Confidentiality and Integrity with Untrusted Hosts

Steve Zdancewic Andrew C. Myers

Cornell University

Abstract

Several *security-typed languages* have recently been proposed to enforce security properties such as confidentiality or integrity by type checking. We propose a new security-typed language, SPL@, that addresses two important limitations of previous approaches.

First, existing languages assume that the underlying execution platform is trusted; this assumption does not scale to distributed computation in which a variety of differently trusted hosts are available to execute programs. Our new approach, *secure program partitioning*, translates programs written assuming complete trust in a single executing host into programs that execute using a collection of variously trusted hosts to perform computation. As the trust configuration of a distributed system evolves, this translation can be performed as necessary for security.

Second, many common program transformations do not work in existing security-typed languages; although they produce equivalent programs, these programs are rejected because of *apparent* information flows. SPL@ uses a novel mechanism based on *ordered linear continuations* to permit a richer class of program transformations, including secure program partitioning.

This report is the technical companion to [ZM00]. It contains expanded discussion and extensive proofs of both the soundness and noninterference theorems mentioned in Section 3.3 of that work.

1 Introduction

Stronger protection for the confidentiality (secrecy, privacy) and integrity of data is becoming increasingly important, because programs containing untrusted code and networks containing untrusted hosts are the rule rather than the exception. An approach that has recently been under investigation is the idea of enforcing static information flow control [DD77] via type systems in which the types include a security attribute [PO95, VSI96, ML97, HR98, SV98, Mye99, ABHR99]. We call these languages *security-typed languages*.

Information flow control is attractive because, unlike ordinary access control, it can enforce *end-to-end security policies*. These policies require that data be protected even if it is used for computation by untrusted code: for example, a simple end-to-end confidentiality policy might require that a particular data item is readable only by a particular set of users—regardless of what code manipulates the data information derived from it. Thus, security is protected even when a program comprises many separately developed components.

End-to-end security policies are particularly attractive given current trends in computation. Increasingly, computations are collaborations between principals with distinct security interests, and are distributed across a set of hosts. Ordinary web transactions already fit this description, but we can expect to see more complex, interesting examples with the increasing use of business-to-business transactions.

E-mail: {zdance,andru}@cs.cornell.edu

This work was sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-99-1-0533. The U.S. Governement is authorized to reproduce and distribute reprints for Governement purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

Because of the support for end-to-end security policies, static information flow control would seem to be ideal for enforcing the security of distributed computation. However, existing languages ignore an important aspect of distributed computation: the hosts on which computation is performed are not universally (or equally) trusted by the participants in the computation. These hosts may leak or damage data even though the programs that run on them are correct.

For security, any distributed computation should be partitioned so that no participant is required to place undue trust in a host. For example, data that is confidential to a particular principal should not be transferred to a host that is not trusted by that principal. This observation applies even to conventional security mechanisms, of course, where the partitioning is performed manually, and computations spanning multiple hosts are usually described as separate programs communicating over the network. However, this idea of program partitioning can be used to guide the automated application of information flow control to decentralized, distributed systems.

This paper proposes a new approach to security in distributed systems, called *secure program partitioning*. A program is written in a language in which security policies can be embedded as type annotations. Although the program is to run on a set of communicating hosts, it need not contain explicit information about where code runs or about inter-host communication. Given a set of hosts that are available for execution, a program is then transformed automatically by partitioning its code across the available hosts while ensuring that security invariants are maintained by each partition. The key invariant maintained by the partitioning transformation is that a host manipulates some data only if the host is sufficiently trusted by all principals that have an interest in the privacy or integrity of that data.

This work makes several contributions towards realizing this new model of security enforcement:

- It introduces the security-typed language SPL@ (pronounced "splat"), which has both a *unlocated* variant in which hosts are not mentioned and a *located* variant in which the host on which each piece of code runs is explicitly stated. SPL@'s type system validates the information flows within programs of either form and has been shown sound for the language with respect to a simple operational semantics.

- We give type-driven rewrite rules that transform a program from the unlocated form to the located form, partitioning the code among various hosts, while preserving typing. Transfers of control between the hosts are automatically introduced to ensure that security constraints are satisfied.

- SPL@ is the first security-typed language with support for *continuations*. Continuations are an important linguistic feature for an intermediate language [FSDF93] because they subsume a variety of control structures and closely model low-level computation: in this language, they are the only control structure other than conditionals. However, continuations make acceptably precise bounds on implicit information flow difficult to obtain. This language uses *ordered linear continuations* in a novel way to recover this precision.

- SPL@ introduces support for security labels that protect both confidentiality and integrity. It also permits selective declassification [ML97, ML00] of both security label components. We demonstrate a coupling between confidentiality and integrity in the presence of such a declassification mechanism.

This work does not address termination channels or timing channels. Furthermore, communication between hosts is assumed to be protected and authenticated by encryption; for example, pairwise private-key encryption will suffice. Section 2.9 discusses assumptions about the run-time environment further.

Programs are single-threaded, so various covert information flows that can arise in a concurrent language are not addressed. Termination channels and timing channels are ignored, so programs can leak one bit of information by failing to terminate, or by controlling their run time. Prevention of information flow is also subject to certain assumptions about the distributed computing environment.

The remainder of the paper is organized as follows. Section 2 overviews SPL@'s features at a high level. It then presents the security model and implementation issues. Section 3 illustrates SPL@'s type system and

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\text{Labels}} \\
\mathsf{P} & = & \text{Principals} \\
\mathcal{L} & = & \{\langle A, B\rangle \mid A, B \subseteq \mathsf{P}\} \\
\ell, \mathsf{pc} & \in & \mathcal{L} \\
\top & \overset{\text{def}}{=} & \langle \mathsf{P}, \emptyset\rangle \\
\bot & \overset{\text{def}}{=} & \langle \emptyset, \mathsf{P}\rangle \\
\\
\multicolumn{3}{l}{\text{Hosts}} \\
h & \in & \mathcal{H} \\
\mathsf{TP}(h) & \subseteq & \mathsf{P}
\end{array}
$$

Figure 1: Syntax for labels and hosts

operational semantics, and Section 4 contains a discussion of the language's formal properties, including the proofs of soundness and noninterference. Section 5 describes the partitioning translation. Sections 6 and 7 finish up with related work and our conclusions.

2 The SPL@ language

This section presents an informal overview of the SPL@ language and sketches how the language can be used to write programs that may then be securely partitioned. Formal definition of the typing rules and operational semantics are given in Section 3 and Appendix A.

2.1 Security labels

As in other security-typed languages [VSI96, ML97, SV98], *secure values*, written bv_ℓ, have both an ordinary value component bv (for *base value*) and an attached *security label* ℓ, expressing the security requirements (confidentiality, integrity or both) for that value. The security labels do not need to be maintained at run time, because all run-time label checks will succeed in a well-typed program.

As shown in Figure 1, the lattice of security labels, $\mathcal{L}$, consists of pairs of principals, written $\langle A, B\rangle$. We use the meta-variables ℓ and pc to range over elements of the security lattice, using the latter to suggest that the label is annotating the program counter, as described below. Labels contain two components: a confidentiality component, A, that controls who is able to view the labeled data, and an integrity component, B, that controls who is able to affect the labeled data. The ordering $\langle A, B\rangle \sqsubseteq \langle A', B'\rangle$ means that a value can be securely relabeled from the label on the left to that on the right. This is the case if the privacy component A' restricts the dissemination of the value at least as much as A did (otherwise the value could go to new places as the result of a relabeling), and the integrity component B' claims at most the integrity that B does (otherwise the integrity of the value might be claimed to increase as the result of relabeling).

The privacy and integrity components of the label are a simplified version of those in the decentralized label model [ML00], retaining only the notion of ownership needed for selective declassification. Both the privacy and integrity components are sets of *principals* drawn from P, which represent users, groups, roles, and other entities with authority. In a privacy component, the set contains the principals that have a stake in the data: they are *owners* or *sources* of the data. The principals are similar to *categories* in classic multilevel security policies [FLR77], except for their interaction with declassification. The ordering on privacy components is ordinary set inclusion: the most restrictive privacy component is the set of all principals P, in which every principal wishes to control the flow of the data; the least restrictive privacy component is the empty set, in which no principal cares about the data.

An integrity component is the set of principals that believe the data not to be corrupted. The ordering relation is reverse set inclusion, because the most restrictive integrity component is the empty set, in which no principal believes that the data has integrity. Such data can only be used in contexts where no integrity is required. The least restrictive integrity is the set of all principals P; since every principal vouches for the integrity of the data, it can be used anywhere.

Given this label structure, the ordering on the security label product lattice is as follows:

$$\frac{A \subseteq A' \quad B \supseteq B'}{\langle A, B\rangle \sqsubseteq \langle A', B'\rangle}$$

The set of all principals is written as P; given this label ordering, the least possible label $\bot$ and the greatest possible label $\top$ are defined as shown in the figure. We use $\sqcap$ and $\sqcup$ to denote the lattice meet and join operators, respectively.

In the located version of the language, hosts may be named explicitly. For each host h in the finite set of all hosts $\mathcal{H}$, $\mathsf{TP}(h)$ a set of principals that trusts that h. This set is used statically to determine what data and computation can be placed at h.

2.2 The security context

At any given point during execution, the executing code possesses some *authority*, a set of principals for which the code is authorized to act. Authority is declared in the program and statements that require authority, such as the declassify primitive, are checked statically against the declared authority.

In addition to authority, each program point has an associated *program counter label*, pc, that indicates the security label of information on which the program counter depends. Instructions executed at a program counter labeled pc are restricted so they can only transmit information to memory locations with labels *higher* than pc in $\mathcal{L}$. The program counter label allows *implicit flows* of information to be controlled conservatively [Mye99].

Together, the authority A and program counter pc constitute a *security context* for the evaluating code. We write this context as $[A, \mathsf{pc}]$.

2.3 SPL@ syntax

The syntax of both the located and unlocated variants of SPL@ is given in Figure 2. The only syntactic difference between the two is that the host annotation on code ($@h$) is omitted in the unlocated variant. An unlocated program is always equivalent to the same program in the located language, in which annotations locate all code at a universally trusted host ($\mathsf{TP}(h) = \mathsf{P}$).

The language is call-by-value, uses continuation-passing style, and includes imperative features intended to be suitable for intermediate stages in a translation of a high-level language such as Jif to lower-level, distributed code.

SPL@ permits only primitive expressions, $prim$, that manipulate constant values v. Expressions, e, function as statements that return no value. They consist of nested **let** or **letlin** expressions that introduce new variables in a sequence terminated by a transfer of control.

The language has state: programs may create, observe, and modify locations in the store using the **ref**, **deref**, and **set** primitives respectively. In the located language, the store locations are situated at the host where the **ref** primitive was executed. Accesses to a location cause appropriate communication between the current host and the host of the location. When a program terminates execution by evaluating the **halt** expression, the final value and also the contents of the store are assumed to be visible to principals in accordance with the privacy components of their security labels.

Types

$$\begin{array}{rcl}
\tau & ::= & \mathsf{int} \mid 1 \mid \sigma\ \mathsf{ref} \mid [A,\ell](\sigma,\kappa)\ \mathsf{cont} \\
\sigma & ::= & \tau_\ell \\
\kappa & ::= & 1 \mid (\sigma,\kappa)\ \mathsf{lcont}
\end{array}$$

Contexts

$$\begin{array}{rcl}
\Gamma & ::= & \bullet \mid \Gamma, x\!:\!\sigma \\
K & ::= & \bullet \mid K, y\!:\!\rho
\end{array}$$

Values

$$\begin{array}{rcl}
bv & ::= & n \mid \langle\rangle \mid L^\sigma @h \\
 & & \mid \mathbf{cont}@h\ f_B\ [A,\ell](x\!:\!\sigma, y\!:\!\kappa) = e \\
v & ::= & bv_\ell \mid x \\
lv & ::= & \langle\rangle \mid \mathbf{lcont}@h[A,\ell](x\!:\!\sigma, y\!:\!\kappa) = e
\end{array}$$

Primops

$$\begin{array}{rcl}
prim & ::= & v \mid v \oplus v \mid \mathbf{deref}(v) \\
 & & \mid \mathbf{declassify}(v,\ell)
\end{array}$$

Expressions

$$\begin{array}{rcl}
e & ::= & \mathbf{let}\ x = prim\ \mathbf{in}\ e \\
 & & \mid \mathbf{let}\ x = (\mathbf{ref}(\sigma)\ v)_\ell\ \mathbf{in}\ e \\
 & & \mid \mathbf{set}\ v := v\ \mathbf{in}\ e \\
 & & \mid \mathbf{if0}\ v\ \mathbf{then}\ e\ \mathbf{else}\ e \\
 & & \mid \mathbf{letlin}\ y = lv\ \mathbf{in}\ e \\
 & & \mid \mathbf{goto}\ v\ v\ lv \\
 & & \mid \mathbf{lgoto}\ lv\ v\ lv \\
 & & \mid \mathbf{actsFor}\ (A)\ \mathbf{then}\ e\ \mathbf{else}\ e \\
 & & \mid \mathbf{setpc}\ (\ell)\ \mathbf{in}\ e \\
 & & \mid \mathbf{halt}_\sigma\ v
\end{array}$$

Figure 2: Syntax

2.4 Types

The types of the language fall into two syntactic classes: *security types* (ranged over by σ) and *linear types* (ranged over by κ). Security types are the types of ordinary values and consist of a simple type component (τ) annotated with a security label ℓ. If $\sigma = \tau_\ell$, then by definition, $\mathsf{label}(\sigma) = \ell$. In a well-typed program, the run-time label of the value of an expression is always bounded above in the lattice by the label component of that expression's security type.

The simple types are the types of integer values, int, a unit type 1, references to a security type, $\sigma\ \mathsf{ref}$, and the type of ordinary continuations, $[A, \mathsf{pc}](\sigma, \kappa)\ \mathsf{cont}$. Correspondingly, an ordinary base value (bv) is an integer n, a unit $\langle\rangle$, a memory location $L^\sigma@h$, or a continuation **cont**$@h\ f_B\ [A, \mathsf{pc}](x : \sigma, y : \kappa) = e$. For simplicity of presentation, we have omitted even standard type constructors such as pairs and sums, although they could be added easily.

To make the examples in this paper more readable, the labels on unit values, which can transmit no information, often omitted. To avoid having to parenthesize location values like this: $(L^\sigma@h)_\ell$, we write $L^\sigma_\ell@h$ instead.

An ordinary continuation value, denoted by **cont**$@h\ f_B\ [A, \mathsf{pc}](x : \sigma, y : \kappa) = e$, is a piece of code (the expression e) located at host h. It accepts a nonlinear argument of type σ and a linear argument of type κ. The code of the continuation has been granted authority at compile time to act for the principals in the set B; presumably those principals have reviewed this code and deemed that it does not violate their security. The notation $[A, \mathsf{pc}]$ indicates that this continuation may only be invoked from a calling context in which authority is known to be at least A and in which the program counter is labeled with at most pc. These restrictions are indicated by its type, $[A, \mathsf{pc}](\sigma, \kappa)\ \mathsf{cont}$. Ordinary continuations may be recursive: the body of the continuation may invoke itself using the name f.

2.5 Ordered linear continuations

Continuations are particularly useful for an intermediate language because they subsume a variety of other control constructs. However, there is a difficulty in supporting continuations in a security-typed language. With straightforward, sound typing rules, implicit flows arising from branches in the control structure can never be eliminated. Consider the following source-level program, which tests the high-security integer x_H and eventually performs the safe assignment to the low-security reference z:

$$\textbf{if0}\ \mathsf{x}_H\ \textbf{then}\ \{\mathsf{y} := 1;\}\ \textbf{else}\ \{\mathsf{y} := 2;\};\ \mathsf{z} := 3;$$

Informally, a CPS translation of the above code results in something like the following, where k is the continuation of the **if0** statement:

$$\textbf{let}\ \mathsf{k} = (\lambda().\ \mathsf{z} := 3)\ \textbf{in}\ \textbf{if0}\ \mathsf{x}_H\ \textbf{then}\ \mathsf{y} := 1; \mathsf{k}()\ \textbf{else}\ \mathsf{y} := 2;\ \mathsf{k}()$$

The problem is that the invocation of the continuation k appears within the branches of the conditional, so to avoid an implicit flow, the simple typing rules of both SLam [HR98] and Jif [Mye99] conservatively require that k's body not write to z.

The SPL@ language introduces *ordered linear continuations* to avoid this loss of precision. Linear continuations differ from ordinary continuations in that, once introduced, they must be invoked exactly once in all subsequent control flow paths (unless the program diverges, in which case the linear continuation may never be invoked). This restriction prevents two different branches in a program from leaking information by discarding or repeating a continuation. To avoid the possibility that information could be leaked by invoking two linear continuations in different orders, SPL@ enforces a stronger property—if used at all, linear continuations that are simultaneously in scope must be used in the same order in every possible execution of the program. In conjunction, these two features allow linear continuations to capture the security context in which they are declared and restore that security context when they are invoked.

```
letlin k = lcont[∅, pc](x:1, y:1) =
      set z := 3 in              (*pc*)
      halt_1 ⟨⟩
in
if0 x_H then
      set y := 1 in              (*pc ⊔ H*)
      lgoto k ⟨⟩ ⟨⟩
else
      set y := 2 in              (*pc ⊔ H*)
      lgoto k ⟨⟩ ⟨⟩
```

Figure 3: Using ordered linear continuations to improve precision.

Figure 3 shows the example translated using a linear continuations to describe the merge-point of the two branches, which is allowed because the branches both terminate in jumps to the continuation k. At k's invocation the program counter contains no information about which branch has been taken, so its label can be restored to what it was at the declaration k. The comments indicate the label of the program counter, assuming it initially starts at label pc. SPL@'s type system enforces a stack discipline for the ordering of linear continuations which can reflect the lexical structure of a higher-level source language. Branches encode information about conditional values into the program counter; linear continuations undo this effect by forcing control back to a single point. A linear continuation is a first-class merge point in the control-flow graph that controls implicit flows with greater precision.

There are two types of linear values (lv): linear unit, used only to indicate the lack of a linear argument to a continuation, and linear continuations themselves. Because a linear continuation restores a captured security context, there are no constraints imposed on that context at the point of invocation. Therefore the types of linear continuations don't need to mention a security context, and have the form (σ, κ) lcont. Linear values and ordinary values are have distinct syntax in SPL@, and are never allowed in the same syntactic position. For example, the first argument to a continuation must be an ordinary, nonlinear value of type σ and the second must be a linear value of linear type κ. Because SPL@ contains explicit syntax for writing down linear continuations, and their intended meaning is to capture a security context, we must restrict initial programs so that all linear continuations are introduced via the **letlin** construct. (Otherwise it would be possible to write down linear continuations that don't capture any appropriate context and hence violate security.)

Linear continuations have another useful property: unlike ordinary values, linear values do not require security annotations, because there is no way to communicate information by selecting among several choices of linear continuations.

2.6 Security invariants

The type system of SPL@ maintains a number of security invariants that are desirable in a system of heterogeneously trusted machines. Type-safety in SPL@ guarantees more than traditional memory safety and progress; there are also guarantees about authority and security.

In general, integrity constraints on data with label ℓ originating *from* a host h are captured by the requirement that $\langle \emptyset, \mathsf{TP}(h) \rangle \sqsubseteq \ell$, that is, the set of principals who trust the data labeled with ℓ is at most the set of principals who trust h. Dually, the confidentiality requirement on data labeled ℓ flowing *to* a host h is expressed by $\ell \sqsubseteq \langle \mathsf{TP}(h), \emptyset \rangle$, which says that the set of principals who own the data are included in those principals who trust the host h. The integrity constraints are enforced when data is sent

```
let secret = $42_{\langle\{\mathsf{root}\},\mathsf{P}\rangle}$ in
...
let check = cont@$h_1$ check$_{\{\}}[\emptyset,\bot](\mathsf{x}:\mathsf{int}_L, \mathsf{y}:1)$ = (
   let unsafe = cont@$h_2$ unsafe$_{\{\mathsf{root}\}}[\emptyset, L](\mathsf{z}:1, \mathsf{y}:1)$ = (
      let s = declassify(secret, $\langle\emptyset,\{\mathsf{root}\}\rangle \sqcup L$) in        $(*\ \mathsf{pc} = L\ *)$
      halt$^{\langle\emptyset,\{\mathsf{root}\}\rangle\sqcup L}$ s
   ) in
   if0 x then goto unsafe $\langle\rangle$ $\langle\rangle$                                $(*\ \mathsf{pc} = L\ *)$
      else halt$^{\langle\emptyset,\{\mathsf{root}\}\rangle\sqcup L}$ 0
)
```

Figure 4: The need for pc integrity

from a host: the **goto**, **lgoto**, **halt**, and assignment operations. Conversely, privacy constraints appear where information can arrive from another host: the rule for values (which arrive via substitution), and in the declaration rules for both kinds of continuations. These constraints apply to the label of the program counter; so $\langle\emptyset, \mathsf{TP}(h)\rangle \sqsubseteq \mathsf{pc} \sqsubseteq \langle\mathsf{TP}(h), \emptyset\rangle$. For a security label ℓ that satisfies these conditions, we write $\vdash \ell$ **ok** $@h$.

The requirement that $\langle\emptyset, \mathsf{TP}(h)\rangle \sqsubseteq \mathsf{pc}$ can be seen by thinking of the operational behavior of the **goto**: it effectively transfers control to a (possibly) different host machine. We think of the values that leave h as being stamped with the security label of the program counter at the point of the call (this is made explicit in the operational semantics). Because host h is trusted by only the principals $\mathsf{TP}(h)$, the values leaving h should never have more integrity, thus the program counter is bounded below by the integrity constraint $\langle\emptyset, \mathsf{TP}(h)\rangle \sqsubseteq \mathsf{pc}$.

The type system also maintains the invariant that in every security context, $A \subseteq \mathsf{TP}(h)$. This guarantees that the authority used in the piece of code running on h does not exceed the maximum authority available to h.

Some simple security properties can be read directly from the operational semantics: for instance, a value with a high security label is never written to a memory location readable by low-security hosts. It is clear that full SPL@ does not exhibit the *noninterference* property introduced by Goguen and Meseguer [GM82, GM84] and studied in a programming languages setting more recently [VSI96, SV98, HR98], because of the presence of **declassify** and **setpc**. Because of the limitations of noninterference, these unsafe operators are important for implementing many programs and are therefore important to model in the language.

2.7 An interaction between privacy and integrity

One subtle aspect of secure program partitioning is that the **declassify** operator introduces a coupling between the integrity and privacy components of security labels in the program. Clearly, the **declassify** operation is unsafe for the principals whose security is weakened by it; it therefore requires that the program has the authority of any such principal p at the point of its use. The security invariant $A \subseteq \mathsf{TP}(h)$ then guarantees that the declassification takes place on a suitably trusted host.

However, there is a another invariant to be maintained. Because the **declassify** operation is unsafe for the principal p, the decision to perform the operation should be based on data that p trusts. This condition is expressed by requiring that the program counter label's integrity component include p, which ensures that the code that affects the decision to declassify is run only on hosts trusted by p.

Figure 4 shows why this constraint is needed. The program in the figure contains a piece of code named check that tests its argument x and uses its value to decide whether to invoke some encapsulated code named unsafe. The unsafe code uses the authority of the principal root to release the secret value contained in the variable secret (note that the privacy component of the label is declassified from $\{\mathsf{root}\}$ to $\emptyset$). Left unspecified in the program are the label L and the two hosts h_1 and h_2. It is clear that the host h_2 must be a

```
int h1 = 0;
int h2 = 0;
int l1 = 0;
int l2 = 0;
h1 = h2;
h2 = h2 + h1;
l1 = l2;
```

Source Program

```
let h1 = ref (int_H) 0 in
let h2 = ref (int_H) 0 in
let l1 = ref (int_L) 0 in
let l2 = ref (int_L) 0 in
let t1 = deref(h2) in
set h1 := t1 in
let t2 = deref(l1) in
let t3 = t1 + t2 in
set h2 := t3 in
let t4 = deref(l2) in
set l1 := t4 in
   halt_L t4
```

SPL@ Program

Figure 5: A simple program

```
let h1 = ref (int_H) 0 in
let h2 = ref (int_H) 0 in
letlin lc1 = lcont@low [∅, L](x:1, y:1) = (
   let l1 = ref (int_L) 0 in
   let l2 = ref (int_L) 0 in
   letlin hc1 = lcont@high [∅, L](x:1, y:1) = (
      let t1 = deref(h2) in
      set h1 := t1 in
      let t2 = deref(l1) in
      let t3 = t1 + t2 in
      set h2 := t3 in
      letlin lc2 = lcont@low [∅, L](x:1, y:1) = (
         let t4 = deref(l2) in
         set l1 := t4 in
            halt_L t4
      ) in
         lgoto lc2 ⟨⟩ ⟨⟩
   ) in
      lgoto hc1 ⟨⟩ ⟨⟩
) in
   lgoto lc1 ⟨⟩ ⟨⟩
```

Figure 6: The simple program partitioned into located form

host that the principal root trusts. What is less obvious is that root must also trust h_1. If the code of check is run on a host that root does not trust, then that host is in charge of determining whether to invoke unsafe and may do so improperly. Because the integrity component of pc must include at least the principal root, according to the condition above, so must the integrity component of L. Therefore, h_1 must be trusted by root. This example shows that integrity must be maintained in order to protect privacy.

2.8 A translation example

Figure 5 gives a small example of a C program and its equivalent in the unlocated version of SPL@. (We have omitted the label on the **ref** operations; assume it is $\perp$.) The label $H = \langle\{\mathsf{p}_1, \mathsf{p}_2\}, \{\mathsf{p}_1\}\rangle$ represents a high-security level for secret data, and the label $L = \langle\emptyset, \{\mathsf{p}_1\}\rangle$ is low-security.

The same code after partitioning might look as shown in Figure 6. Note that code that can run on the machine low has been moved there. In this simple example, the host high is trusted by $\{\mathsf{p}_1, \mathsf{p}_2\}$ and low by $\{\mathsf{p}_1\}$. Therefore the data labeled with H is not allowed to move to low ($\not\vdash H$ **ok** $@\mathsf{low}$). Execution begins on high, and proceeds as follows: $\mathsf{high} \to \mathsf{low} \to \mathsf{high} \to \mathsf{low}$.

2.9 Implementation issues

This work assumes a secure network: messages between hosts cannot be intercepted or damaged. A secure network can be constructed atop an insecure network through well-known encryption techniques. However, the security of SPL@ also requires that some dynamic checks be performed by the run-time system when inter-host actions, such as remote reads or writes, are performed.

To invoke an ordinary continuation with type $([A', \mathsf{pc}'](\tau, \sigma)\ \mathsf{cont})_\ell$, the security context $[A, \mathsf{pc}]$ at the point of the invocation must satisfy the constraints $A' \subseteq A$ and $\mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{pc}'$. A legal invocation of the continuation can only come from one of the hosts satisfying the conditions $\vdash \mathsf{pc}'$ **ok** $@h$ and $A \subseteq \mathsf{TP}(h)$. This condition is checked dynamically to ensure that an untrusted host does not simulate a call to the continuation and thus fabricate authority. Legal reads and writes to a store location whose contents are labeled with ℓ are also checked dynamically to come only from a host h such that $\vdash l$ **ok** $@h$.

Enforcing the properties of linear continuations at run time requires one-time capabilities. the form (**lcont**$@h \ldots$) is evaluated, the host h constructs a closure object and hands the remote reference back to the originating host. The call-site must present the capability to invoke the continuation, after which both the closure and the capability are destroyed.

3 Semantics

This section discusses the novel features of the SPL@ type system and operational semantics. It also sketches some of the formal properties of the language.

3.1 Type system

The type system for SPL@ is similar to those found in the literature [VSI96, ML97, HR98, SV98, Mye99]. This section examines some of the more interesting typing rules; the complete set can be found in Appendix A.

Two separate contexts are maintained for type-checking purposes. We treat Γ as a finite partial map from nonlinear variables (x and f) to security types, whereas K is an *ordered* list of pairs of linear variables (y) and their corresponding types. The two contexts are separated by $\|$ in the typing judgments. The rules maintain the ordering on K: if $K = \bullet, (y_n : \kappa_n), \ldots, (y_1 : \kappa_1)$ then the continuation y_1 will be invoked before any of the $y_2 \ldots y_n$ are (if at all). K can be strengthened by dropping unit variables[1]. Also, to

[1]Note that this is sound because it is impossible for linear unit values to reach an active position during evaluation.

avoid problems with associativity of the context concatenation operator, we treat linear contexts as being equivalent up to the rules described in Appendix A.

The first kind of typing judgment is of the form $\Gamma \vdash v : \sigma$. It determines when the value v is well-typed, and it captures invariants that are independent of the host on which the value is residing. These rules are, for the most part, standard. Ordinary values cannot contain free linear variables because discarding the value would break the linearity constraint.

The rules for linear values are of the form $\Gamma \parallel K \ \vdash lv : \kappa$. A linear variable may be mentioned only if no other linear variables are present in the context.

A memory location $L^{\sigma}_{\ell}@h$ is a remote pointer to a value of type σ stored on host h. The pointer itself has been annotated with the security label ℓ. In order for such a location to be well formed, it must be the case that the data pointed to by the reference is allowed to reside on h. The security label of σ must have a confidentiality requirement no greater than $\mathsf{TP}(h)$, the maximum confidentiality guaranteed by the machine h. Likewise, the integrity of data pointed to by the reference can be no better than $\mathsf{TP}(h)$, because only those principals in $\mathsf{TP}(h)$ trust the data generated by h. These constraints are captured in the following rules:

$$\frac{\vdash \mathsf{label}(\sigma)\ \mathbf{ok}\ @h}{\Gamma \vdash L^{\sigma}_{\ell}@h : \sigma\ \mathsf{ref}_{\ell}} \qquad\qquad \frac{\langle \emptyset, \mathsf{TP}(h)\rangle \sqsubseteq \ell \sqsubseteq \langle \mathsf{TP}(h), \emptyset\rangle}{\vdash \ell\ \mathbf{ok}\ @h}$$

Here is the rule for linear continuations:

$$\frac{\begin{array}{c}\mathsf{pc} \sqsubseteq \langle \mathsf{TP}(h), \emptyset\rangle \\ \mathsf{label}(\sigma) \sqsubseteq \langle \mathsf{TP}(h), \emptyset\rangle \\ \Gamma, x\!:\!\sigma \parallel y\!:\!\kappa, K\ [A, \mathsf{pc}]\ @h\ \vdash e\end{array}}{\Gamma \parallel K\ \vdash \mathbf{lcont}@h[A, \mathsf{pc}](x\!:\!\sigma, y\!:\!\kappa) = e : (\sigma, \kappa)\ \mathsf{lcont}}$$

The first two conditions require that the labels on the program counter and argument x be public enough that h is trusted to manipulate them. The linear argument, y, can be thought of as the tail of the stack of continuations that have yet to be invoked. Intuitively, this judgment says that the continuation promises to invoke the continuations in K before jumping to the continuation represented by y, and it is under these assumptions that the body e is checked.

During computation, the substitution operation will cause values to propagate to various hosts. To prevent secret data from being sent to untrusted hosts, we also require a rule that determines when it is permissible for a value to be substituted into a located expression. The following rule expresses this privacy constraint:

$$\frac{\Gamma \vdash v : \tau_{\ell} \quad \ell \sqsubseteq \langle \mathsf{TP}(h), \emptyset\rangle}{\Gamma\ @h\ \vdash v : \tau_{\ell}}$$

The rules for both linear and nonlinear continuations need to determine when the body of the continuation is well-typed. The judgments for type-checking code expressions are of the form $\Gamma \parallel K\ [A, \mathsf{pc}]\ @h\ \vdash e$. The remainder of the judgment can be read as "in a context where the program has at least the authority to act for the principals in A, and the control flow depends on values of at most security label pc, the code e can be executed on h."

As an example, consider the following rule for type-checking conditional branches:

$$\frac{\Gamma\ @h\ \vdash v : \mathsf{int}_{\ell} \quad \Gamma \parallel K\ [A, \mathsf{pc} \sqcup \ell]\ @h\ \vdash e_i \quad \langle \emptyset, \mathsf{TP}(h)\rangle \sqsubseteq \mathsf{pc}}{\Gamma \parallel K\ [A, \mathsf{pc}]\ @h\ \vdash \mathbf{if0}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2}$$

This security annotation ℓ of the integer regulating the branch is propagated to the label of the program counter when type-checking the branches, reflecting the fact that information about v can be learned by observing the program counter in either e_1 or e_2.

The rule for assignment, shown below, checks that the label of the program counter is no more restrictive than the security label, $\mathsf{label}(\sigma)$, of the values that can be stored in the reference. To prevent information from being leaked via aliasing, the label ℓ of the reference value itself may be no more restrictive than the security label of its contained values [DD77, ML97, HR98].

$$\frac{\Gamma\ @h\ \vdash v : \sigma\ \mathsf{ref}_\ell \quad \Gamma\ @h\ \vdash v' : \sigma \quad \mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{label}(\sigma) \quad \Gamma \parallel K\ [A, \mathsf{pc}]\ @h\ \vdash e}{\Gamma \parallel K\ [A, \mathsf{pc}]\ @h\ \vdash \textbf{set}\ v := v'\ \textbf{in}\ e}$$

Together, these two rules ensure that information learned by branching on an integer cannot be stored into a low-security memory location.

Linear continuations are introduced with the following typing rule:

$$\frac{\begin{array}{c}\Gamma \parallel K_2\ \vdash \textbf{lcont}@h'[A', \mathsf{pc}'](x' \!:\! \sigma', y' \!:\! \kappa') = e' : \kappa'' \\ \kappa'' = (\sigma, \kappa)\ \mathsf{lcont} \qquad A' \subseteq A \qquad \mathsf{pc} \sqsubseteq \mathsf{pc}' \\ \Gamma \parallel K_1, y \!:\! \kappa''\ [A, \mathsf{pc}]\ @h\ \vdash e\end{array}}{\Gamma \parallel K_1, K_2\ [A, \mathsf{pc}]\ @h\ \vdash \textbf{letlin}\ y = \textbf{lcont}@h'[A', \mathsf{pc}'](x' \!:\! \sigma', y' \!:\! \kappa') = e'\ \textbf{in}\ e}$$

The rule above says that the actual authority declared by the linear continuation (A') is less than that available (A) when the continuation is introduced, whereas the declared program counter pc' is more restrictive. These measures ensure that the linear continuation "captures" the security context (or one more restrictive) at the point of declaration.

Maintaining the ordering property on requires some work. The linear context is split into two parts, K_1 and K_2. The body of e is checked under the assumption that continuation y is invoked before any of the continuations in K_1. Because the continuation bound to y invokes the continuations in K_2 before its argument (as seen above in the lcont checking rule), the ordering relation K_1, K_2 in subsequent computation will be respected.

Compare the rules for **lgoto** and **goto**:

$$\frac{\begin{array}{c}\Gamma \parallel K_2\ \vdash lv_1 : (\sigma, \kappa)\ \mathsf{lcont} \\ \Gamma\ @h\ \vdash v : \sigma \\ \Gamma \parallel K_1\ \vdash lv_2 : \kappa \\ \langle \emptyset, \mathsf{TP}(h) \rangle \sqsubseteq \mathsf{pc} \sqsubseteq \mathsf{label}(\sigma) \quad A \subseteq \mathsf{TP}(h)\end{array}}{\Gamma \parallel K_1, K_2\ [A, \mathsf{pc}]\ @h\ \vdash \textbf{lgoto}\ lv_1\ v\ lv_2} \qquad \frac{\begin{array}{c}\Gamma\ @h\ \vdash v : [A', \mathsf{pc}'](\sigma, \kappa)\ \mathsf{cont}_\ell \\ \Gamma\ @h\ \vdash v' : \sigma \quad \mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{label}(\sigma) \\ \Gamma \parallel K\ \vdash lv : \kappa \\ \langle \emptyset, \mathsf{TP}(h) \rangle \sqsubseteq \mathsf{pc} \quad \mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{pc}' \quad A' \subseteq A \subseteq \mathsf{TP}(h)\end{array}}{\Gamma \parallel K\ [A, \mathsf{pc}]\ @h\ \vdash \textbf{goto}\ v\ v'\ lv}$$

They both require that the arguments to the continuation be of the right type, and that the label of the argument to the continuation be more restrictive than the current program counter. The key differences are the constraints on the security context of the ordinary continuation. The target of a **goto** *inherits* the calling security context, while the target of a **lgoto** *restores* the security context of the point at which it was declared.

The proper ordering of the linear context is maintained in the **lgoto** rule because lv_1 is being called and promises to invoke its argument *after* evaluating the continuations in K_2. The argument, lv_2 will, in turn, invoke the continuations in K_1, hence keeping the order K_1, K_2.

The key difference between this rule and the one for normal continuations is that the security label of the calling context does not influence the security label of the body of the linear continuation. This means that when a linear continuation is invoked, the security context is reset to be that of the point where the continuation was declared.

The last typing judgment form in the language determines when a primitive operation is suitable for execution on a particular host in a given security context. The most interesting example is the rule for declassifying values:

$$\frac{\Gamma\ @h\ \vdash v : \tau_{\ell'} \quad B \subseteq A \quad \ell' \sqcap \langle \mathsf{P}, B \rangle \sqsubseteq \ell \sqcup \langle B, \mathsf{P} \rangle \quad \mathsf{pc} \sqsubseteq \langle \mathsf{P}, B \rangle \quad \mathsf{pc} \sqsubseteq \ell}{\Gamma\ [A, \mathsf{pc}]\ @h\ \vdash \textbf{declassify}(v, \ell) : \tau_\ell}$$

This construct allows a set of principals to give up ownership of a piece of data (decrease privacy) or endorse a piece of untrusted data (increase integrity). The authority of the principals necessary to perform the declassification (B) must be included in the authority of the current security context. The relation between ℓ', the original label of the value, and ℓ, the new label can best be seen as a combination of two simpler inequalities:

$$\ell' \sqcap \langle \mathsf{P}, B\rangle \sqsubseteq \ell \qquad\qquad \ell' \sqsubseteq \ell \sqcup \langle B, \mathsf{P}\rangle$$

The left inequality says that only principals in B can add their endorsement to the integrity of the data, so that it is now trusted by those principals who originally trusted it plus the members of B. The right inequality says that only the principals in B may remove their privacy constraints from the data, all the other privacy interests of the original label must still be observed.

The condition $\mathsf{pc} \sqsubseteq \ell$ implies that it is never possible to declassify a value to below the security label on the current program counter. Effectively, each value's label is implicitly joined with the label of the program counter—the only way to remove the effects of that stamping is to use the **setpc** expression to declassify the program counter.

The condition $\mathsf{pc} \sqsubseteq \langle B, \mathsf{P}\rangle$ says that principals B trust that the program counter has been computed correctly, as discussed in Section 2.7.

3.2 Operational semantics

The operational semantics for this language are given in full in Appendix B. We mention a few salient points here. The model is given in terms of a transition relation between machine configurations of the form

$$\langle M[A, \mathsf{pc}]@h \triangleright e\rangle$$

This configuration denotes a program expression e being run on host h with run-time security context $[A, \mathsf{pc}]$ and memory, M.

Memories are finite partial maps from locations to values. A memory M is considered to be well-formed (written M **wf**) if for each location L^σ in M's domain $\bullet \vdash M(L^\sigma) : \sigma$ and, furthermore, that M is closed under dereference. That is, if $L^\sigma \in Dom(M)$ and $M(L^\sigma) = L'^{\sigma'}$ then $L'^{\sigma'} \in Dom(M)$. The notation $M[L^\sigma \mapsto v]$ is used to denote both update and extension of the partial map. We use the notation $Loc(e)$ to denote the set of locations appearing in a program e.

The notations $e\{v/x\}$ and $e\{lv/y\}$ denote capture-avoiding substitution of ordinary and linear values.

Evaluation of primitive operations causes no side effects and yields a value. The notation $[\![\oplus]\!]$ is the semantic operation corresponding to $\oplus$; thus $[\![+]\!]$ is the integer addition operation.

The operational semantics takes the view that values are stamped with their security labels and these labels are checked at runtime to ensure that no explicit information leaks occur.

4 Soundness and Security

The purpose of this section is to rigorously prove the soundness of SPL@ and show that the fragment not containing **declassify** and **setpc** exhibits non-interference.

The proofs in this paper assume that all typing derivations for values and linear values are in *canonical form*—the derivations alternate between subsumption and non-subsumption rules, ending with a subsumption rule. (It is always possible to put proofs in this canonical form because of the reflexivity and transitivity of the $\leq$ relation.) Similarly, we always assume that the linear contexts, K, are equivalent up to the $\equiv$ relation, and so omit specific equivalence rules unless needed for clarity. We also adopt the convention that $\mathsf{label}(\tau_\ell) = \ell$.

Ordinary contexts, Γ, are subject to the usual Strengthening, Weakening, and Exchange operations; however, linear contexts, K, are not.

As mentioned earlier, we must rule out certain well-typed programs that violate security by starting out with linear continuations that don't correspond to merge points. For example, we need to rule out programs such as

$$\begin{array}{l}\textbf{if0}\ h\ \textbf{then}\\ \quad \textbf{lgoto}\ (\textbf{lcont}@h[A,\bot](x\!:\!1, y\!:\!1) = \textbf{set}\ l := 0\ \textbf{in}\ \ldots)\ \langle\rangle\ \langle\rangle\\ \textbf{else}\\ \quad \textbf{lgoto}\ (\textbf{lcont}@h[a,\bot](x\!:\!1, y\!:\!1) = \textbf{set}\ l := 1\ \textbf{in}\ \ldots)\ \langle\rangle\ \langle\rangle\end{array}$$

We thus make the provision that in a well-formed initial program, all linear continuations are introduced via **letlin**.

Definition 4.1 (Initial Program) *An* ***initial program*** *is an expression* e *such that* $Loc(e) = \emptyset$*, and, for each* **lgoto** $lv_1\ v\ lv_2$ *subexpression of* e*,* lv_1 *is a variable and* lv_2 *is either a variable or* $\langle\rangle$*.*

4.1 Soundness

This section proves the soundness theorem for SPL@. The proof is, for the most part, standard, following in the style of Wright and Felleisen [WF92, WF94]. We omit discussion of the Subject Reduction and Progress lemmas.

A simple proposition that we shall not prove is the following:

Proposition 4.1 (Base Value Relabeling) *If* $\Gamma \vdash bv_\ell : \tau_\ell$ *then* $\Gamma \vdash bv_{\ell'} : \tau_{\ell'}$.

We shall use the Base Value Relabeling proposition without mentioning it explicitly in the proofs below.

Lemma 4.1 (Substitution I) *Assume* $\Gamma\ @h\ \vdash v : \sigma$ *then*

(i) *If* $\Gamma, x\!:\!\sigma\ @h'\ \vdash v' : \sigma'$ *then* $\Gamma\ @h'\ \vdash v'\{v/x\} : \sigma'$.

(ii) *If* $\Gamma, x\!:\!\sigma \parallel K\ \vdash lv : \kappa$ *then* $\Gamma \parallel K\ \vdash lv\{v/x\} : \kappa$.

(iii) *If* $\Gamma, x\!:\!\sigma\ [A, \mathsf{pc}]\ @h'\ \vdash prim : \sigma'$ *then* $\Gamma\ [A, \mathsf{pc}]\ @h'\ \vdash prim\{v/x\} : \sigma'$.

(iv) *If* $\Gamma, x\!:\!\sigma \parallel K\ [A, \mathsf{pc}]\ @h'\ \vdash e$ *then* $\Gamma \parallel K\ [A, \mathsf{pc}]\ @h'\ \vdash e\{v/x\}$.

Proof: By mutual induction on the (canonical) derivations of (i)–(iv).

(i) By assumption, there exists a derivation of the form

$$\cfrac{\cfrac{\Gamma, x\!:\!\sigma \vdash v' : \sigma'' \quad \vdash \sigma'' \le \sigma'}{\Gamma, x\!:\!\sigma \vdash v' : \sigma'} \qquad \mathsf{label}(\sigma') \sqsubseteq \langle \mathsf{TP}(h'), \emptyset\rangle}{\Gamma, x\!:\!\sigma\ @h'\ \vdash v' : \sigma'}$$

We proceed by cases on the rule used to conclude $\Gamma, x : \sigma\ \vdash\ v' : \sigma''$. In cases $[TV1]$, $[TV2]$, and $[TV3]$ we have $v'\{v/x\} = v'$, and the result follows by Strengthening and derivation above. In the case of $[TV4]$, we have either $x\{v/x\} = v$, which, by assumption, has type $\sigma = \sigma''$ or $x'\{v/x\} = x'$, also of type σ''. In either case, this information plus the derivation above yields the desired result. The case of $[TV5]$ follows from inductive hypothesis (iv).

(ii) This cases follows analogously to the case for (i); the rule $[TL3]$ makes use of inductive hypothesis (iv).

(iii) This follows directly from the inductive hypothesis (i).

(iv) Follows by inductive hypotheses (i)–(iv).

□

Note that neither ordinary values nor primitive operations may contain free linear variables. This means that substitution of a linear value in them has no effect. The following lemma strengthens substitution to *open* linear values and also shows that the ordering on the linear context is maintained.

Lemma 4.2 (Substitution II) *Assume* $\bullet \parallel K \vdash lv : \kappa$ *and that all bound linear variables appearing in* lv' *and* e *are disjoint from* K. *Furthermore, assume that* K *contains only linear continuation types. Then*

(i) *If* $\Gamma \parallel K_1, y{:}\kappa, K_2 \vdash lv' : \kappa'$ *then* $\Gamma \parallel K_1, K, K_2 \vdash lv'\{lv/y\} : \kappa'$.

(ii) *If* $\Gamma \parallel K_1, y{:}\kappa, K_2\ [A, \mathsf{pc}]\ @h' \vdash e$ *then* $\Gamma \parallel K_1, K, K_2\ [A, \mathsf{pc}]\ @h' \vdash e\{lv/y\}$.

Proof: By mutual induction on the (canonical) typing derivations of (i) and (ii).

(i) The canonical typing derivation for lv' is:

$$\frac{\Gamma \parallel K_1, y{:}\kappa, K_2 \vdash lv' : \kappa'' \quad \vdash \kappa'' \leq \kappa'}{\Gamma \parallel K_1, y{:}\kappa, K_2 \vdash lv' : \kappa'}$$

We proceed by cases on the rule used to conclude $\Gamma \parallel K_1, y{:}\kappa, K_2 \vdash lv' : \kappa''$.

$[TL1]$ Then $\kappa'' = 1$ and $K_1, y{:}\kappa, K_2 \equiv \bullet$. It follows that $\kappa = 1$ and all the types mentioned in K_1, K_2, and K must be 1 as well. Thus, $K = \bullet$. By the equivalence rules for linear contexts, we have $K_1, K, K_2 \equiv K_1, K_2 \equiv \bullet$. It thus follows that $\langle\rangle\{lv/y\} = \langle\rangle$, and $\Gamma \parallel K_1, K_2 \vdash \langle\rangle : \kappa''$.

$[TL2]$ If $lv' = y$ and $\kappa'' = 1$ then $\kappa = 1$. It follows that $K \equiv \bullet \equiv K_1, y : 1, K_2 \equiv K_1, K, K_2$. By definition of substitution, $lv'\{lv/y\} = lv$ and so we have $\Gamma \parallel K_1, K_2 \vdash lv : \kappa''$. This, plus the fact that $\kappa'' \leq \kappa'$ concludes this case. If $lv' = y$ and $\kappa'' = \kappa \neq 1$, it follows that $K_1, y : \kappa, K_2 \equiv y : \kappa$, and hence $K_1 \equiv K_2 \equiv \bullet$. This implies that $K_1, K, K_2 \equiv K$ and thus by the assumption that $lv = y\{lv/y\}$ is well-typed under K, we have $\Gamma \parallel K_1, K, K_2 \vdash lv : \kappa$, and the result follows from the subtyping of $\kappa \leq \kappa'$. If $lv' = y' \neq y$ then $\sigma = 1$ and it follows that $K \equiv \bullet$. Thus $K_1, K, K_2 \equiv K_1, K_2$. Furthermore, $lv'\{lv/y\} = lv' = y'$ and we have $\Gamma \parallel K_1, K, K_2 \vdash lv' : \kappa''$. The result follows via subtyping.

$[TL3]$ This case follows immediately from inductive hypothesis (ii) using the assumption that the bound linear variable in the linear continuation is disjoint from K.

(ii) This part of the lemma follows almost immediately from the inductive hypotheses. The interesting cases are $[TE5]$ and $[TE7]$, which must ensure that the ordering on the linear context is maintained. We show the case for $[TE5]$, as $[TE7]$ uses the same technique.

$[TE5]$ By assumption, there is a derivation of the following form:

$$\frac{\begin{array}{c}\Gamma \parallel K_b \vdash \textbf{lcont}@h''[A', \mathsf{pc}'](x'{:}\sigma', y'{:}\kappa') = e' : \kappa'' \\ \kappa'' = (\sigma, \kappa)\ \mathsf{lcont} \qquad A' \subseteq A \qquad \mathsf{pc} \sqsubseteq \mathsf{pc}' \\ \Gamma \parallel K_a, y{:}\kappa''\ [A, \mathsf{pc}]\ @h' \vdash e\end{array}}{\Gamma \parallel K_1, y{:}\kappa, K_2\ [A, \mathsf{pc}]\ @h' \vdash \textbf{letlin}\ y'' = \textbf{lcont}@h''[A', \mathsf{pc}'](x'{:}\sigma', y'{:}\kappa') = e'\ \textbf{in}\ e}$$

Where $K_1, y : \kappa, K_2 \equiv K_a, K_b$. If y appears in K_a then $K_a \equiv K_1, y : \kappa, K_2^-$ and $K_b \equiv K_2^+$ where $K_2^-, K_2^+ \equiv K_2$. In this case, y can't appear in K_b and it follows that $e'\{lv/y\} = e'$. Inductive hypothesis (ii) applied to $\Gamma \parallel K_1, y : \kappa, K_2^-\ [A, \mathsf{pc}]\ @h' \vdash e$ yields the judgement

$\Gamma \parallel K_1, K, K_2^- \; [A, \mathsf{pc}] \; @h' \; \vdash e\{lv/y\}$. Thus, an application of rule $[TE5]$ yields the desired result, as $K_1, K, K_2^-, K_2^+ \equiv K_1, K, K_2$.

The case in which y appears in K_b is similar to the one above, except that K_1 is split into K_1^- and K_1^+. If y appears in neither K_a or K_b then $\kappa = 1$ and it follows that $K \equiv \bullet$, which also yields the desired result.

□

Lemma 4.3 (Canonical Forms I) *If* $\bullet \vdash v : \sigma$ *then*

(i) *If* $\sigma = \mathsf{int}_\ell$ *then* $v = n_{\ell'}$ *for some integer* n *and* $\ell' \sqsubseteq \ell$.

(ii) *If* $\sigma = 1_\ell$ *then* $v = \langle\rangle_{\ell'}$ *for some* $\ell' \sqsubseteq \ell$.

(iii) *If* $\sigma = \sigma' \; \mathsf{ref}_\ell$ *then* $v = L_{\ell'}^{\sigma'} @h$ *for some* h *and* $\ell' \sqsubseteq \ell$.

(iv) *If* $\sigma = [A, \mathsf{pc}](\sigma', \kappa) \; \mathsf{cont}_\ell$ *then* $v = (\mathbf{cont}@h \; f_B \; [A', \mathsf{pc}'](x\!:\!\sigma'', y\!:\!\kappa') = e)_{\ell'}$ *where* $\ell' \sqsubseteq \ell$, $A' \subseteq A$, $\mathsf{pc} \sqsubseteq \mathsf{pc}'$, $\vdash \sigma' \leq \sigma''$, *and* $\vdash \kappa \leq \kappa'$.

Proof (sketch): By inspection of the typing rules and the form of values. □

Lemma 4.4 (Canonical Forms II) *If* $\bullet \parallel \bullet \; \vdash lv : \kappa$ *then*

(i) *If* $\kappa = 1$ *then* $lv = \langle\rangle$ *or* $lv = y$.

(ii) *If* $\kappa = (\sigma, \kappa') \; \mathsf{lcont}$ *then* $lv = \mathbf{lcont}@h[A, \mathsf{pc}](x\!:\!\sigma', y\!:\!\kappa'') = e$ *where* $\vdash \sigma \leq \sigma'$ *and* $\vdash \kappa' \leq \kappa''$.

Proof (sketch): By inspection of the typing rules and the form of linear values. □

Definition 4.2 (Memory Well-formedness) *A memory, M, is a partial finite map from the set of locations to closed values. M is said to be well formed, written M* **wf**, *if the following conditions hold:*

(i) *For each L^σ in the domain of M, $\bullet \vdash M(L^\sigma) : \sigma$.*

(ii) *M is closed under dereference, that is if $M(L^\sigma) = L'^{\sigma'}$ then $M(L'^{\sigma'})$ is defined.*

Definition 4.3 (Locations) *For any well-typed primitive operation, $prim$ (respectively program, e), let $Loc(prim)$ (respectively $Loc(e)$), be the set of all locations $L^\sigma @h$ appearing in $prim$ (respectively e).*

Lemma 4.5 (Primitive Evaluation) *If* $\bullet \; [A, \mathsf{pc}] \; @h \; \vdash prim : \sigma$ *and* M **wf** *and* $Loc(prim) \subseteq Dom(M)$ *and* $M[A, \mathsf{pc}]@h \vdash prim \Downarrow v$ *then* $\bullet \; @h \; \vdash v : \sigma$.

Proof: By cases on the evaluation rule used.

$[P1]$ By assumption, we have

$$\dfrac{\dfrac{\dfrac{\bullet \vdash bv_\ell : \tau_\ell \quad \vdash \tau_\ell \leq \tau'_{\ell'}}{\bullet \vdash bv_\ell : \tau'_{\ell'}} \qquad \ell' \sqsubseteq \langle \mathsf{TP}(h), \emptyset \rangle}{\bullet \; @h \; \vdash bv_\ell : \tau'_{\ell'}} \qquad \mathsf{pc} \sqsubseteq \ell'}{\bullet \; [A, \mathsf{pc}] \; @h \; \vdash bv_\ell : \tau'_{\ell'}}$$

We need to show $\bullet \; @h \; \vdash bv_{\ell \sqcup \mathsf{pc}} :$. It follows from the derivation above that $\ell \sqsubseteq \ell'$, and as $\mathsf{pc} \sqsubseteq \ell'$ it is the case that $\ell \sqcup \mathsf{pc} \sqsubseteq \ell'$. Thus we have the inequality $\tau_{\ell \sqcup \mathsf{pc}} \leq \tau'_{\ell'}$, and so the result follows by subsumption and an application of the rule $[TV7]$.

$[P2]$ This case is similar to the previous one.

$[P3]$ In this case, we assume a derivation of the following form:

$$\dfrac{\dfrac{\bullet \vdash L_{\ell'}^{\tau_\ell}@h' : \tau_\ell\ \mathsf{ref}_{\ell'} \quad \vdash \tau_\ell\ \mathsf{ref}_{\ell'} \sqsubseteq \tau_\ell\ \mathsf{ref}_{\ell''}}{\bullet\ @h\ \vdash L_{\ell'}^{\tau_\ell}@h' : \tau_\ell\ \mathsf{ref}_{\ell''}} \qquad \mathsf{pc} \sqsubseteq \ell \sqcup \ell'' \sqsubseteq \langle \mathsf{TP}(h), \emptyset \rangle}{\bullet\ [A, \mathsf{pc}]\ @h\ \vdash \textbf{deref}(L_{\ell'}^{\tau_\ell}@h') : \tau_{\ell \sqcup \ell''}}$$

By the well-formedness of M, $M(L^{\tau_\ell}) = bv_{\ell'''}$ and we also have $\bullet \vdash bv_{\ell'''} : \tau_\ell$. This implies that $\ell''' \sqsubseteq \ell$. We must show that $\bullet\ @h\ \vdash bv_{\ell' \sqcup \ell''' \sqcup \mathsf{pc}} : \tau_{\ell \sqcup \ell''}$ but this follows from subsumption and rule $[TV7]$ because $\ell' \sqcup \ell''' \sqcup \mathsf{pc} \sqsubseteq \ell \sqcup \ell''$.

$[P4]$ This case follows, as in the first two cases, from the assumption that $\mathsf{pc} \sqsubseteq \ell$ in the preconditions of rule $[TP4]$ plus subsumption.

□

Lemma 4.6 (Program Counter Label) *If* $\Gamma \parallel K\ [A, \mathsf{pc}]\ @h\ \vdash e$ *then* $\langle \emptyset, \mathsf{TP}(h) \rangle \sqsubseteq \mathsf{pc} \sqsubseteq \langle \mathsf{TP}(h), \emptyset \rangle$

Proof (sketch): By induction on the derivation that e is well-typed. The left inequality follows from the presence of $\langle \emptyset, \mathsf{TP}(h) \rangle \sqsubseteq \mathsf{pc}$ on rules for expressions which end basic blocks. The right inequality follows from the presence of inequalities of the form $\mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{label}(\sigma)$, where σ is the type of a primitive operation or value typechecked at h. Because the rule for typechecking values (or primitive expressions) at h includes the condition $\mathsf{label}(\sigma) \sqsubseteq \langle \mathsf{TP}(h), \emptyset \rangle$ it follows that $\mathsf{pc} \sqsubseteq \langle \mathsf{TP}(h), \emptyset \rangle$. □

Lemma 4.7 (Program Counter Variance) *If* $\Gamma \parallel K\ [A, \mathsf{pc} \sqcup \ell]\ @h\ \vdash e$ *and* $\langle \emptyset, \mathsf{TP}(h) \rangle \sqsubseteq \mathsf{pc}$ *and* $\ell' \sqsubseteq \ell$ *then* $\Gamma \parallel K\ [A, \mathsf{pc} \sqcup \ell']\ @h\ \vdash e$.

Proof: The proof is by induction on the derivation that e is well-typed. Note that because $\mathsf{pc} \sqcup \boldsymbol{\ell} \sqsubseteq \mathsf{pc} \sqcup \ell$ all inequalities involving $\mathsf{pc} \sqcup \ell$ on the left of $\sqsubseteq$ in the typing rules will still be valid with $\mathsf{pc} \sqcup \boldsymbol{\ell}$. To see that inequalities in which $\mathsf{pc} \sqcup \ell$ appear on the right of $\sqsubseteq$ still hold, note that these occur only when the left-hand side is $\langle \emptyset, \mathsf{TP}(h) \rangle$. Because $\langle \emptyset, \mathsf{TP}(h) \rangle \sqsubseteq \mathsf{pc} \sqsubseteq \mathsf{pc} \sqcup \ell'$ these inequalities are still valid as well. □

Lemma 4.8 (Authority) *If* $\Gamma \parallel K\ [A, \mathsf{pc}]\ @h\ \vdash e$ *then* $A \subseteq \mathsf{TP}(h)$.

Proof: By induction on the typing derivation for e. The base cases, **halt**, **goto**, and **lgoto** all explicitly mention this condition. The remainder of the cases follow immediately by the inductive hypothesis. □

Lemma 4.9 (Subject Reduction) *If* $\bullet \parallel K\ [A, \mathsf{pc}]\ @h\ \vdash e$, *and* M **wf**, *and* $Loc(e) \subseteq Dom(M)$ *and*

$$\langle M[A, \mathsf{pc}]@h \rhd e \rangle \longmapsto \langle M'[A', \mathsf{pc}']@h' \rhd e' \rangle$$

Then $\bullet \parallel K\ [A', \mathsf{pc}']\ @h'\ \vdash e'$ *and* M' **wf**, *and* $Loc(e') \subseteq Dom(M')$.

Proof: By cases on the transition step taken:

$[O1]$ Because **let** $x = prim$ **in** e is well-typed, $prim$ is too. Thus by the Primitive Evaluation Lemma, $prim$ evaluates to a value v of the same type. Substitution I, part (iv) tells us that $e\{v/x\}$ is well-typed. Because M doesn't change it is still well-formed, and to see that $Loc(e\{v/x\}) \subseteq Dom(M)$ consider that the only way $Loc(e\{v/x\})$ could be larger than $Loc($**let** $x = prim$ **in** $e)$ is if $prim$ is a dereference operation and the memory location contains another location not in e. This case is covered by the requirement that M be closed under dereference.

$[O2]$ By assumption, that $\bullet \parallel K\ [A, \mathsf{pc}]\ @h\ \vdash$ **let** $x = (\textbf{ref}(\sigma)\ bv_\ell)_{\ell'}$ **in** e. Working backwards through the canonical derivation yields the following antecedents: $\bullet \vdash bv_\ell : \tau_\ell$, and $\vdash \tau_\ell \leq \sigma$, and $\mathsf{label}(\sigma) \sqsubseteq \langle \mathsf{TP}(h), \emptyset\rangle$, and $\mathsf{pc} \sqsubseteq \mathsf{label}(\sigma)$, and $\mathsf{pc} \sqsubseteq \ell' \sqsubseteq \langle \mathsf{TP}(h), \emptyset\rangle$, and $x{:}\sigma\ \mathsf{ref}_{\ell'} \parallel K\ [A, \mathsf{pc}]\ @h\ \vdash e$. From these we conclude that $\mathsf{pc} \sqcup \ell' \sqsubseteq \ell' \sqsubseteq \langle \mathsf{TP}(h), \emptyset\rangle$ and via $[TV3]$ and $[TV7]$ it follows that $\bullet\ @h\ \vdash L^\sigma_{\ell' \sqcup \mathsf{pc}} @h : \sigma\ \mathsf{ref}_{\ell'}$. This fact, plus the well-typedness of e lets us apply Substitution Lemma I, (iv) to conclude $\bullet \parallel K\ [A, \mathsf{pc}]\ @h\ \vdash e\{L^\sigma_{\ell' \sqcup \mathsf{pc}} @h/x\}$. Now, to see that the conditions on M are maintained, note that if bv is a location, then it is contained in the set of locations of the entire let expression and thus, by assumption must occur in the domain of M. This implies that $M[L^\sigma \mapsto bv_{\ell \sqcup \mathsf{pc}}]$ is still closed under dereference. Finally, we must check that $\bullet \vdash bv_{\ell \sqcup \mathsf{pc}} : \sigma$, but this follows from subsumption and the facts that $\tau_\ell \leq \sigma$ and $\mathsf{pc} \sqsubseteq \mathsf{label}(\sigma)$.

$[O3]$ This case follows similarly to the previous case.

$[O4]$ Assume that $\bullet \parallel K\ [A, \mathsf{pc}]\ @h\ \vdash$ **if0** 0_ℓ **then** e_1 **else** e_2. It follows that $\bullet \vdash 0_\ell : \mathsf{int}_{\ell'}$ and $\ell \sqsubseteq \ell'$ and that $\ell' \sqsubseteq \langle \mathsf{TP}(h), \emptyset\rangle$ and $\bullet \parallel K\ [A, \mathsf{pc} \sqcup \ell']\ @h\ \vdash e_1$. Furthermore, $\mathsf{pc} \sqsubseteq \langle \mathsf{TP}(h), \emptyset\rangle$ and so it follows by the Program Counter Variance Lemma that $\bullet \parallel K\ [A, \mathsf{pc} \sqcup \ell]\ @h\ \vdash e_1$. Because M doesn't change and it initially satisfied the well-formedness conditions and the locations of e_1 are a subset of the locations of the entire conditional, M is still valid after the transition.

$[O5]$ This case is nearly identical to the previous one.

$[O6]$ This case follows from Substitution II, (ii), and the fact that the conditions on M are satisfied after the substitution. Note that the order of K is preserved by the step.

$[O7]$ This case follows from the well-typedness of the body of the continuation being jumped to, plus two applications of Substitution I, (iv) and one application of Substitution II, (ii). The fact that $bv_{\ell' \sqcup \ell \sqcup \mathsf{pc}}$ has type σ follows from subsumption and the fact that $\mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{label}(\sigma)$.

$[O8]$ This case is similar to the previous case.

$[O9]$ This case follows immediately from the fact that the **setpc** body is well-typed under pc'. The memory invariants carry through because the locations of the entire expression are just those in the body of the **setpc**.

□

Definition 4.4 *A configuration $\langle M[A, \mathsf{pc}]@h \triangleright e\rangle$ is* stuck *if e is not* **halt**$_\sigma$ *v for some value v or no transition rule applies.*

Lemma 4.10 (Progress) *If $\bullet \parallel \bullet\ [A, \mathsf{pc}]\ @h\ \vdash e$ and M* **wf** *and $Loc(e) \subseteq Dom(M)$, then either e is of the form* **halt**$_\sigma$ *v or there exist M', A', pc', and e' such that*

$$\langle M[A, \mathsf{pc}]@h \triangleright e\rangle \longmapsto \langle M'[A', \mathsf{pc}']@h' \triangleright e'\rangle$$

Proof (sketch): By the Canonical Forms lemmas and inspection of the rules. We must ensure that conditions such as $\vdash \ell \sqcup \mathsf{pc}$ **ok** $@h$ on rule $[P1]$ are met by well-typed terms. These follow from the Program Counter Label lemma and the fact that ℓ is obtained by typing a value at h, which implies that $\ell \sqsubseteq \langle \mathsf{TP}(h), \emptyset\rangle$. The conditions on the authority are implied by the Authority lemma. □

Theorem 4.1 (Type Soundness) *Well-typed programs do not get stuck.*

Proof: By induction on the number of transition steps taken, using Subject Reduction and Progress. □

4.2 Noninterference

This section proves a noninterference result for the subset of SPL@ that excludes **declassify** and **setpc**. The proof is inspired by the preservation-style proof of noninterference used by Volpano and Smith [SV98], and shares some similarities with the logical-relations style proof given by Heintze and Riecke [HR98].

Informally, the noninterference result shows that low-security computations are not able to observe high-security data. Here, "low-security" refers to an arbitrary label ζ or labels $\sqsubseteq \zeta$, and "high-security" values are those whose labels are $\not\sqsubseteq \zeta$. At a high level, the proof works by showing that at each point in the program's evaluation, the high-security data can be "factored out" of the computation, and arbitrarily changed. The noninterference result then states that the original computation and the one with altered high-security data eventually compute the *same* low-security integer as output. Furthermore, the memory locations visible to low-security observers (those location storing data labeled $\sqsubseteq \zeta$) will also be indistinguishable at the end of the computation.

It is worth making a few comments about the strength of this result before describing it in more detail. As with previous noninterference results for call-by-value languages [HR98], the noninterference result holds only for programs which halt regardless of the high-security data. This means that information about high-security values can affect whether the program terminates. There are other possible channels that are not captured by this notion of noninterference: high-security values can alter the amount of time it takes for the program to compute an answer or alter the memory consumption of the program. We are able to prove noninterference despite these apparent security leaks because the language itself provides no means for observing these resources (for instance access to the system clock, or the ability to detect available memory).

Intuitively, there are two things that we must keep track of to determine whether a program exhibits the noninterference property. First, values with labels higher than ζ might potentially leak to a low-security observer. This is addressed in the type system by guaranteeing that the program counter label is at least as high as the label on any value that has influenced the control flow of the program, plus checks to make sure that, for instance, high security data is not written into low-security memory locations. It is easy to verify from the operational and static semantics that the program counter label is monotonically increasing *except* in the case that a linear continuation is invoked. Thus, the second thing which must be carefully regulated is how linear continuations are used—the ordering property is the crucial feature here.

To capture these two important features, we will use substitutions.

Definition 4.5 (Substitutions) *If Γ is any context, then we write $\gamma \models \Gamma$ to indicate that γ is a finite map from variables to closed values such that $Dom(\gamma) = Dom(\Gamma)$ and for every $x \in Dom(\Gamma)$ it is the case that $\bullet \vdash \gamma(x) : \Gamma(x)$.*

Similarly, for K a linear context, we write $\Gamma \vdash k \models K$ to indicate that k is a finite map of variables to linear values (possibly with free variables from Γ) with the same domain as K and such that for every $y \in Dom(K)$ we have $\Gamma \parallel \bullet \vdash k(y) : K(y)$.

A substitution γ will be applied to an expression e, written $\gamma(e)$ to indicate the capture-avoiding substitution of the value $\gamma(x)$ for free occurrences of x in the expression e for each variable $x \in Dom(\gamma)$. We use the similar notation $k(e)$ for application of linear substitutions to a term e.

One of the invariants maintained by well-typed programs is that at each evaluation step it is possible to factor out the relevant high-security values and those linear continuations that reset the program-counter label to be $\sqsubseteq \zeta$.

The idea that the substitutions contain the "relevant" information is contained in the following definitions. First, we must allow linear continuations which set the program counter label $\not\sqsubseteq \zeta$ to appear in the term, because, from the low-security point of view, they are not relevant. This observation motivate the following definition:

Definition 4.6 (lgoto Invariant) *A term satisfies the* **lgoto** *invariant if for every subexpression of the form* **lgoto** $lv_1\ v\ lv_2$*, it is the case that if* $lv_i = (\textbf{lcont}@h_i[A_i, \mathsf{pc}_i](\ldots) = \ldots)$ *then* $\mathsf{pc}_i \not\sqsubseteq \zeta$*.*

If k is a substitution containing only low-security continuations, and $k(e)$ is a closed-term such that e satisfies the **lgoto** Invariant, then all of the low-security continuations used in $k(e)$ must be obtained from k. In this way, we ensure that we have factored out all of the relevant continuations.

What does it mean for us to have factored out the appropriate high-security data? Assume that we wish to show that two e_1 and e_2 behave the same from the low-point of view. If the program counter is $\sqsubseteq \zeta$, meaning that e_1 and e_2 can perform actions visible to the low observer, a necessary condition for them to be equivalent is that they must perform the same computation on low-security values. Thus, if e_i are allowed to differ in their behavior on high-security data, we should at least be able to find substitutions γ_1 and γ_2 such that both substitutions contain only high-security data and $e_1 = \gamma_1(e)$ and $e_2 = \gamma_2(e)$. Note that e_1 and e_2 can *both* be written in terms of e, after factoring out the high-security data. On the other hand, if the program counter is $\not\sqsubseteq \zeta$, then no-matter what e_1 and e_2 do, their actions should not be visible from the low point of view. If we extend these intuitions to values and memories, we obtain the formal definitions below:

Definition 4.7 (Substitution Equivalence) $\boxed{\Gamma \vdash \gamma_1 \approx_\zeta \gamma_2}$ *and* $\boxed{\Gamma \parallel K \vdash k_1 \approx_\zeta k_2}$
We write $\Gamma \vdash \gamma_1 \approx_\zeta \gamma_2$ *if and only if* $\gamma_1, \gamma_2 \models \Gamma$ *and for every* $x \in Dom(\Gamma)$ *it is the case that* $\mathsf{label}(\gamma_1(x)) \not\sqsubseteq \zeta$ *and* $\mathsf{label}(\gamma_2(x)) \not\sqsubseteq \zeta$*. Furthermore, we require that* $\gamma_i(x)$ *satisfy the* **lgoto** *invariant.*

We write $\Gamma \parallel K \vdash k_1 \approx_\zeta k_2$ *if and only if* $\Gamma \vdash k_1, k_2 \models K$ *and for every* $y \in Dom(K)$ *it is the case that* $k_1(y) \equiv_\alpha k_2(y) = \textbf{lcont}@h[A, \mathsf{pc}](x : \sigma, y' : \kappa) = e$ *and* $\mathsf{pc} \sqsubseteq \zeta$*. It is also necessary that* e *satisfy the* **lgoto** *invariant.*

Value Equivalence $\boxed{v_1 \approx_\zeta v_2 : \sigma}$

We write $v_1 \approx_\zeta v_2 : \sigma$ if and only if there exist Γ, γ_1, and γ_2 plus terms $v_1' \equiv_\alpha v_2'$ such that $\Gamma \vdash \gamma_1 \approx_\zeta \gamma_2$ and $\Gamma \vdash v_1' : \sigma$ and $\Gamma \vdash v_2' : \sigma$ and $v_1 = \gamma_1(v_1')$ and $v_2 = \gamma_2(v_2')$. Furthermore, v_i' must each satisfy the **lgoto** invariant.

Memory Equivalence $\boxed{M_1 \approx_\zeta M_2}$
We write $M_1 \approx_\zeta M_2$ if an only if for all $L^\sigma \in Dom(M_1) \cup Dom(M_2)$ if $\mathsf{label}(\sigma) \sqsubseteq \zeta$ then $L^\sigma \in Dom(M_1) \cap Dom(M_2)$ and $M_1(L^\sigma) \approx_\zeta M_2(L^\sigma) : \sigma$. Furthermore, M_1 **wf** and M_2 **wf**.

Note that this equivalence on memories does not say anything about the locations containing high-security data.

Definition 4.8 (Noninterference Invariant) *The* ***noninterference invariant*** *is a predicate*

$$\Gamma \parallel K \vdash \langle M_1[A_1, \mathsf{pc}_1]@h_1 \triangleright e_1\rangle \approx_\zeta \langle M_2[A_2, \mathsf{pc}_2]@h_2 \triangleright e_2\rangle$$

which holds if and only if the following conditions are all met:

(i) *There exist substitutions* γ_1*,* γ_2*,* k_1*, and* k_2 *and terms* e_1' *and* e_2' *such that* $e_1 = \gamma_1(k_1(e_1'))$ *and* $e_2 = \gamma_2(k_2(e_2'))$*.*

(ii) *Either (a)* $\mathsf{pc}_1 = \mathsf{pc}_2 \sqsubseteq \zeta$ *and* $h_1 = h_2$ *and* $e_1' \equiv_\alpha e_2'$ *or (b)* $\Gamma \parallel K\ [A, \mathsf{pc}_1]\ @h \ \vdash e_1'$ *and* $\Gamma \parallel K\ [A', \mathsf{pc}_2]\ @h' \vdash e_2'$ *and* $\mathsf{pc}_i \not\sqsubseteq \zeta$*.*

(iii) $\Gamma \vdash \gamma_1 \approx_\zeta \gamma_2$

(iv) $\Gamma \parallel K \vdash k_1 \approx_\zeta k_2$

(v) *$Loc(e_1) \subseteq Dom(M_1)$ and $Loc(e_2) \subseteq Dom(M_2)$ and $M_1 \approx_\zeta M_2$.*

(vi) *Both e'_1 and e'_2 satisfy the* **lgoto** *invariant.*

The Noninterference Invariant conditions spell out the intuition above. Clause (i) says that we can factor the important high-security information and low-security linear continuation out of the computations. Clause (ii) says that if the programs are evaluating under low-security pc, then the underlying computations being performed are the same, whereas when the pc is high, it shouldn't matter what the two computations do. Clauses (iii), (iv), and (vi) say collectively that we have factored out the relevant information. Finally, clause (v) says that the memories contain equivalent information from the low-security perspective.

The strategy for proving Noninterference is to show that evaluation preserves the invariant. When the program counter is low, equivalent configurations should evaluate in lock-step (modulo the values of high-security data). After the program has branched on high-security information (or jumped to a high-security continuation), the two configurations may temporarily get out of sync, but during that time, they may only affect high-security data. Eventually, when the program counter drops low again (via a linear continuation) both computations return to lock-step evaluation.

The first lemma we prove shows that the linear continuations do indeed get called in the order described by the linear context.

Lemma 4.11 (Linear Continuation Ordering) *Assume $K = y_n : k_n, \ldots, y_1 : k_1$ and each k_i is a linear continuation type and $\bullet \parallel K\ [A, \mathsf{pc}]\ @h\ \vdash e$. If $\bullet \vdash k \models K$ then in any well-formed configuration $\langle M[A, \mathsf{pc}]@h \triangleright k(e)\rangle$ the continuation $k(y_1)$ will be invoked before any of the $k(y_i)$.*

Proof: The operational semantics is valid for open terms and, indeed, Subject Reduction was proved for open terms. The Progress Lemma, however, does not hold for open terms. Evaluate the open term e in the configuration $\langle M[A, \mathsf{pc}]@h \triangleright e\rangle$. If the computation diverges, then none of the y_i's ever reach an active position, and hence are not invoked. Otherwise, the computation must get stuck (it can't halt because Subject Reduction implies that all configurations are well-typed, and the **halt** expression requires that the linear context be empty). The stuck term must be of the form **lgoto** $y_i\ v\ lv$, and, because this term is well-typed, rule $[TE7]$ implies that $y_i = y_1$. Thus, in the computation of $k(e)$, the continuation $k(y_1)$ is invoked first. □

Next, we prove that equivalent configurations evaluate in lock-step as long as the program counter is low.

Lemma 4.12 (Low-pc Step) *If $\Gamma \parallel K \vdash \langle M_1[A_1, \mathsf{pc}]@h \triangleright e_1\rangle \approx_\zeta \langle M_2[A_2, \mathsf{pc}]@h \triangleright e_2\rangle$ and $\mathsf{pc} \sqsubseteq \zeta$ and*

$$\langle M_1[A_1, \mathsf{pc}]@h \triangleright e_1\rangle \longmapsto \langle M'_1[A'_1, \mathsf{pc}_1]@h_1 \triangleright e'_1\rangle$$

then

$$\langle M_2[A_2, \mathsf{pc}]@h \triangleright e_2\rangle \longmapsto \langle M'_2[A'_2, \mathsf{pc}_2]@h_2 \triangleright e'_2\rangle$$

and there exist Γ' and K' such that $\Gamma' \parallel K' \vdash \langle M'_1[A'_1, \mathsf{pc}_1]@h_1 \triangleright e'_1\rangle \approx_\zeta \langle M'_2[A'_2, \mathsf{pc}_2]@h_2 \triangleright e'_2\rangle$.

Proof: Let $e_1 = \gamma_1(k_1(e''_1))$ and $e_2 = \gamma_2(k_2(e''_2))$ where the substitutions are as described by the conditions of the Noninterference Invariant. Note that because $\mathsf{pc} \sqsubseteq \zeta$, clause (ii) implies that e''_1 and e''_2 must be alpha-equivalent expressions. This means that the only difference in their operational behavior arises due to the substitutions being supplied, or the different memories. We proceed by cases on the transition step taken by the first program.

$[O1]$ By α-equivalence, each e''_i must be of the form **let** x = $prim$ **in** e. Conditions (iv), (v), and (vi) will hold after the transition step because those parts of the configuration don't change. It suffices to find γ'_i such that $e'_1 = \gamma'_1(k_1(e''_1))$ and $e'_2 = \gamma'_2(k_2(e''_2))$ where $e''_1 \equiv_\alpha e''_2$ Consider the evaluation $M_1[A, \mathsf{pc}]@h \vdash \gamma_1(prim) \Downarrow bv_\ell$. If $\ell \sqsubseteq \zeta$ then $prim$ cannot contain any free variables, for otherwise condition (iii) would be violated. Thus, $\gamma_1(prim) = prim = \gamma_2(prim)$, and because $M_1 \approx_\zeta M_2$ it follows that $M_2[A', \mathsf{pc}]@h \vdash \gamma_2(prim) \Downarrow bv_\ell$. Thus, we take $\gamma'_1 = \gamma_1$, $\gamma'_2 = \gamma_2$ and $e''_i = e\{bv_\ell/x\}$. Condition (iii) holds because it originally did; conditions (i) and (ii) are easily verified based on the operational semantics and the fact that $\mathsf{pc}_1 = \mathsf{pc}_2 = \mathsf{pc}$.

On the other hand, if $\ell \not\sqsubseteq \zeta$ then $M_2[A', \mathsf{pc}]@h \vdash \gamma_2(prim) \Downarrow bv'_{\ell'}$ where it is also the case that $\ell' \not\sqsubseteq \zeta$. ($prim$ either contains a variable, which forces ℓ to be high, or $prim$ contains a value explicitly labeled with a high-label.) It follows that $\bullet \vdash bv_\ell \approx_\zeta bv'_{\ell'}$ and so we take $\gamma'_1 = \gamma_1\{x \mapsto bv_\ell\}$ and $\gamma'_2 = \gamma_2\{x \mapsto bv'_{\ell'}\}$. Take $e''_i = e$. Condition (iii) continues to hold because the Primitive Evaluation lemma implies that the mapped values have the correct types and have high-labels. Conditions (i) and (ii) follow from the definition of the operational semantics and the fact that $e \equiv_\alpha e$, respectively.

$[O2]$ Each e''_i must be of the form **let** x = (**ref**(σ) $v_i)_\ell$ **in** e where $\gamma_i(v_i) = bv_{i\ell_i}$. By condition (ii) it follows that $v_1 \equiv_\alpha v_2$ and hence that $\gamma_1(v_1) \approx_\zeta \gamma_2(v_2)$. We may thus choose $M'_i = M_i[L^\sigma \mapsto \gamma_i(v_i)]$ and still maintain that $M'_1 \approx_\zeta M'_2$, which implies (v). Now, $e'_i = \gamma_i(k_i(e\{x/L^\sigma_{\ell \sqcup \mathsf{pc}}@h\}))$ and from the operational semantics, we have $\mathsf{pc}_1 = \mathsf{pc}_2 = \mathsf{pc}$ so that both conditions (i) and (ii) still hold. Conditions (iii), (iv), and (vi) are still satisfied.

$[O3]$ This case follows similarly to the previous two cases.

$[O4]$ In this case, each e''_i is of the form **if0** v **then** e_a **else** e_b. Furthermore, $\gamma_1(v_1) = 0_\ell$. In the case that $\ell \sqsubseteq \zeta$, it must be the case that $\gamma_2(v_2) = 0_\ell$, otherwise one of conditions (ii) or (iii) would be violated. The operational semantics thus allow us to choose $\gamma'_i = \gamma_i$, $k'_i = k_i$, $M'_i = M_i$, $\mathsf{pc}_1 = \mathsf{pc}_2 = \mathsf{pc} \sqcup \ell$, and $e'_1 = \gamma_1(k_1(e_a))$ and $e'_2 = \gamma_2(k_2(e_a))$. All of the conditions hold easily, as $\mathsf{pc} \sqcup \ell \sqsubseteq \zeta$.

In the case that $\ell \not\sqsubseteq \zeta$, it is possible for $\gamma_1(v)$ to equal 0_ℓ and $\gamma_2(v)$ to equal $n_{\ell'}$ where $n \neq 0$. In this case the first compuation takes the the first branch and the second compuation takes the second branch. However, it must be the case that $\ell' \not\sqsubseteq \zeta$, so in both computations the program counter after the step is $\not\sqsubseteq \zeta$, and we ma relate the resulting configurations via part (b) of clause (ii). Concretely, $\gamma'_i = \gamma_i$, $k'_i = k_i$, $M'_i = M_i$, $pc_1 = \mathsf{pc} \sqcup \ell$, $pc_2 = \mathsf{pc} \sqcup \ell'$, $e'_1 = \gamma_1(k_1(e_a))$ and e'_2 is either $\gamma_2(k_2(e_a))$ or $\gamma_2(k_2(e_b))$. These choices correspond to the operational semantics, and allow us to verify that all of the Noninterference Invariant conditions hold.

$[O5]$ This case is nearly identical to the previous one.

$[O6]$ Then each e''_i is of the form **letlin** y = **lcont**$@h'[A, \mathsf{pc}'](x : \sigma, y' : \kappa) = e'$ **in** e. Furthermore, because the term is well-typed under Γ and K, we have $K \equiv K_1, K_2$ such that the continuation typechecks under K_2 and e typechecks under $K_1, y : (\sigma', \kappa')$ lcont. If $\mathsf{pc}' \not\sqsubseteq \zeta$, then we simply take $e'_i = \gamma_i(k_i(e\{$**lcont**$@h'[A, \mathsf{pc}'](x : \sigma, y' : \kappa) = e'/y\}))$. If y appears as the target of **lgoto**, this satisfies invariant (vi). The rest of the invariants follow from Substitution II part (ii), the definition of the operational semantics, and the fact that the memories and program counters do not change.

Otherwise, $\mathsf{pc}' \sqsubseteq \zeta$. Let lv_i = **lcont**$@h'[A, \mathsf{pc}'](x : \sigma, y' : \kappa) = e'\{k_i(y_j)/y_j\}$ where the y_j are the members of $Dom(K_2)$. Note that $lv_1 \equiv_\alpha lv_2$. Let K' be $K_1, y : (\sigma', \kappa')$ lcont and let $k'_i = k_i|_{K_1}\{y \mapsto lv_i\}$. This choice of linear contexts satisfies part (iv) of the invariant. As above, the memories don't change, and the operational semantics justifies the choice of $e'_i = \gamma_i(k'_i(e))$, which is easily seen to satisfy the remaining invariants.

$[O7]$ In this case, each $e_i'' = \textbf{goto}\ v\ v'\ lv$. It must be the case that $\gamma_1(v) = (\textbf{cont}@h'\ f_B\ [A, \mathsf{pc}'](x\!:\!\sigma, y\!:\!\kappa) = e)_\ell$. If $\ell \sqsubseteq \zeta$, it follows that $v = (\textbf{cont}@h'\ f_B\ [A, \mathsf{pc}'](x\!:\!\sigma, y\!:\!\kappa) = e')_\ell$ where $e' = \gamma_1(e)$ because, by invariant (iii), the continuation could not be found in γ_1. There are two cases, depending on whether $\gamma_1(v')$ has label $\sqsubseteq \zeta$ or not. If so, then $\gamma_1(v') \approx_\zeta \gamma_2(v')$. It thus suffices to take $\Gamma' = \Gamma$, $K' = K$, and leave the substitutions unchanged, for we have $e_i' = \gamma_i(k_i(e\{v/f\}\{\gamma_i(v') \sqcup \mathsf{pc} \sqcup \ell/x\}\{lv/y\}))$. If the label of $\gamma_1(v') \not\sqsubseteq \zeta$, we take $\Gamma' = \Gamma, x\!:\!\sigma$ and $\gamma_i' = \gamma_i\{x \mapsto \gamma_i(v') \sqcup \mathsf{pc} \sqcup \ell\}$. The necessary constraints are then met by $e_i' = \gamma_i'(k_i(e\{v/f\}\{lv/y\}))$.

The other possibility is that $\ell \not\sqsubseteq \zeta$. From α-equivalence it follows that the label of $\gamma_2(v)$ is also $\not\sqsubseteq \zeta$. Thus, we have $\mathsf{pc}_1 = \mathsf{pc} \sqcup \ell \not\sqsubseteq \zeta$ and similarly, $\mathsf{pc}_2 \not\sqsubseteq \zeta$. This implies that the resulting configurations may use part (b) of clause (ii), and hence it does not matter what the bodies of the continuations being invoked are, as long as the other invariants are maintained. Clause (v) is easily seen to be satisfied, because the memory does not change. Similarly, clause (vi) follows by the conditions of clause (iii) of the source configurations. We build the new value substitutions as in the previous paragraph, depending on the label of $\gamma_1(v')$. Note that the same linear substitutions will suffice after the transition, which is evident from the Substitution Lemma II and the fact that both continuations must consume all of the linear resources.

$[O8]$ Each e_i'' must be of the form $\textbf{lgoto}\ lv\ v\ lv'$. If lv is a variable, y, then the result follows easily from the fact that $k_1(y) \equiv_\alpha k_2(y)$. We take K' to be the prefix of K excluding y (which must be the last variable in the list, by typing rule $[TE7]$). Then $k_i' = k_i|_{K'}$, and the rest of the invariants follow similarly to the case of normal continuations discussed above.

If lv is not a variable, then by clause (vi) of the invariant, the program counter being jumped to his $\not\sqsubseteq \zeta$, and, by α-equivalence, this holds for both configurations. The rest of the invariants are seen to hold as in the case for normal continuations, but this time, $K' = K$.

□

Next, we use the ordering lemma to prove that related high-security configurations stay related by a transition step. The tricky case is when the program counter becomes low.

Lemma 4.13 (High-pc Step) *If* $\Gamma \parallel K \vdash \langle M_1[A_1, \mathsf{pc}_1]@h_1 \triangleright e_1\rangle \approx_\zeta \langle M_2[A_2, \mathsf{pc}_2]@h_2 \triangleright e_2\rangle$ *and* $\mathsf{pc}_1, \mathsf{pc}_2 \not\sqsubseteq \zeta$ *then*

$$\langle M_1[A_1, \mathsf{pc}_1]@h_1 \triangleright e_1\rangle \longmapsto \langle M_1'[A_1', \mathsf{pc}_1']@h_1' \triangleright e_1'\rangle$$

implies that either e_2 *diverges or*

$$\langle M_2[A_2, \mathsf{pc}_2]@h_2 \triangleright e_2\rangle \longmapsto^* \langle M_2'[A_2', \mathsf{pc}_2']@h_2' \triangleright e_2'\rangle$$

and there exist Γ' *and* K' *such that* $\Gamma' \parallel K' \vdash \langle M_1'[A_1', \mathsf{pc}_1']@h_1' \triangleright e_1'\rangle \approx_\zeta \langle M_2'[A_2', \mathsf{pc}_2']@h_2' \triangleright e_2'\rangle$

Proof: The proof is by cases on the transition step taken by the first configuration. The intention is that because $\mathsf{pc}_1 \not\sqsubseteq \zeta$ and all of the transition rules except $[O8]$ increase the label of the program counter we may choose zero steps for the second configuration and still show that $\approx_\zeta$ is preserved. Condition (ii) of the Noninterference Invariant will hold via part (b). It suffices to show that in all cases except for $[O8]$ we simply choose $\Gamma' = \Gamma$ and $K' = K$ and show that the remainder of the invariants can still be met. This follows easily because all of the values computed, memory locations written to, and memory locations created must have labels at least as high as pc_1 (and hence $\not\sqsubseteq \zeta$). Thus the only memory locations affected are high-security and it follows easily that $M_1' \approx_\zeta M_2$. Similarly, the conditions on the **letlin** rule force any linear continuation introduced by e_1 to contain a program counter annotation that is $\not\sqsubseteq \zeta$. We may thus substitute it in e_1 without violating clause (vi) of the invariant. Thus for all cases except $[O8]$ it is easy to show that $\Gamma \parallel K \vdash \langle M_1'[A_1', \mathsf{pc}_1']@h_1' \triangleright e_1' \rangle \approx_\zeta \langle M_2[A_2, \mathsf{pc}_2]@h_2 \triangleright e_2 \rangle$ where we pick 0 steps for the second configuration.

We now consider the case for $[O8]$. Let $e_1 = \gamma_1(k_1(e_1''))$ Then $e_1'' =$ **lgoto** $lv\ v_1\ lv_1$ for some lv. In the case that lv is not a variable, clause (vi) of the invariant ensures that the program counter in the body of lv is $\not\sqsubseteq \zeta$. We may thus use the trick above of picking 0 steps for the second configuration, and it easily follows that the resulting configurations are $\approx_\zeta$ under Γ and K. If lv is a variable, y, then the typing rule for **lgoto** guarantees that $K = K', y : \kappa$. By assumption, $k_1(y) = ($**lcont**$@h[A, \mathsf{pc}](x : \sigma, y' : \kappa') = e)$. Where $\mathsf{pc} \sqsubseteq \zeta$. Assume e_2 does not diverge. Then by the Linear Continuation Ordering Lemma, we have $\langle M_2[A_2, \mathsf{pc}_2]@h_2 \triangleright e_2 \rangle \longmapsto^* \langle M_2'[A_2', \mathsf{pc}_2']@h_2' \triangleright$ **lgoto** $k_2(y)\ v_2\ lv_2 \rangle$. A simple induction on the length of this derivation sequence shows that $M_2 \approx_\zeta M_2'$ because the program counter in this sequence may not become $\sqsubseteq \zeta$. We thus have $M_1' = M_1 \approx_\zeta M_2 \approx_\zeta M_2'$. By invariant (iv) $k_2(y) \equiv_\alpha k_1(y)$. Furthermore, the typing rules require that $\mathsf{label}(\sigma) \not\sqsubseteq \zeta$. Thus we may take $\Gamma' = \Gamma, x : \sigma$, $\gamma_1' = \gamma_1\{x \mapsto \gamma_1(v_1) \sqcup \mathsf{pc}_1\}$, $\gamma_2' = \gamma_2\{x \mapsto \gamma_2(v_2) \sqcup \mathsf{pc}_2\}$. We take k_1' and k_2' to be the restrictions of k_1 and k_2 to the domain of K'. We may then choose $e_1' = \gamma_1'(k_1'(e))$ and $e_2' = \gamma_2'(k_2'(e))$. All of the necessary conditions are satisfied as is easily verified via the operational semantics, and so we are done. □

Finally, we put these lemmas together and use induction to obtain the noninterference result.

Theorem 4.2 (Noninterference) *Suppose* $x : \sigma \parallel y : (\mathsf{int}_\zeta, 1)\ \mathsf{lcont}\ [A, \mathsf{pc}]\ @h\ \vdash e$ *and* $\mathsf{label}(\sigma) \not\sqsubseteq \zeta$. *We also assume that* e *is an initial program. Further suppose that* $\bullet \vdash v_1, v_2 : \sigma$. *Let* $\mathit{stop} =$ **lcont**$@h'[\emptyset, \zeta](x : \mathsf{int}_\zeta, y : 1) =$ **halt**$_{\mathsf{int}_\zeta}\ x$. *Then*

$$\langle \emptyset[A, \mathsf{pc}]@h \triangleright e\{v_1/x\}\{\mathit{stop}/y\} \rangle \longmapsto^* \langle M_1[\emptyset, \zeta]@h' \triangleright \textbf{halt}_{\mathsf{int}_\zeta}\ n_{\ell_1} \rangle$$
$$\mathit{and}$$
$$\langle \emptyset[A, \mathsf{pc}]@h \triangleright e\{v_2/x\}\{\mathit{stop}/y\} \rangle \longmapsto^* \langle M_2[\emptyset, \zeta]@h' \triangleright \textbf{halt}_{\mathsf{int}_\zeta}\ m_{\ell_2} \rangle$$

implies that $M_1 \approx_\zeta M_2$ *and* $n = m$.

Proof: It is easy to verify that

$$x : \sigma \parallel y : (\mathsf{int}_\zeta, 1)\ \mathsf{lcont} \vdash \langle \emptyset[A, \mathsf{pc}]@h \triangleright e\{v_1/x\}\{\mathit{stop}/y\} \rangle \approx_\zeta \langle \emptyset[A, \mathsf{pc}]@h \triangleright e\{v_2/x\}\{\mathit{stop}/y\} \rangle$$

by letting $\gamma_1 = \{x \mapsto v_1\}$, $\gamma_2 = \{x \mapsto v_2\}$, and $k_1 = k_2 = \{y \mapsto \mathit{stop}\}$. Induction on the length of the evaluation sequence of the first expression, using the Low- and High-pc Step lemmas plus the fact that the second evaluation sequence terminates implies that it must be the case that

$$\Gamma \parallel K \vdash \langle M_1[\emptyset, \zeta]@h' \triangleright \textbf{halt}_{\mathsf{int}_\zeta}\ n_{\ell_1} \rangle \approx_\zeta \langle M_2[\emptyset, \zeta]@h' \triangleright \textbf{halt}_{\mathsf{int}_\zeta}\ m_{\ell_2} \rangle$$

Clause (v) of the definition of the Noninterference Invariant implies that $M_1 \approx_\zeta M_2$. Soundness of the operational semantics implies that $\ell_1 \sqsubseteq \zeta$ and $\ell_2 \sqsubseteq \zeta$. This means, because of clause (iii) neither n_{ℓ_1} nor m_{ℓ_2} are contained in the range of γ_i'. Thus, the integer values present in the **halt** expressions do not arise from substitution. Because $\zeta \sqsubseteq \zeta$, clause (ii) then implies that **halt**$_{\mathsf{int}_\zeta}\ n_{\ell_1} \equiv_\alpha$ **halt**$_{\mathsf{int}_\zeta}\ m_{\ell_2}$ from which we obtain $n = m$ as desired. □

5 Translation

This section discusses the translation rules for converting an un-located program to one which contains host annotations.

The source language is obtained from the located variant by making the additional assumption that there is a single host h, for which $\mathsf{TP}(h) = \mathsf{P}$. It is easy to verify that with this additional assumption, all of the label inequalities involving $\mathsf{TP}(h)$ become trivially true. To make it clear which inference rules and expressions are from the source language, we drop all occurrences of the $@h$ marker from them.

The goal of the partitioning translation is to find suitable conditions under which it is safe to locate a particular computation or value at a host. Our approach is to specify a set of nondeterministic rewrite rules which are guaranteed to produce valid partitions of the code, and then design algorithms which apply the rewrite rules based on trade-offs between network communication overhead and the cost or availability of trusted machines. Note that for some host machine configurations there may not be a valid partitioning.

The benefit of formalizing the degree to which hosts are trusted in the programming language is that the typing rules guide the development of a sound partitioning translation. In fact, the translation rules can be generated nearly automatically from the typing rules of the target language. Here we describe how to generate an appropriate set of rewrite rules, and illustrate the process with a few examples.

Secure values translate to themselves with the exception that continuations (both linear and nonlinear) are translated by locating them on hosts suitable for the translation of their bodies.

Primitive computations must take place on a single host, thus the primitive $prim$ may be located at a host h if all of the values mentioned in the primitive may be located there and the $\mathsf{TP}(h)$ is such that $\Gamma\ [A, \mathsf{pc}]\ @h\ \vdash prim' : \sigma$. For example, the rule for translating primitive values is given by:

$$\frac{\begin{array}{c}\Gamma \vdash v : \sigma \quad \Longrightarrow \quad \Gamma\ @h\ \vdash v' : \sigma \\ \mathsf{pc}' \sqsubseteq \mathsf{label}(\sigma)\end{array}}{\begin{array}{l}\Gamma\ [A, \mathsf{pc}]\ \vdash v : \sigma \\ \quad \Longrightarrow \\ \Gamma\ [A', \mathsf{pc}']\ @h\ \vdash v' : \sigma\end{array}}$$

Here, the symbol $\Longrightarrow$ indicates that the source judgment on the left of the arrow translates to the target judgment on the right, and we drop the $@h$ from source judgments.

The strategy for expressions is to provide several rewrite rules corresponding to whether or not subexpressions can be located on the same host. Consider the source expression **let** x = $prim$ **in** e. In the case that the subexpression e can be located at the same host as $prim$ there is no need to split the computation across hosts:

$$\frac{\begin{array}{c}\Gamma\ [A, \mathsf{pc}]\ \vdash prim : \sigma \quad \Longrightarrow \quad \Gamma\ [A', \mathsf{pc}']\ @h\ \vdash prim : \sigma \\ \Gamma, x\!:\!\sigma \parallel K\ [A, \mathsf{pc}]\ \vdash e \quad \Longrightarrow \quad \Gamma, x\!:\!\sigma \parallel K\ [A', \mathsf{pc}']\ @h\ \vdash e'\end{array}}{\begin{array}{l}\Gamma \parallel K\ [A, \mathsf{pc}]\ \vdash \textbf{let}\ x = prim\ \textbf{in}\ e \\ \quad \Longrightarrow \\ \Gamma \parallel K'\ [A', \mathsf{pc}']\ @h\ \vdash \textbf{let}\ x = prim'\ \textbf{in}\ e'\end{array}}$$

If e cannot be located on the same host as $prim$, we use a linear continuation to split the computation across hosts:

$$\frac{\begin{array}{c}\Gamma\ [A, \mathsf{pc}]\ \vdash prim : \sigma \quad \Longrightarrow \quad \Gamma\ [A', \mathsf{pc}']\ @h\ \vdash prim' : \sigma \\ \Gamma, x\!:\!\sigma \parallel K\ [A, \mathsf{pc}]\ \vdash e \quad \Longrightarrow \quad \Gamma, x\!:\!\sigma \parallel K\ [A'', \mathsf{pc}'']\ @h'\ \vdash e' \\ A'' \subseteq A' \subseteq \mathsf{TP}(h) \cap A \quad \mathsf{pc} \sqcup \langle \emptyset, \mathsf{TP}(h) \rangle \sqsubseteq \mathsf{pc}' \sqsubseteq \mathsf{pc}'' \sqsubseteq \langle \mathsf{TP}(h'), \emptyset \rangle \\ z \notin FV(e) \quad y \notin FLV(e)\end{array}}{\begin{array}{l}\Gamma \parallel K\ [A, \mathsf{pc}]\ \vdash \textbf{let}\ x = prim\ \textbf{in}\ e \\ \quad \Longrightarrow \\ \Gamma \parallel K\ [A', \mathsf{pc}']\ @h\ \vdash \\ \textbf{let}\ x = prim'\ \textbf{in} \\ \textbf{letlin}\ k = \textbf{lcont}@h'[A'', \mathsf{pc}''](z\!:\!1_{\mathsf{pc}'}, y\!:\!1) = \\ \qquad \textbf{in} \\ \textbf{lgoto}\ k\ \langle\rangle_{\mathsf{pc}'}\ \langle\rangle\end{array}}$$

The constraints on authority and program counter labels were obtained by assuming typing derivations for e' and $prim'$ and then determining what extra conditions are required for the entire translated term to type-check. This rule also forces the translated term require no more authority than the source and that the target's program counter label be at least as restrictive as the source's.

Translation rules for all of the expressions can be created in this way, but space limitations prevent us from including all of them. See Appendix C for several more examples.

This procedure for generating the translation rules guarantees that the resulting term is well-typed. Furthermore, because no annotations on values are changed, the security properties of the source program carry over onto the target. Correctness of the translation is easy to show using a simulation argument: disregarding the host annotations on the target code and memory locations, a source program transition step corresponds to a sequence of transitions in the translated program. These observations are captured in the following theorems.

Theorem 5.1 (Translation Preserves Typing) *If* $\bullet \parallel \bullet\ [A, \mathsf{pc}]\ \vdash e$ *and*

$$\bullet \parallel \bullet\ [A, \mathsf{pc}]\ \vdash e \quad \Longrightarrow \quad \bullet \parallel \bullet\ [A', \mathsf{pc}']\ @h\ \vdash e'$$

then $\bullet \parallel \bullet\ [A', \mathsf{pc}']\ @h\ \vdash e'$ *and* $A' \subseteq A \cap \mathsf{TP}(h)$ *and* $\mathsf{pc} \sqcup \langle \emptyset, \mathsf{TP}(h) \rangle \sqsubseteq \mathsf{pc}'$.

Theorem 5.2 (Translation Correctness) *If* $\bullet \parallel \bullet\ [A, \mathsf{pc}]\ \vdash e_S$ *and* M_S *is a well-formed memory such that* $Loc(e_S) \subseteq Dom(M_S)$ *and memory locations in* $Dom(M_S)$ *can be assigned locations to create the well-formed memory* M_T*, and*

$$\bullet \parallel \bullet\ [A_S, \mathsf{pc}_S]\ \vdash e_S \quad \Longrightarrow \quad \bullet \parallel \bullet\ [A_T, \mathsf{pc}_T]\ @h\ \vdash e_T$$

then

$$\langle M_S[A_S, \mathsf{pc}_S] \triangleright e_S \rangle \longmapsto \langle M'_S[A'_S, \mathsf{pc}'_S] \triangleright e'_S \rangle$$

implies that

$$\langle M_T[A_T, \mathsf{pc}_T]@h \triangleright e_T \rangle \longmapsto^* \langle M'_T[A'_t, \mathsf{pc}'_T]@h' \triangleright e'_T \rangle$$

and

$$\bullet \parallel \bullet\ [A'_S, \mathsf{pc}'_S]\ \vdash e'_S \quad \Longrightarrow \quad \bullet \parallel \bullet\ [A'_T, \mathsf{pc}'_T]\ @h'\ \vdash e'_T$$

The substitution model of evaluation presented in this paper is standard for dealing with semantics of high level programs, but allowing variables **let**-bound on one host to be used on another is not realistic from the implementation standpoint. Fortunately, closure conversion provides a way of making explicit the data which must be sent to a remotely located continuation. There is one twist, however, in the security setting: our type system requires that arguments to a continuation on h be acceptable for the machine h. The problem arises when a continuation located at a high-security host is syntactically nested within the body of a low-security continuation.

$$\textbf{cont}@h_{lo}\ f\ [A, \mathsf{pc}](x\!:\!\sigma, y\!:\!\kappa) = \ldots (\textbf{cont}@h_{hi}\ g\ [A', \mathsf{pc}'](x'\!:\!\sigma', y'\!:\!\kappa') = \ldots z_{hi} \ldots) \ldots$$

If the inner, high-security piece of code (g) mentions a high-security variable(z_{hi}) from some outer context, the standard closure conversion algorithm will put z_{hi} into the closure for f, even though this is disallowed by the type system. To circumvent this problem, we intend to use a level of indirection. Rather than putting z_{hi} in f's closure, we will put in a (read-only) reference to z_{hi}.

6 Related work

There are three primary areas of research related to the work described in this paper: enforcement of information flow policies, systems supporting mobile processes, and systems supporting mobile code security.

There has been much research on end-to-end security policies for information flow and mandatory access control in multilevel secure systems. Most practical systems for enforcing such policies have opted for dynamic enforcement using a mix of mandatory and discretionary access control. A classic example is the Orange Book [DOD85], but others [Fen73, Fen74, MR92] have also used dynamic techniques.

Static analysis of information flow also has a long history, although it has not been as widely used. Denning [Den76, DD77] originally proposed a language with static checking, but it was never implemented. Palsberg and Ørbæk developed a simple type system for checking integrity [PO95]. Volpano, Smith and Irvine showed Denning's rules sound using standard programming language techniques [VSI96]. Heintze and Riecke [HR98] have shown that security labels can be applied to a the typed lambda calculus with reference types.Agat shows how to blind certain eliminate covert timing channels [Aga00] by translation.

Linear language constructs [Wad90, Wad93, Abr93] have been used to regulate resource consumption. Linear continuations have been introduced earlier, but in order to study their category-theoretic semantics [Fil92]. Riely et al. have developed a notion of partial typing that allows statements to be made about type correctness in a distributed system containing malicious hosts [RH99].

Amoeba and Sprite [DOKT91] are examples of operating systems that provide transparent distribution of programs to improve performance. Our translation is driven by security needs, however, and these operating systems do not enforce end-to-end security.

Emerald [BHJL86] and Obliq [Car95] are two examples of transparently distributed programming languages. However, distribution in accordance with information flow policies is not supported. Modern distributed interface languages such as CORBA [OMG91] or Java RMI [Jav99] also do not enforce end-to-end policies. while distribution is somewhat transparent, program code executes on the host at which its object resides. This model allows enforcement of discretionary access control, but not end-to-end security policies: confidential data may easily leak to untrusted hosts through which the computation is threaded.

The focus of the work in this paper is on protecting data against untrusted hosts by moving computation away from them when necessary. A major current thread of ongoing research focuses on the different problem of protecting hosts from mobile computation. These two threads of research seem likely to be complementary. The Java bytecode verifier [LY96] is an early and much-used example of these techniques. More recent work includes Proof Carrying Code [Nec97] and Typed Assembly Language [MWCG98].

7 Conclusions

Security-typed languages are an attractive security approach because of their ability to enforce end-to-end confidentiality and integrity policies. However, previous work in this area has suffered from important limitations that prevent practical application. In this paper, we have proposed a new security-typed language, SPL@, that addresses some of these limitations.

Previous security-typed languages assume that the host executing the program is trusted. In a single-host system, this assumption is reasonable, but computation increasingly is distributed across heterogeneously trusted hosts. Computation may be run securely (to the satisfaction of all participating principals) on a partially trusted host only if certain security invariants are satisfied, as identified in this paper.

Writing programs in an explicitly distributed form is likely not to scale to more dynamic networks and larger numbers of hosts. To address this issue, this paper introduces secure program partitioning, a new approach to writing secure distributed systems. Programs are compiled to SPL@ assuming complete trust in a single executing host, and then translated into programs that contain explicit host locations for various pieces of code.

This fine-grained partitioning would not be possible in previous security-typed languages, because the programs created by translation in these languages often contain apparent (but non-existent) information flows. Thus, these languages have an excessively restrictive type system for information flow. To address this problem, the SPL@ language introduces ordered linear continuations. The type system is designed in such a way that ordered linear continuations can only be called once and in a fixed order with respect to other linear continuations.

This work is only a first step. SPL@ is a simple calculus that needs to be more expressive. Support for object types and polymorphism, as in Jif [Mye99], would make the language more expressive. Many improvements of the rewrite rules are possible, and more investigation of partitioning heuristics is needed.

References

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 147–160, San Antonio, TX, USA, January 1999.

[Abr93] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[Aga00] Johan Agat. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, January 2000.

[BHJL86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. *Proceedings of the ACM Special Interest Group on Programming Languages*, 21(11):78–86, November 1986. *Proc. of OOPSLA '86*, edited by Norman Meyrowitz, September 1986, Portland, Oregon.

[Car95] Luca Cardelli. A language with distributed scope. In *Proc. 22th ACM Symp. on Principles of Programming Languages (POPL)*, pages 286–297, San Francisco, CA, January 1995.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

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

[DOD85] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*, DOD 5200.28-STD (The Orange Book) edition, December 1985.

[DOKT91] Fred Douglis, John K. Ousterhout, M. Frans Kaashoek, and Andrew S. Tanenbaum. A comparison of two distributed systems: Amoeba and sprite. *ACM Transactions on Computer Systems*, 4(4), Fall 1991.

[Fen73] J. S. Fenton. *Information Protection Systems*. PhD thesis, University of Cambridge, Cambridge, England, 1973.

[Fen74] J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, May 1974.

[Fil92] Andrzej Filinski. Linear continuations. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages*, 1992.

[FLR77] R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving multilevel security of a system design. *Proc. 6th ACM Symp. on Operating System Principles (SOSP), ACM Operating Systems Review*, 11(5):57–66, November 1977.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM '93 Conference on Programming Language Design and Implementation*, Albuquerque, New Mexico, June 1993.

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

[GM84] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. IEEE Symposium on Security and Privacy*, pages 75–86, April 1984.

[HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, San Diego, California, January 1998.

[Jav99] JavaSoft. Java remote method invocation. http://java.sun.com/products/jdk/rmi, 1999.

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine*. Addison-Wesley, Englewood Cliffs, NJ, May 1996.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, Saint-Malo, France, 1997.

[ML00] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 2000. To appear.

[MR92] M. D. McIlroy and J. A. Reeds. Multilevel security in the UNIX tradition. *Software—Practice and Experience*, 22(8):673–694, August 1992.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, San Diego, California, January 1998.

[Mye99] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, San Antonio, TX, USA, January 1999.

[Nec97] George C. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 106–119, January 1997.

[OMG91] OMG. *The Common Object Request Broker: Architecture and Specification*, December 1991. OMG TC Document Number 91.12.1, Revision 1.1.

[PO95] Jens Palsberg and Peter Ørbæk. Trust in the λ-calculus. In *Proc. 2nd International Symposium on Static Analysis*, number 983 in Lecture Notes in Computer Science, pages 314–329. Springer, September 1995.

[RH99] James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 93–104, San Antonio, TX, January 1999.

[SV98] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, San Diego, California, January 1998.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[Wad90] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Progarmming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.

[Wad93] Philip Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science*, volume 711 of *Lecture Notes in Computer Science*. Springer-Verlag, August-September 1993.

[WF92] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Rice University, June 1992.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. Preliminary version in Rice TR 91-160.

[ZM00] Steve Zdancewic and Andrew C. Myers. Confidentiality and integrity with untrusted hosts. Submitted for publication, 2000.

A Typing rules

$$\boxed{\vdash \sigma \leq \sigma'}$$

$$\frac{}{\vdash \tau \leq \tau}$$

$$\frac{A \subseteq A' \quad \ell' \sqsubseteq \ell \quad \vdash \sigma' \leq \sigma \quad \vdash \kappa' \leq \kappa}{[A, \ell](\sigma, \kappa)\ \mathsf{cont} \quad \leq \quad [A', \ell'](\sigma', \kappa')\ \mathsf{cont}}$$

$$\frac{\vdash \tau \leq \tau \quad \ell \sqsubseteq \ell'}{\vdash \tau_\ell \leq \tau'_{\ell'}}$$

$$\frac{\vdash \sigma \leq \sigma' \quad \vdash \sigma' \leq \sigma''}{\vdash \sigma \leq \sigma''}$$

$$\boxed{\vdash \kappa \leq \kappa'}$$

$$\frac{}{\vdash \kappa \leq \kappa}$$

$$\frac{\vdash \sigma' \leq \sigma \quad \vdash \kappa' \leq \kappa}{(\sigma, \kappa)\ \mathsf{lcont} \quad \leq \quad (\sigma', \kappa')\ \mathsf{lcont}}$$

$$\frac{\vdash \kappa \leq \kappa' \quad \vdash \kappa' \leq \kappa''}{\vdash \kappa \leq \kappa''}$$

$$\boxed{K \equiv K'}$$

$$\frac{}{(K_1, K_2), K_3 \equiv K_1, (K_2, K_3)} \qquad \frac{}{y\!:\!\mathsf{1} \equiv \bullet}$$

$$\frac{}{\bullet, K \equiv K} \qquad \frac{}{K, \bullet \equiv K}$$

$$\frac{K_1 \equiv K_2}{K_2 \equiv K_1}$$

$$\frac{K_1 \equiv K_2 \quad K_2 \equiv K_3}{K_1 \equiv K_3}$$

$$\frac{K_1 \equiv K'_1}{K_1, K_2 \equiv K'_1, K_2}$$

$$\frac{K_2 \equiv K'_2}{K_1, K_2 \equiv K_1, K'_2}$$

$\boxed{\Gamma \vdash v : \sigma}$

$[\mathit{TV1}]$ $$\frac{}{\Gamma \vdash n_\ell : \mathsf{int}_\ell}$$

$[\mathit{TV2}]$ $$\frac{}{\Gamma \vdash \langle\rangle_\ell : 1_\ell}$$

$[\mathit{TV3}]$ $$\frac{\vdash \mathsf{label}(\sigma)\ \mathbf{ok}\ @h}{\Gamma \vdash L^\sigma_\ell @h : \sigma\ \mathsf{ref}_\ell}$$

$[\mathit{TV4}]$ $$\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma}$$

$[\mathit{TV5}]$ $$\frac{\begin{array}{c} f, x \notin \mathit{Dom}(\Gamma) \\ \mathsf{pc} \sqsubseteq \langle \mathsf{TP}(h), \emptyset \rangle \\ \mathsf{label}(\sigma) \sqsubseteq \langle \mathsf{TP}(h), \emptyset \rangle \\ \sigma' = [A, \mathsf{pc}](\sigma, \kappa)\ \mathsf{cont}_\ell \\ \Gamma, f{:}\sigma', x{:}\sigma \parallel y{:}\kappa\ [A \cup B, \mathsf{pc}]\ @h\ \vdash e \end{array}}{\Gamma \vdash (\mathbf{cont}@h\ f_B\ [A, \mathsf{pc}](x{:}\sigma, y{:}\kappa) = e)_\ell : \sigma'}$$

$[\mathit{TV6}]$ $$\frac{\Gamma \vdash v : \sigma \quad \vdash \sigma \le \sigma'}{\Gamma \vdash v : \sigma'}$$

$\boxed{\Gamma\ @h\ \vdash v : \sigma}$

$[\mathit{TV7}]$ $$\frac{\Gamma \vdash v : \tau_\ell \quad \ell \sqsubseteq \langle \mathsf{TP}(h), \emptyset \rangle}{\Gamma\ @h\ \vdash v : \tau_\ell}$$

$\boxed{\Gamma \parallel K\ \vdash lv : \kappa}$

$[\mathit{TL1}]$ $$\frac{}{\Gamma \parallel \bullet\ \vdash \langle\rangle : 1}$$

$[\mathit{TL2}]$ $$\frac{}{\Gamma \parallel y{:}\kappa\ \vdash y : \kappa}$$

$[\mathit{TL3}]$ $$\frac{\begin{array}{c} x \notin \mathit{Dom}(\Gamma), y \notin \mathit{Dom}(K) \\ \mathsf{pc} \sqsubseteq \langle \mathsf{TP}(h), \emptyset \rangle \\ \mathsf{label}(\sigma) \sqsubseteq \langle \mathsf{TP}(h), \emptyset \rangle \\ \Gamma, x{:}\sigma \parallel y{:}\kappa, K\ [A, \mathsf{pc}]\ @h\ \vdash e \end{array}}{\Gamma \parallel K\ \vdash \mathbf{lcont}@h[A, \mathsf{pc}](x{:}\sigma, y{:}\kappa) = e : (\sigma, \kappa)\ \mathsf{lcont}}$$

$[\mathit{TL4}]$ $$\frac{\Gamma \parallel K\ \vdash lv : \kappa \quad \vdash \kappa \le \kappa'}{\Gamma \parallel K\ \vdash lv : \kappa'}$$

$[\mathit{TL5}]$ $$\frac{\Gamma \parallel K\ \vdash lv : \kappa \quad K \equiv K'}{\Gamma \parallel K'\ \vdash lv : \kappa}$$

$\boxed{\Gamma\,[A,\mathsf{pc}]\ @h\ \vdash prim : \sigma}$

$$[\mathit{TP1}]\qquad \frac{\Gamma\ @h\ \vdash v : \sigma \quad \mathsf{pc} \sqsubseteq \mathsf{label}(\sigma)}{\Gamma\,[A,\mathsf{pc}]\ @h\ \vdash v : \sigma}$$

$$[\mathit{TP2}]\qquad \frac{\Gamma\ @h\ \vdash v : \mathsf{int}_\ell \quad \Gamma\ @h\ \vdash v' : \mathsf{int}_{\ell'} \quad \mathsf{pc} \sqsubseteq \ell \sqcup \ell'}{\Gamma\,[A,\mathsf{pc}]\ @h\ \vdash v \oplus v' : \mathsf{int}_{\ell\sqcup\ell'}}$$

$$[\mathit{TP3}]\qquad \frac{\Gamma\ @h\ \vdash v : \sigma\ \mathsf{ref}_\ell \quad \mathsf{pc} \sqsubseteq \mathsf{label}(\sigma) \sqcup \ell \sqsubseteq \langle \mathsf{TP}(h), \emptyset\rangle}{\Gamma\,[A,\mathsf{pc}]\ @h\ \vdash \textbf{deref}(v) : \sigma \sqcup \ell}$$

$$[\mathit{TP4}]\qquad \frac{\Gamma\ @h\ \vdash v : \tau_{\ell'} \quad B \subseteq A \quad \ell' \sqcap \langle \mathsf{P}, B\rangle \sqsubseteq \ell \sqcup \langle B, \mathsf{P}\rangle \quad \mathsf{pc} \sqsubseteq \langle \mathsf{P}, B\rangle \quad \mathsf{pc} \sqsubseteq \ell}{\Gamma\,[A,\mathsf{pc}]\ @h\ \vdash \textbf{declassify}(v,\ell) : \tau_\ell}$$

$\boxed{\Gamma \parallel K\ [A, \mathsf{pc}]\ @h \vdash e}$

$$[\mathit{TE1}] \qquad \frac{\Gamma\ [A, \mathsf{pc}]\ @h \vdash \mathit{prim} : \sigma \quad \Gamma, x\!:\!\sigma \parallel K\ [A, \mathsf{pc}]\ @h \vdash e}{\Gamma \parallel K\ [A, \mathsf{pc}]\ @h \vdash \textbf{let}\ x = \mathit{prim}\ \textbf{in}\ e}$$

$$[\mathit{TE2}] \qquad \frac{\Gamma\ @h \vdash v : \sigma \quad \mathsf{pc} \sqsubseteq \ell \sqsubseteq \langle \mathsf{TP}(h), \emptyset \rangle \quad \mathsf{pc} \sqsubseteq \mathsf{label}(\sigma) \quad \Gamma, x\!:\!\sigma\ \mathsf{ref}_\ell \parallel K\ [A, \mathsf{pc}]\ @h \vdash e}{\Gamma \parallel K\ [A, \mathsf{pc}]\ @h \vdash \textbf{let}\ x = (\textbf{ref}(\sigma)\ v)_\ell\ \textbf{in}\ e}$$

$$[\mathit{TE3}] \qquad \frac{\Gamma\ @h \vdash v : \sigma\ \mathsf{ref}_\ell \quad \Gamma\ @h \vdash v' : \sigma \quad \mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{label}(\sigma) \quad \Gamma \parallel K\ [A, \mathsf{pc}]\ @h \vdash e}{\Gamma \parallel K\ [A, \mathsf{pc}]\ @h \vdash \textbf{set}\ v := v'\ \textbf{in}\ e}$$

$$[\mathit{TE4}] \qquad \frac{\Gamma\ @h \vdash v : \mathsf{int}_\ell \quad \Gamma \parallel K\ [A, \mathsf{pc} \sqcup \ell]\ @h \vdash e_i \quad \langle \emptyset, \mathsf{TP}(h) \rangle \sqsubseteq \mathsf{pc}}{\Gamma \parallel K\ [A, \mathsf{pc}]\ @h \vdash \textbf{if0}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2}$$

$$[\mathit{TE5}] \qquad \frac{\begin{array}{c} \Gamma \parallel K_2 \vdash \textbf{lcont}@h'[A', \mathsf{pc}'](x'\!:\!\sigma', y'\!:\!\kappa') = e' : \kappa'' \\ \kappa'' = (\sigma, \kappa)\ \mathsf{lcont} \qquad A' \subseteq A \qquad \mathsf{pc} \sqsubseteq \mathsf{pc}' \\ \Gamma \parallel K_1, y\!:\!\kappa''\ [A, \mathsf{pc}]\ @h \vdash e \end{array}}{\Gamma \parallel K_1, K_2\ [A, \mathsf{pc}]\ @h \vdash \textbf{letlin}\ y = \textbf{lcont}@h'[A', \mathsf{pc}'](x'\!:\!\sigma', y'\!:\!\kappa') = e'\ \textbf{in}\ e}$$

$$[\mathit{TE6}] \qquad \frac{\begin{array}{c} \Gamma\ @h \vdash v : [A', \mathsf{pc}'](\sigma, \kappa)\ \mathsf{cont}_\ell \\ \Gamma\ @h \vdash v' : \sigma \quad \mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{label}(\sigma) \\ \Gamma \parallel K \vdash lv : \kappa \\ \langle \emptyset, \mathsf{TP}(h) \rangle \sqsubseteq \mathsf{pc} \quad \mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{pc}' \quad A' \subseteq A \subseteq \mathsf{TP}(h) \end{array}}{\Gamma \parallel K\ [A, \mathsf{pc}]\ @h \vdash \textbf{goto}\ v\ v'\ lv}$$

$$[\mathit{TE7}] \qquad \frac{\begin{array}{c} \Gamma \parallel K_2 \vdash lv_1 : (\sigma, \kappa)\ \mathsf{lcont} \\ \Gamma\ @h \vdash v : \sigma \\ \Gamma \parallel K_1 \vdash lv_2 : \kappa \\ \langle \emptyset, \mathsf{TP}(h) \rangle \sqsubseteq \mathsf{pc} \sqsubseteq \mathsf{label}(\sigma) \qquad A \subseteq \mathsf{TP}(h) \end{array}}{\Gamma \parallel K_1, K_2\ [A, \mathsf{pc}]\ @h \vdash \textbf{lgoto}\ lv_1\ v\ lv_2}$$

$$[\mathit{TE8}] \qquad \frac{\begin{array}{cc} B \subseteq A & \mathsf{pc} \sqcap \langle \mathsf{P}, B \rangle \sqsubseteq \mathsf{pc}' \sqcup \langle B, \mathsf{P} \rangle \\ \Gamma \parallel K\ [A, \mathsf{pc}']\ @h \vdash e & \vdash \mathsf{pc}'\ \textbf{ok}\ @h \end{array}}{\Gamma \parallel K\ [A, \mathsf{pc}]\ @h \vdash \textbf{setpc}\ (\mathsf{pc}')\ \textbf{in}\ e}$$

$$[\mathit{TE9}] \qquad \frac{\Gamma\ @h \vdash v : \sigma \quad \langle \emptyset, \mathsf{TP}(h) \rangle \sqsubseteq \mathsf{pc} \sqsubseteq \mathsf{label}(\sigma) \quad A \subseteq \mathsf{TP}(h)}{\Gamma \parallel \bullet\ [A, \mathsf{pc}]\ @h \vdash \textbf{halt}_\sigma\ v}$$

$$[\mathit{TE10}] \qquad \frac{\Gamma \parallel K\ [A, \mathsf{pc}]\ @h \vdash e \quad K \equiv K'}{\Gamma \parallel K'\ [A, \mathsf{pc}]\ @h \vdash e}$$

B Operational semantics

$$[P1] \quad \frac{\vdash \ell \sqcup \mathsf{pc}\ \mathbf{ok}\ @h}{M[A,\mathsf{pc}]@h \vdash bv_\ell \Downarrow bv_{\ell\sqcup\mathsf{pc}}}$$

$$[P2] \quad \frac{\vdash \ell \sqcup \ell' \sqcup \mathsf{pc}\ \mathbf{ok}\ @h}{M[A,\mathsf{pc}]@h \vdash n_\ell \oplus n'_{\ell'} \Downarrow (n[\![\oplus]\!]n')_{\ell\sqcup\ell'\sqcup\mathsf{pc}}}$$

$$[P3] \quad \frac{M(L^\sigma) = bv_{\ell'} \quad \vdash \ell \sqcup \ell' \sqcup \mathsf{pc}\ \mathbf{ok}\ @h}{M[A,\mathsf{pc}]@h \vdash \mathbf{deref}(L^\sigma_\ell @h') \Downarrow bv_{\ell\sqcup\ell'\sqcup\mathsf{pc}}}$$

$$[P4] \quad \frac{B \subseteq A \quad \ell' \sqcap \langle \mathsf{P}, B\rangle \sqsubseteq \ell \sqcup \langle B, \mathsf{P}\rangle \quad \mathsf{pc} \sqsubseteq \langle \mathsf{P}, B\rangle \quad \vdash \ell \sqcup \mathsf{pc}\ \mathbf{ok}\ @h}{M[A,\mathsf{pc}]@h \vdash \mathbf{declassify}(bv_{\ell'}, \ell) \Downarrow bv_{\ell\sqcup\mathsf{pc}}}$$

$$[O1] \quad \frac{M[A,\mathsf{pc}]@h \vdash prim \Downarrow v}{\langle M[A,\mathsf{pc}]@h \triangleright \mathbf{let}\ x = prim\ \mathbf{in}\ e\rangle \longmapsto \langle M[A,\mathsf{pc}]@h \triangleright e\{v/x\}\rangle}$$

$$[O2] \quad \frac{\vdash \mathsf{label}(\sigma)\ \mathbf{ok}\ @h \quad \ell \sqcup \mathsf{pc} \sqsubseteq \mathsf{label}(\sigma) \quad L^\sigma \notin Dom(M)}{\langle M[A,\mathsf{pc}]@h \triangleright \mathbf{let}\ x = (\mathbf{ref}(\sigma)\ bv_\ell)_{\ell'}\ \mathbf{in}\ e\rangle \longmapsto \langle M[L^\sigma \mapsto bv_{\ell\sqcup\mathsf{pc}}][A,\mathsf{pc}]@h \triangleright e\{L^\sigma_{\ell'\sqcup\mathsf{pc}}@h/x\}\rangle}$$

$$[O3] \quad \frac{\ell \sqcup \ell' \sqcup \mathsf{pc} \sqsubseteq \mathsf{label}(\sigma) \quad L^\sigma \in Dom(M)}{\langle M[A,\mathsf{pc}]@h \triangleright \mathbf{set}\ L^\sigma_\ell @h' := bv_{\ell'}\ \mathbf{in}\ e\rangle \longmapsto \langle M[L^\sigma \mapsto bv_{\ell\sqcup\ell'\sqcup\mathsf{pc}}][A,\mathsf{pc}]@h \triangleright e\rangle}$$

$$[O4] \quad \langle M[A,\mathsf{pc}]@h \triangleright \mathbf{if0}\ 0_\ell\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\rangle \longmapsto \langle M[A,\mathsf{pc}\sqcup\ell]@h \triangleright e_1\rangle$$

$$[O5] \quad \langle M[A,\mathsf{pc}]@h \triangleright \mathbf{if0}\ n_\ell\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\rangle \longmapsto \langle M[A,\mathsf{pc}\sqcup\ell]@h \triangleright e_2\rangle \qquad (n \neq 0)$$

$$[O6] \quad \langle M[A,\mathsf{pc}]@h \triangleright \mathbf{letlin}\ y = lv\ \mathbf{in}\ e\rangle \longmapsto \langle M[A,\mathsf{pc}]@h \triangleright e\{lv/y\}\rangle$$

$$[O7] \quad \frac{A' \cup B \subseteq \mathsf{TP}(h') \quad \vdash \mathsf{pc}'\ \mathbf{ok}\ @h'}{\begin{array}{l}\langle M[A,\mathsf{pc}]@h \triangleright \mathbf{goto}\ (\mathbf{cont}@h'\ f_B\ [A',\mathsf{pc}'](x\!:\!\sigma, y\!:\!\kappa) = e)_\ell\ bv_{\ell'}\ lv\rangle \\ \quad\longmapsto \\ \langle M[A'\cup B,\mathsf{pc}']@h' \triangleright e\{(\mathbf{cont}@h'\ f_B\ [A',\mathsf{pc}'](x\!:\!\sigma, y\!:\!\kappa) = e)_\ell/f\}\{bv_{\ell'\sqcup\ell\sqcup\mathsf{pc}}/x\}\{lv/y\}\rangle\end{array}}$$

$$[O8] \quad \frac{A' \subseteq \mathsf{TP}(h') \quad \vdash \mathsf{pc}'\ \mathbf{ok}\ @h'}{\begin{array}{l}\langle M[A,\mathsf{pc}]@h \triangleright \mathbf{lgoto}\ (\mathbf{lcont}@h'[A',\mathsf{pc}'](x\!:\!\sigma, y\!:\!\kappa) = e)\ bv_\ell\ lv\rangle \\ \quad\longmapsto \\ \langle M[A',\mathsf{pc}']@h' \triangleright e\{bv_{\ell\sqcup\mathsf{pc}}/x\}\{lv/y\}\rangle\end{array}}$$

$$[O9] \quad \frac{B \subseteq A \quad \mathsf{pc} \sqcap \langle \mathsf{P}, B\rangle \sqsubseteq \mathsf{pc}' \sqcup \langle B, \mathsf{P}\rangle \quad \vdash \ell\ \mathbf{ok}\ @h}{\langle M[A,\mathsf{pc}]@h \triangleright \mathbf{setpc}\ (\mathsf{pc}')\ \mathbf{in}\ e\rangle \longmapsto \langle M[A,\mathsf{pc}']@h \triangleright e\rangle}$$

C Example translation rules

$$\frac{}{\Gamma \vdash x : \sigma \quad \Longrightarrow \quad \Gamma \vdash x : \sigma}$$

$$\frac{}{\Gamma \vdash \langle\rangle_\ell : 1_\ell \quad \Longrightarrow \quad \Gamma \vdash \langle\rangle_\ell : 1_\ell}$$

$$\frac{\begin{array}{c} \mathsf{pc}' \sqsubseteq \langle \mathsf{TP}(h), \emptyset \rangle \\ \mathsf{label}(\sigma) \sqsubseteq \langle \mathsf{TP}(h), \emptyset \rangle \\ \sigma' = [A, \mathsf{pc}](\sigma, \kappa)\ \mathsf{cont}_\ell \qquad A' \subseteq A \\ \Gamma, f{:}\sigma', x{:}\sigma \parallel y{:}\kappa\ [A \cup B, \mathsf{pc}]\ \vdash e \quad \Longrightarrow \quad \Gamma, f{:}\sigma', x{:}\sigma \parallel y{:}\kappa\ [A' \cup B, \mathsf{pc}']\ @h\ \vdash e' \end{array}}{\begin{array}{l} \Gamma \vdash (\textbf{cont}\ f_B\ [A, \mathsf{pc}](x{:}\sigma, y{:}\kappa) = e)_\ell : \sigma' \\ \quad \Longrightarrow \\ \Gamma \vdash (\textbf{cont}@h\ f_B\ [A', \mathsf{pc}'](x{:}\sigma, y{:}\kappa) = e')_\ell : \sigma' \end{array}}$$

$$\frac{\begin{array}{c} \mathsf{pc}' \sqsubseteq \langle \mathsf{TP}(h), \emptyset \rangle \\ \mathsf{label}(\sigma) \sqsubseteq \langle \mathsf{TP}(h), \emptyset \rangle \\ \Gamma, x{:}\sigma \parallel y{:}\kappa, K\ [A, \mathsf{pc}]\ \vdash e \quad \Longrightarrow \quad \Gamma, x{:}\sigma \parallel y{:}\kappa, K\ [A', \mathsf{pc}']\ @h\ \vdash e' \end{array}}{\begin{array}{l} \Gamma \parallel K\ \vdash \textbf{lcont}[A, \mathsf{pc}](x{:}\sigma, y{:}\kappa) = e : (\sigma, \kappa)\ \mathsf{lcont} \\ \quad \Longrightarrow \\ \Gamma \parallel K\ \vdash \textbf{lcont}@h[A', \mathsf{pc}'](x{:}\sigma, y{:}\kappa) = e' : (\sigma, \kappa)\ \mathsf{lcont} \end{array}}$$

$$\frac{\begin{array}{c} \Gamma \parallel K_2\ \vdash lv_1 : (\sigma, \kappa)\ \mathsf{lcont} \quad \Longrightarrow \quad \Gamma \parallel K_2\ \vdash lv_1' : (\sigma, \kappa)\ \mathsf{lcont} \\ \Gamma \vdash v : \sigma \quad \Longrightarrow \quad \Gamma\ @h\ \vdash v' : \sigma \\ \Gamma \parallel K_1\ \vdash lv_2 : \kappa \quad \Longrightarrow \quad \Gamma \parallel K_1\ \vdash lv_2' : \kappa \\ \mathsf{pc} \sqcup \langle \emptyset, \mathsf{TP}(h) \rangle \sqsubseteq \mathsf{pc}' \sqsubseteq \mathsf{label}(\sigma) \qquad A' \subseteq \mathsf{TP}(h) \cap A \end{array}}{\begin{array}{l} \Gamma \parallel K_1, K_2\ [A, \mathsf{pc}]\ \vdash \textbf{lgoto}\ lv_1\ v\ lv_2 \\ \quad \Longrightarrow \\ \Gamma \parallel K_1, K_2\ [A', \mathsf{pc}']\ @h\ \vdash \textbf{lgoto}\ lv_1'\ v'\ lv_2' \end{array}}$$

$$\frac{\begin{array}{c} k_i, y_i \notin FLV(e_i) \qquad x_i \notin FV(e_i) \\ A_i \subseteq A' \subseteq \mathsf{TP}(h_i) \cap A \quad \mathsf{pc} \sqsubseteq \mathsf{pc}' \sqsubseteq \mathsf{pc}_i \sqcup \ell \sqsubseteq \langle \mathsf{TP}(h_i), \emptyset \rangle \\ A \subseteq \mathsf{TP}(h_i) \quad \mathsf{pc} \sqcup \ell \sqsubseteq \langle \mathsf{TP}(h_i), \emptyset \rangle \quad \Gamma\ @h\ \vdash v : \mathsf{int}_\ell \\ \Gamma \parallel K\ [A, \mathsf{pc} \sqcup \ell]\ \vdash e_i \quad \Longrightarrow \quad \Gamma \parallel K\ [A_i, \mathsf{pc}_i \sqcup \ell]\ @h_i\ \vdash e_i' \end{array}}{\begin{array}{l} \Gamma \parallel K\ [A, \mathsf{pc}]\ \vdash \textbf{if0}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \\ \quad \Longrightarrow \\ \Gamma \parallel K\ [A', \mathsf{pc}']\ @h\ \vdash \\ \textbf{if0}\ v\ \textbf{then} \\ \qquad \textbf{letlin}\ k_1 = \textbf{lcont}@h_1[A_1, \mathsf{pc}_1](x_1{:}1_{\mathsf{pc}}, y_1{:}1) = e_1'\ \textbf{in} \\ \qquad\qquad \textbf{lgoto}\ k_1\ \langle\rangle_{\mathsf{pc}'}\ \langle\rangle \\ \textbf{else} \\ \qquad \textbf{letlin}\ k_2 = \textbf{lcont}@h_2[A_2, \mathsf{pc}_2](x_2{:}1_{\mathsf{pc}}, y_2{:}1) = e_2'\ \textbf{in} \\ \qquad\qquad \textbf{lgoto}\ k_2\ \langle\rangle_{\mathsf{pc}'}\ \langle\rangle \end{array}} \quad (i \in \{1, 2\})$$