c_1},\ m \uplus mst\rangle \overset{\alpha}{\Longrightarrow}\overset{t}{\longrightarrow} \langle\square,\ m' \uplus mst'\rangle$$

Applying the induction hypothesis and semantic rules for $c_1; c_2$ we show a step of $\langle\langle c,\ m\rangle,\ \langle\Gamma,\ \Delta\rangle\rangle$:

$$\begin{array}{ll} & \langle\langle c_1; c_2,\ m\rangle,\ \langle\Gamma,\ \Delta\rangle\rangle \\ \overset{\sigma}{\longrightarrow} & \\ & \langle\langle c_2,\ m'\rangle,\ \langle\Gamma',\ \Delta'\rangle\rangle \end{array}$$

With the help of the induction hypothesis and the execution trace above, we can easily see that substituting $G' = G$, $S' = S$ and $I' = I$ in $G', S', I' \vdash c' \blacktriangleright \tilde{c}'$ gives us a valid transformation rule which is a part of our assumptions:

$$\frac{(43)}{G, S, I \vdash c_2 \blacktriangleright \tilde{c_2}}$$

And from the induction hypothesis we get $cfgm' \cong (G', S', I', mst')$:

End of case distinction.

End of structural induction.

□

Theorem 1 Observational equivalence.

For all $c, \tilde{c}, m, m', mst, \Gamma, \Delta$, if
$\langle\langle c,\ m\rangle,\ \langle\Gamma,\ \Delta\rangle\rangle \sim \langle\tilde{c},\ m \uplus mst\rangle$ then we have

(a) For all c', Γ', Δ', if $\langle\langle c,\ m\rangle,\ \langle\Gamma,\ \Delta\rangle\rangle \stackrel{\vec{\omega}}{\Longrightarrow} \langle\langle c',\ m'\rangle,\ \langle\Gamma',\ \Delta'\rangle\rangle$ then there is $\vec{\beta}$ such that $\vec{\beta} \rhd \vec{\omega}$ and $\langle c,\ m \uplus mst\rangle \stackrel{\vec{\beta}}{\Longrightarrow} \langle\tilde{c}',\ m' \uplus mst'\rangle$ and $\langle\langle c',\ m'\rangle,\ \langle\Gamma',\ \Delta'\rangle\rangle \sim \langle\tilde{c}',\ m' \uplus mst'\rangle$.

(b) For all $\tilde{c}', mst'$, if $\langle\tilde{c},\ m \uplus mst\rangle \stackrel{\vec{\beta}}{\Longrightarrow}\!\!\!\!\!\!\rightarrow\ \langle\tilde{c}',\ m' \uplus mst'\rangle$ then there is $\vec{\omega}$ such that $\vec{\beta} \rhd \vec{\omega}$ and $\langle\langle c,\ m\rangle,\ \langle\Gamma,\ \Delta\rangle\rangle \stackrel{\vec{\omega}}{\Longrightarrow}$ $\langle\langle c',\ m'\rangle,\ \langle\Gamma',\ \Delta'\rangle\rangle$ and $\langle\langle c',\ m'\rangle,\ \langle\Gamma',\ \Delta'\rangle\rangle \sim \langle\tilde{c}',\ m' \uplus mst'\rangle$.

Proof: Part **(a)** Induction on number of steps in $\langle\langle c,\ m\rangle,\ \langle\Gamma,\ \Delta\rangle\rangle \stackrel{\vec{\omega}}{\Longrightarrow} \langle\langle c',\ m'\rangle,\ \langle\Gamma',\ \Delta'\rangle\rangle$:

Case 0 steps : Immediate.

Case n+1 steps : Consider a VM trace $\langle\langle c,\ m\rangle,\ \langle\Gamma,\ \Delta\rangle\rangle \stackrel{\vec{\omega}'}{\Longrightarrow} \langle\langle c'',\ m''\rangle,\ \langle\Gamma'',\ \Delta''\rangle\rangle$ of length n, followed by a step

$$\langle\langle c'',\ m''\rangle,\ \langle\Gamma'',\ \Delta''\rangle\rangle \stackrel{\sigma}{\longrightarrow} \langle\langle c',\ m'\rangle,\ \langle\Gamma',\ \Delta'\rangle\rangle$$

where σ is either a β or an empty label (unlabeled transition). So the trace of length $n+1$ has the form

$$\langle\langle c,\ m\rangle,\ \langle\Gamma,\ \Delta\rangle\rangle \stackrel{\vec{\omega}'}{\Longrightarrow}\!\!\!\!\!\!\rightarrow\ \langle\langle c',\ m'\rangle,\ \langle\Gamma',\ \Delta'\rangle\rangle$$

By induction on the trace $\stackrel{\vec{\omega}'}{\Longrightarrow}$ there is a corresponding transformed trace $\langle\tilde{c},\ m \uplus mst\rangle \stackrel{\vec{\beta}'}{\Longrightarrow} \langle\tilde{c}'',\ m'' \uplus mst''\rangle$ such that $\langle\langle c'',\ m''\rangle,\ \langle\Gamma'',\ \Delta''\rangle\rangle \sim \langle\tilde{c}'',\ m'' \uplus mst''\rangle$ and $\vec{\beta}' \rhd \vec{\omega}'$. We apply lemma 4 which yields $\tilde{c}'$ and mst' such that $\langle\tilde{c}'',\ m'' \uplus mst''\rangle \stackrel{\alpha}{\Longrightarrow}\!\!\!\!\!\!\rightarrow\ \langle\tilde{c}',\ m' \uplus mst'\rangle$ and $\langle\langle c',\ m'\rangle,\ \langle\Gamma',\ \Delta'\rangle\rangle \sim \langle\tilde{c}',\ m' \uplus mst'\rangle$ and $\alpha \triangleright \sigma$. Using $\vec{\beta}' \rhd \vec{\omega}'$ we get $\vec{\beta}'\alpha \rhd \vec{\omega}'\sigma$ completes the argument for the matching trace

$$\langle\tilde{c},\ m \uplus mst\rangle \stackrel{\vec{\beta}'}{\Longrightarrow}\!\!\!\!\stackrel{\alpha}{\Longrightarrow}\!\!\!\!\!\!\rightarrow\ \langle\tilde{c}',\ m' \uplus mst'\rangle$$

End of induction.

Part **(b)** Induction on number of t-events in $\vec{\beta}$:

Case 0 t-events : We can apply lemma 6 to the entire trace $\langle\tilde{c},\ m \uplus mst\rangle \stackrel{\vec{\beta}}{\Longrightarrow}\!\!\!\!\!\!\rightarrow\ \langle\tilde{c}',\ m' \uplus mst'\rangle$ to obtain the result. Observe that we can apply the lemma directly because $\stackrel{\vec{\beta}}{\Longrightarrow}\!\!\!\!\!\!\rightarrow$ contains exactly one t.

Case n+1 t-events : So the trace has the form

$$\langle\tilde{c},\ m \uplus mst\rangle \stackrel{\vec{\beta}'}{\Longrightarrow}\!\!\!\!\!\!\rightarrow\ \langle\tilde{c}'',\ m'' \uplus mst''\rangle \stackrel{\alpha}{\Longrightarrow}\!\!\!\!\!\!\rightarrow\ \langle\tilde{c}',\ m' \uplus mst'\rangle$$

where $\vec{\beta}'t$ has n t-events. By induction on the $\vec{\beta}'t$ trace, there is a VM-trace $\langle\langle c,\ m\rangle,\ \langle\Gamma,\ \Delta\rangle\rangle \stackrel{\vec{\omega}'}{\Longrightarrow}$ $\langle\langle c'',\ m''\rangle,\ \langle\Gamma'',\ \Delta''\rangle\rangle$ such that

$$\langle\langle c'',\ m''\rangle,\ \langle\Gamma'',\ \Delta''\rangle\rangle \sim \langle\tilde{c}'',\ m'' \uplus mst''\rangle$$

and $\vec{\beta}' \rhd \vec{\omega}'$. (Or, if you like, $\vec{\beta}'t \rhd \vec{\omega}'$ which is equivalent by definition of $\rhd$.) So we can apply lemma 6 to the next steps, $\langle\tilde{c}'',\ m'' \uplus mst''\rangle \stackrel{\alpha}{\Longrightarrow}\!\!\!\!\!\!\rightarrow\ \langle\tilde{c}',\ m' \uplus mst'\rangle$, which yields c', σ and Γ' and Δ' such that

$$\langle\langle c'',\ m''\rangle,\ \langle\Gamma'',\ \Delta''\rangle\rangle \stackrel{\sigma}{\longrightarrow} \langle\langle c',\ m'\rangle,\ \langle\Gamma',\ \Delta'\rangle\rangle$$

and $\langle\langle c',\ m'\rangle,\ \langle\Gamma',\ \Delta'\rangle\rangle \sim \langle\tilde{c}',\ m' \uplus mst'\rangle$ and $\alpha \triangleright \sigma$. So we have the matching trace

$$\langle\langle c,\ m\rangle,\ \langle\Gamma,\ \Delta\rangle\rangle \stackrel{\vec{\omega}'}{\Longrightarrow} \langle\langle c'',\ m''\rangle,\ \langle\Gamma'',\ \Delta''\rangle\rangle \stackrel{\sigma}{\longrightarrow} \langle\langle c',\ m'\rangle,\ \langle\Gamma',\ \Delta'\rangle\rangle$$

with coupled final configurations. Combining $\vec{\beta}' \rhd \vec{\omega}'$ with $\alpha \triangleright \sigma$ we get $\vec{\beta}'\alpha \rhd \vec{\omega}'\sigma$.

End of induction.

□