3-1-2005

A Monadic Analysis of Information Flow Security with Mutable State

Karl Crary
Carnegie Mellon University, crary@cs.cmu.edu

Aleksey Kliger
Carnegie Mellon University

Frank Pfenning
Carnegie Mellon University, fp@cs.cmu.edu

JFP **15** (2): 249–291, 2005.
DOI: 10.1017/S0956796804005441 Printed in the United Kingdom

A monadic analysis of information flow security with mutable state

KARL CRARY, ALEKSEY KLIGER and FRANK PFENNING

Carnegie Mellon University, 5000 Forbes Avenue Pittsburgh, PA 15213 USA
(*e-mail:* {crary,aleksey,fp}@cs.cmu.edu)

Abstract

We explore the logical underpinnings of higher-order, security-typed languages with mutable state. Our analysis is based on a logic of information flow derived from lax logic and the monadic metalanguage. Thus, our logic deals with mutation explicitly, with impurity reflected in the types, in contrast to most higher-order security-typed languages, which deal with mutation implicitly via side-effects. More importantly, we also take a *store-oriented* view of security, wherein security levels are associated with elements of the mutable store. This view matches closely with the operational semantics of low-level imperative languages where information flow is expressed by operations on the store. An interesting feature of our analysis lies in its treatment of upcalls (low-security computations that include high-security ones), employing an "informativeness" judgment indicating under what circumstances a type carries useful information.

Capsule Review

Type-based information flow analyses enrich the traditional structure of types with security levels, so that a type tells not only how a value may be used, but also how much information its use may reveal. In Heintze and Riecke's SLam Calculus, every node in a type, be it an arrow, a product, or a sum, carries a security level. This makes things simple: to "taint" a type at level ℓ, one simply joins ℓ to the annotation found at its root. This convention is, however, verbose and redundant. In Abadi *et al.*'s Dependency Core Calculus, on the other hand, the standard type constructors carry no annotation: instead, a new type constructor, the "monad," is added for this purpose. This design is more orthogonal, but makes things more subtle: to taint a type at level ℓ, one must traverse it until places where an annotation can be attached are found. This process is defined by a "protectedness" predicate. The present paper proposes yet another way of attaching security annotations to types. Again, the standard type constructors carry no annotation: instead, only *reference* types carry an annotation. An "informativeness" predicate defines how types are tainted. This leads to a store-oriented view of information flow, which seems to make the most sense in the context of low-level languages, such as typed assembly language, where the store is everything. This original paper also provides a nice occasion to think about the many ways to design a type-based information flow analysis.

1 Introduction

Security-typed languages use type systems to track the flow of information within a program to provide properties such as secrecy and integrity. Secrecy states that

high-security information does not flow to low-security agents, and integrity dually states that low-security agents cannot corrupt high-security information. In this paper, we will restrict our attention to secrecy properties. A variety of security-typed languages have been proposed, and several of them are both higher-order (i.e. support first-class functions) and provide mutable state (Heintze & Riecke, 1998; Myers, 1999; Pottier & Simonet, 2003; Zdancewic & Myers, 2002).

However, when adopting one of these languages to the Typed Assembly Language (Morrisett *et al.*, 1999) setting, one faces a tension between the high-level view of information flowing from the values of sub-terms to the result value of a complete term and the assembly-language imperative view of instructions operating on a mutable store. What is needed is a typed calculus in which values have structure (i.e. like in high level languages) but information flows through the store (i.e. like in a low-level language).

In this paper, we explore this *store-oriented* view of information flow: one of the steps towards a TAL with information flow, we look at a language with a clean separation between values and computations. A suitable starting point is Moggi's monadic metalanguage (Moggi, 1989; Moggi, 1991) and its corresponding logic (via a Curry-Howard isomorphism).

Our presentation of lax logic is based on that of Pfenning and Davies (2001). The principal distinctive feature of Pfenning and Davies's account is a syntactic distinction between *terms* and *expressions,* where terms are pure and expressions are (possibly) effectful. They show that this distinction allows the logic to possess some desirable properties (local soundness and local completeness) that state in essence that the logic's presentation is canonical. Although our work inherits these properties, they are not particularly important here. However the term/expression distinction also provides a clean separation between the pure and effectful parts of our analysis, which greatly simplifies our system.

Our system bears some resemblance to the work of Abadi *et al.* (1999), who also use a monadic structure to reason about information flow. However whereas we use monads in a conventional manner to separate values from computations, they use a monad to endow values with a security level. It is not clear how to adopt their work to a low-level setting where the values and operations ought to correspond to those of a real machine.

A natural question is whether this store-oriented security discipline limits the expressive power of our account relative to ones based on a value-oriented discipline, but we show (in Section 6) that it does not.

Overview The static semantics of our analysis is based on two typing judgments, one for terms (M) and one for expressions (E). Recall that terms are pure and that security is associated with effects, so the typing judgment for terms makes no mention of security levels. Thus, the typing judgment takes the form $\Sigma; \Gamma \vdash M : A$ (where A is a type, Γ is the usual context and Σ assigns a type to the store).

Expressions, on the other hand, may have effects and therefore may interact with the security discipline. Each location in the store has a security level associated with it indicating the least security level that is authorized to read that location. Thus,

the typing judgment for expressions tracks the security levels of all locations an expression reads or writes. Only the reads are of direct importance to the security discipline (recall that we do not address integrity), but writes must also be tracked since they provide a means of information flow. The judgment takes the form: $\Sigma; \Gamma \vdash E \div_{(r,w)} A$ indicating that r is an upper bound to the levels of E's reads, and w is a *lower* bound to the levels of its writes, and also that E has type A. Naturally we require that $r \sqsubseteq w$, or else E could manifestly be leaking information.

Our language can be seen as a conservative extension to purely functional languages such as Haskell. Existing terms continue to be type-safe. On the other hand new effectful code that makes use of the security discipline can be cleanly separated.

In lax logic, expressions are internalized as terms using the monadic type $\bigcirc A$. Terms of type $\bigcirc A$ are suspended expressions of type A. Thus, the introduction form for the monadic type is a term construct, and the elimination form (which releases the suspended expression) is an expression construct. Similarly, our expressions are internalized as terms using a monadic type written $\bigcirc_{(r,w)} A$. Since the effects of the suspended expression will be released when the monad is eliminated, the levels of those effects must be recorded in the monad type.

Most of the rules in our account follow from the intuitions above. One remaining novelty deals with the information content of types. Ordinarily, an expression would be deemed to be leaking information if it were to read from a high-security location, use the result of the read to form a value, and pass that value to a low-security computation. However, that expression would *not* be leaking information if one could show that the type of that value contained no information, or contained information usable only by a high-security computation (who could have performed the read anyway). Thus the type system contains a judgment $\vdash A \nearrow a$ stating that the type A contains information only for computations at the level a at least. This notion of *informativeness* is essential to accounting for the key issue of upcalls (low-security computations that include high-security computations).

The rest of this paper is organized as follows. In section 2 we present our basic logical account, including static and dynamic semantics, but omitting the key issue of upcalls. In section 3 we extend our account to deal with upcalls. In section 5 we state and prove a non-interference theorem. In section 6 we show that our store-oriented account provides at least the expressive power of value-oriented accounts by embedding one value-oriented language into our language. Section 7 discusses some related work, section 8 offers some concluding remarks.

2 Secure monadic calculus

We now describe the syntax, typing rules and operational semantics of our language.

As in other work on information flow, we have in mind an arbitrary lattice $(\mathscr{L}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ of security levels.

Operation levels To track the flow of information, we classify expressions not only by the value that they return, but also by the security levels of their effects. In particular, we keep track of an *operation level* $o = (r, w)$, for each expression. The

$$
\begin{array}{rcll}
A, B, C & \in & \textit{types} & ::= \quad 1 \mid \mathsf{bool} \mid A \to B \\
 & & & \mid \mathsf{ref}_a\, A \mid \mathsf{refr}_a\, A \mid \mathsf{refw}_a\, A \\
 & & & \mid \bigcirc_o A \\
M, N & \in & \textit{terms} & ::= \quad x & \text{variables} \\
 & & & \mid * & \text{unit} \\
 & & & \mid \mathsf{true} \mid \mathsf{false} & \text{boolean values} \\
 & & & \mid \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 & \text{conditional} \\
 & & & \mid \lambda x : A.M & \text{abstraction} \\
 & & & \mid MN & \text{application} \\
 & & & \mid \ell & \text{store location} \\
 & & & \mid \mathsf{val}\ E & \text{suspended expression} \\
E, F & \in & \textit{expressions} & ::= \quad [M] & \text{return} \\
 & & & \mid \mathsf{let\ val}\ x = M\ \mathsf{in}\ E & \text{sequencing} \\
 & & & \mid \mathsf{ref}_a\,(M : A) & \text{store allocation} \\
 & & & \mid\ !M & \text{store read} \\
 & & & \mid M := N & \text{store write} \\
\\
\Gamma & \in & \textit{contexts} & ::= \quad \cdot \mid \Gamma, x : A \\
\Sigma & \in & \textit{store types} & ::= \quad \{\} \mid \Sigma\{\ell : A\} \\
\\
V & \in & \textit{values} & ::= \quad * \mid \mathsf{true} \mid \mathsf{false} \\
 & & & \mid \lambda x : A.M \mid \ell \mid \mathsf{val}\ E \\
H & \in & \textit{stores} & ::= \quad \{\} \mid H\{\ell \mapsto V\} \\
S & \in & \textit{comp. states} & ::= \quad (H, \Sigma, E)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{let}\ x = E\ \mathsf{in}\ F & \equiv & \mathsf{let\ val}\ x = \mathsf{val}\ E\ \mathsf{in}\ F \\
\mathsf{run}\ M & \equiv & \mathsf{let\ val}\ x = M\ \mathsf{in}\ [x]
\end{array}
$$

Fig. 1. Syntax.

security level r is an upper bound on the security levels of the store locations that the expression reads, while w is a lower bound on the security level of the store locations to which it writes.

Since expressions that write at a security level below their read level may obviously be insecure, henceforth we restrict our attention only to operation levels (r, w) with $r \sqsubseteq w$.

The operation levels have a natural ordering $(r, w) \preceq (r', w')$. Given some expression E, if it reads from level at most r, then it surely reads from level at most r', provided that $r \sqsubseteq r'$. Similarly, if it writes at level at least w, then it writes at level at least w', provided that $w' \sqsubseteq w$. That is, operation levels are covariant in the reads and contravariant in the writes: $(r, w) \preceq (r', w')$ iff $\big(r \sqsubseteq r'$ and $w' \sqsubseteq w\big)$.

2.1 Syntax

The full syntax of our language is given in Figure 1. The language is split into two syntactic categories: pure terms M that are evaluated to values V and expressions E that are executed for effect as part of compuation states S.

Terms At the term level, we have variables, unit, booleans and conditional terms, function abstractions and applications. For simplicity, we did not include a mechanism for defining recursive terms, although the inclusion of such a facility would not pose a problem. Store locations are also terms, with each location ℓ having a fixed security level $\mathsf{Level}(\ell)$. The store associates locations with the values they contain. A subtyping relation, allows us to treat store cells as either read-write, read-only, or write-only. The term $\mathsf{val}\ E$ allows expressions to be included at the term level as an element of the monadic type $\bigcirc_o A$. Since terms are pure, a $\mathsf{val}\ E$ does not execute the expression E, but rather represents a suspended computation.

Expressions The expressions include a trivial return expression $[M]$. The return expression has no effect, and simply returns the value to which M evaluates. In general, when an expression has no read effects, we say its read level is $\bot$, and if an expression has no write effects, we say its write level is $\top$. Accordingly, the operation level of $[M]$ is $(\bot, \top)$. Note that $(\bot, \top)$ is the least element in the $\preceq$ ordering, so our subsumption principle will let us weaken the operation level of $[M]$ to any operation level.

The sequencing expression $\mathsf{let\ val}\ x = M\ \mathsf{in}\ F$ evaluates M down to some $\mathsf{val}\ E$, and executes E followed by F. The return value of expression E is bound to the variable x in F. If E and F both have operation level o, then so does the sequencing expression.

We will often write $\mathsf{let}\ x = E\ \mathsf{in}\ F$ as syntactic sugar for $\mathsf{let\ val}\ x = \mathsf{val}\ E\ \mathsf{in}\ F$, and $\mathsf{run}\ M$ for $\mathsf{let\ val}\ y = M\ \mathsf{in}\ [y]$.

In addition, there are expressions that allocate, read from, and write to the store. A read expression $!M$ has operation level $(a, \top)$, where a is the security level of the store location being read, and returns the contents of the store location. Dually, a write expression $M := N$ has operation level $(\bot, a)$ and updates the store location with the value of N; it does not return an interesting value (i.e. it returns unit).

Store allocation $\mathsf{ref}_a\,(M : A)$ specifies the security level a and type A of the new store location.

Allocation cannot leak information in our setting. Evidently, it is not a read operation. Less obviously, it is not a write operation either. With a write, another expression may learn something about the current computation by observing a change in the value stored at a particular store location. However, the key to this scenario is that the same location is mentioned by more than one expression. On the other hand, allocation creates a new location that is not aliased. Thus, there can be no implicit flow of information via an allocation expression. As a result, allocation has operation level $(\bot, \top)$. Of course if there were a primitive mechanism in place to distinguish locations (for example by comparing locations for equality), allocation would once again be observable.

Although there is not a primitive mechanism for recursion at the level of expressions, recursion can be encoded at the level of expressions using back-patching, see an example in section 3.3.

Table 1. *Typing judgments*

Judgment	Meaning
$\Sigma;\Gamma \vdash M : A$	Term M has type A
$\Sigma;\Gamma \vdash E \div_o A$	Expression E has type A and operation level o
$\vdash A \leqslant B$	Type A is a subtype of B
$\vdash H : \Sigma$	Store H has type Σ
$\vdash S \div_o A$	Computation state S is well-typed

States A computation state is a partially executed program, and consists of a triple (H, Σ, E) of a store H, a store type Σ and a closed expression E. The store maps locations to values, and the store type maps locations to the types of those values.

We assume that in a state (H, Σ, E), the store binds occurrences of store locations ℓ in H and E, and we identify computation states up to level-preserving renaming of store locations. In addition, as usual, we identify all constructs up to renaming of bound variables.

2.2 Static semantics

The type system of our language consists of two main mutually recursive judgments for typing terms and expressions, and some judgments for typechecking stores, and computation statesthat are summarized in Table 1. The first judgment $\Sigma;\Gamma \vdash M : A$ says that the term M has type A in the context Γ, where the store has type Σ. The jugment for expressions $\Sigma;\Gamma \vdash E \div_o A$ says that E returns a value of type A and performs only operations within level o. Each rule is given with its rule number, and the full set of rules appears in Appendix A.2.

We assume that contexts Γ are well-formed, that is, they contain at most one occurrence of each variable x. We tacitly rename bound variables prior to adding them to a context to maintain well-formedness. Similarly, we assume that store types are well-formed, that is, they contain at most one occurrence of each store location ℓ.

Terms The typing rules for terms are unsurprising for a simply-typed lambda calculus with unit, bool and function types. A store location ℓ (provided that it is in $\mathrm{dom}(\Sigma)$) has type $\mathsf{ref}_{\mathsf{Level}(\ell)}\Sigma(\ell)$. A computation term $\mathsf{val}\ E$ has the type $\bigcirc_o A$, provided the expression E has type A and operation level o. The rules are given in Appendix A.2.

Expressions The typing rules for expressions (given in Figure 2) follow our informal description. Trivial computations have the type of their return value, and operation level $(\bot, \top)$ (rule 29). By rule (30), the sequencing expression $\mathsf{let\ val}\ x = M\ \mathsf{in}\ E$ is well-typed provided both of the sub-computations have the same operation level

$$\boxed{\Sigma;\Gamma \vdash E \div_o A}$$

$$\frac{\Sigma;\Gamma \vdash M : A}{\Sigma;\Gamma \vdash [M] \div_{(\bot,\top)} A}\ (29) \qquad \frac{\Sigma;\Gamma \vdash M : \bigcirc_o A \quad \Sigma;\Gamma, x : A \vdash E \div_o A}{\Sigma;\Gamma \vdash \mathsf{let\ val}\ x = M\ \mathsf{in}\ E \div_o A}\ (30)$$

$$\frac{\Sigma;\Gamma \vdash M : A}{\Sigma;\Gamma \vdash \mathsf{ref}_a\,(M : A) \div_{(\bot,\top)} \mathsf{ref}_a\,A}\ (31) \qquad \frac{\Sigma;\Gamma \vdash M : \mathsf{refr}_a\,A}{\Sigma;\Gamma \vdash\ !M \div_{(a,\top)} A}\ (32)$$

$$\frac{\Sigma;\Gamma \vdash M : \mathsf{refw}_a\,A \quad \Sigma;\Gamma \vdash N : A}{\Sigma;\Gamma \vdash M := N \div_{(\bot,a)} 1}\ (33)$$

$$\frac{\Sigma;\Gamma \vdash E \div_o A \quad o \preceq o'}{\Sigma;\Gamma \vdash E \div_{o'} A}\ (34) \qquad \frac{\Sigma;\Gamma \vdash E \div_o A \quad \vdash A \leqslant B}{\Sigma;\Gamma \vdash E \div_o B}\ (36)$$

Fig. 2. Typing rules (expressions).

$$\boxed{\vdash A \leqslant B}$$

$$\frac{\vdash A \leqslant B \quad a \sqsubseteq b}{\vdash \mathsf{refr}_a\,A \leqslant \mathsf{refr}_b\,B}\ (17) \qquad \frac{\vdash B \leqslant A \quad b \sqsubseteq a}{\vdash \mathsf{refw}_a\,A \leqslant \mathsf{refw}_b\,B}\ (18)$$

$$\frac{\vdash A \leqslant B \quad a \sqsubseteq b}{\vdash \mathsf{ref}_a\,A \leqslant \mathsf{refr}_b\,B}\ (15) \qquad \frac{\vdash B \leqslant A \quad b \sqsubseteq a}{\vdash \mathsf{ref}_a\,A \leqslant \mathsf{refw}_b\,B}\ (16) \qquad \frac{\vdash A \leqslant B \quad o \preceq o'}{\vdash \bigcirc_o A \leqslant \bigcirc_{o'} B}\ (14)$$

Fig. 3. Selected subtyping rules.

(which may require using rule (34) to weaken the operation level of the sub-computations). Allocation (rule 31) returns a new read/write store location. For read and write expressions (rules 32 and 33) we only require that the corresponding store location is readable or writable, respectively.

Subtyping Subsumption (rules 28, 36) allows us to weaken the type A of a term M or an expression E, provided A is a subtype of B. Selected subtyping rules are given in Figure 3.

Stores and states A store H is well-typed with store type Σ, provided that each value V_i in the store is well typed under Σ and the empty context, where Σ has the same domain as H. A computation state (H, Σ, E) is well-typed provided that the store and the expression are each well-typed with the same store type:

$$\frac{\mathrm{dom}(\Sigma) = \{\ell_1, \ldots, \ell_n\} \quad \Sigma;\cdot \vdash V_i : \Sigma(\ell_i)\ \text{ for } 1 \leqslant i \leqslant n}{\vdash \{\ell_1 \mapsto V_1, \ldots \ell_n \mapsto V_n\} : \Sigma}\ (37) \qquad \frac{\vdash H : \Sigma \quad \Sigma;\cdot \vdash E \div_o A}{\vdash (H, \Sigma, E) \div_o A}\ (38)$$

2.3 Operational semantics

A computation state is called *terminal* if it is of the form $(H, \Sigma, [V])$. An evaluation relation $S \to S'$ gives the small-step operational semantics for computation states.

We write $S \downarrow$ if for some terminal state S', $S \rightarrow^* S'$. Since terms are pure and do not have an effect on the store, their evaluation rules may be given simply by the relation $M \rightarrow M'$ (no store is required). The entire set of evaluation rules is given in Appendix B.

We write $M[N/x]$ and $E[N/x]$ for the capture-avoiding substitution of N for x in the term M or expression E. We write $H\{\ell \mapsto V\}$ for finite map that extends H with V at ℓ.

It is instructive to consider how a computation in state $S_0 = (H, \Sigma, \mathsf{let\ val}\ x = M\ \mathsf{in}\ F)$ would evaluate. There are three stages:

1. LETVAL1 is repeatedly applied until M is evaluated down to a value $\mathsf{val}\ E$, $S_1 = (H, \Sigma, \mathsf{let\ val}\ x = \mathsf{val}\ E\ \mathsf{in}\ F)$
2. LETVALVAL is then applied until the subcomputation (H, Σ, E) is evaluated to a terminal state $(H', \Sigma', [V])$, $S_2 = (H', \Sigma', \mathsf{let\ val}\ x = \mathsf{val}\ [V]\ \mathsf{in}\ F)$
3. LETVAL substitutes the value V for x in F and computation continues in state $S_2 = (H', \Sigma', F[V/x])$.

For the proof of non-interference (specifically for the proof of the Hexagon Lemma), it will be useful to have the following lemma. It says that if a term evaluates to a value (or if a computation state evaluates to a terminal state) then the syntactic subterms (or subexpressions) of the given term (or computation state) will likewise evaluate to values (or terminal states). That is, our account is call-by-value.

Lemma 2.1 (*Subterm/Subexpression Termination*)

- If $(H, \Sigma, E) \downarrow$ in n steps, then
  1. if $E = [M]$ then $M \rightarrow^n V$
  2. if $E = \mathsf{let\ val}\ x = M\ \mathsf{in}\ F$ then $M \rightarrow^k \mathsf{val}\ E'$, $(H, \Sigma, E') \downarrow$ in m steps and $k + m < n$
  3. if $E = \mathsf{ref}_a\,(M : A)$ then $M \rightarrow^k V$ and $k < n$
  4. if $E = !M$ then $M \rightarrow^k V$ and $k < n$
  5. if $E = M := N$ then $M \rightarrow^k V_1$, $N \rightarrow^m V_2$ and $k + m < n$
- If $M \rightarrow^n V$ then
  1. If $M = N_1 N_2$, then $N_1 \rightarrow^k V_1$ and $V_1 N_2 \rightarrow^m V_1 V_2$ and $k + m < n$
  2. If $M = \mathsf{if}\ N_1\ \mathsf{then}\ N_2\ \mathsf{else}\ N_3$ then $N_1 \rightarrow^k V_1$ and $k < n$

Proof
by induction on the number of steps in the evaluation relation, by cases on the last rule. □

Our operational semantics are deterministic up to renaming of store locations: recall that we consider store locations to be bound by the store in a computation state. We allow a bound store location ℓ to be renamed ℓ', as long as $\mathsf{Level}(\ell) = \mathsf{Level}(\ell')$. (Alternately, think of each security level as determining a collection of store locations; each bound store location may be renamed only to a location within the same collection.) Determinacy is used in the proof of non-interference.

Lemma 2.2 (*Determinacy*)
If $M \to M_1$ and $M \to M_2$ then $M_1 = M_2$. If $S \to S_1$ and $S \to S_2$ then $S_1 = S_2$ (upto renaming of store locations).

Proof
by induction on the evaluation relations. By cases on $M \to M_1$ (or $S \to S_1$).

In each case, by the structure of M (resp., S), there is a single evaluation rule for $M \to M_2$ (resp., $S \to S_2$), then by IH. □

Since allocation extends the store, the following lemma shows that in any sequence of evaluation steps (of a not necessarily well-typed state), the store type only grows. We use this fact in the HSS Lemma.

Lemma 2.3 (*Store Size*)
If $(H, \Sigma, E) \to^* (H', \Sigma', E')$ then $\Sigma' \supseteq \Sigma$.

Proof
Suffices to show for one step: if $(H, \Sigma, E) \to (H', \Sigma', E')$ then $\Sigma' \supseteq \Sigma$. The multi-step result follows because $\supseteq$ is reflexive and transitive. We proceed by induction on the evaluation derivation $(H, \Sigma, E) \to (H', \Sigma', E')$, by cases on the last rule in the derivation. □

3 Upcalls

Although the approach discussed so far is secure, it falls short of a practical language. There is no way to include a computation that reads from the high-security store in a larger low security computation. In any program with a high security read, the read level of the entire program is pushed up. However, many programs that contain *upcalls* to high security computations followed by low security code are secure.

Consider the program $\mathsf{let}\ z = P\ \mathsf{in}\ E$ where $P \div_{(\top,\top)} 1$ and E has operation level $(\bot, \bot)$. P does not leak information because 1 carries no useful information, and P's writes are above E's reading level. Thus we would like to give the entire program the operation level $(\bot, \bot)$. However the type system we have presented so far would instead promote the operation level of E and the entire program to $(\top, \top)$.

In order to have a logic of information flow, we must offer an account of upcalls. Indeed, the power to perform high security computations interspersed in a larger low-security computation is the *sine qua non* of useful secure programming languages. We offer a detailed analysis of two cases where upcalls do not violate our intuitive notion of security. From these examples, we develop a general principle for treating upcalls – our notion of *informativeness* – discussed in section 3.3. We take up the question of non-interference in section 5.

3.1 An example with unit

Let E be some expression with type A and operation level (r, w) (recall that this implies that $r \sqsubseteq w$). In general, E may read values from store locations with security

level below r, write values to store locations with security level at least w, and return some value of type A.

Suppose that $A = 1$. In that case, no matter what E does, if it terminates, it must return $*$. *The return value is not informative.*[1] Any other computation F that may gain information through the execution of E must be able to read store locations at security level at least w. But since $r \sqsubseteq w$, F could just directly read any store locations that E reads. On the other hand, any computation with operation level (r', w') where $w \not\sqsubseteq r'$ can neither observe E's effects nor gain any information from its (uninformative) return value.

As a result, in either case, we can say that E has an effective read level of $\bot$ just as if it had no reads:

$$\frac{\Sigma; \Gamma \vdash E \div_{(r,w)} 1}{\Sigma; \Gamma \vdash E \div_{(\bot,w)} 1} \ (*)$$

Note that the read level now refers only to informative reads, not all reads.

The new rule allows us to have some high-security computations prior to low security ones. Suppose $\Sigma; \cdot \vdash E \div_{(\top,\top)} 1$, and $\Sigma; x : 1 \vdash F \div_{(\bot,\bot)} A$ for some A. That is, E is a high-security computation, and F is a low-security one. With the new rule, the upcall to E, followed by the low-security computation F, can be type checked using the new rule $(*)$, E has operation level $(\bot, \top)$, which can be weakened to $(\bot, \bot)$ by rule (34), and thus:

$$\frac{\dfrac{\begin{array}{c}\vdots \\ \Sigma; \cdot \vdash E \div_{(\bot,\bot)} 1\end{array}}{\Sigma; \cdot \vdash \mathsf{val}\ E : \bigcirc_{(\bot,\bot)} 1}\ (27) \qquad \Sigma; x : 1 \vdash F \div_{(\bot,\bot)} A}{\Sigma; \cdot \vdash \mathsf{let\ val}\ x = \mathsf{val}\ E\ \mathsf{in}\ F \div_{(\bot,\bot)} A}\ (30)$$

Note that the rule $(*)$ does not alter the write level of the expression (that is, the operation level in the conclusion is not $(\bot, \top)$). Such a rule would allow programs to leak information.

3.2 A more general example

Now consider an expression E with operation level (r, w), but this time, suppose that E has type $\mathsf{ref}_a B$ for some type B. Are there any situations where E may be given a different operation level?

Suppose that $r \sqsubseteq a$. In that case, any computation that may read the $\mathsf{ref}_a B$ is also able to read any store locations that E may read. Again, any computation can either do what E does itself, or it cannot gain information from E's return value.

On the other hand, consider the case where $r \not\sqsubseteq a$. The particular value of type $\mathsf{ref}_a B$ that E returns may carry information from store locations at security level r. For example, E may return one of two such store locations ℓ_1 or ℓ_2 from level a based on some boolean value V from a store location at security level r. In that

[1] We are dealing here with weak non-interference: the knowledge that E terminated at all is deemed not to carry any information.

$$\boxed{\vdash A \nearrow a}$$

$$\frac{}{\vdash A \nearrow \bot}\ (1) \qquad \frac{\vdash A \nearrow a \quad b \sqsubseteq a}{\vdash A \nearrow b}\ (10) \qquad \frac{\vdash A \nearrow a \quad \vdash A \nearrow b}{\vdash A \nearrow a \sqcup b}\ (11)$$

$$\frac{}{\vdash 1 \nearrow a}\ (2) \qquad \frac{\vdash B \nearrow a}{\vdash A \to B \nearrow a}\ (3) \qquad \frac{\vdash A \nearrow a}{\vdash \bigcirc_{(r,w)} A \nearrow w \sqcap a}\ (4)$$

$$\frac{}{\vdash \mathsf{ref}_a\, A \nearrow a}\ (5) \qquad \frac{\vdash A \nearrow a}{\vdash \mathsf{ref}_b\, A \nearrow a}\ (6) \qquad \frac{\vdash A \nearrow a}{\vdash \mathsf{refr}_b\, A \nearrow a}\ (7) \qquad \frac{}{\vdash \mathsf{refr}_b\, A \nearrow b}\ (8) \qquad \frac{}{\vdash \mathsf{refw}_a\, A \nearrow a}\ (9)$$

Fig. 4. Informativeness judgment.

case, a computation that reads at security level a may learn something about E's reads (at level r) by reading from E's return value. Since $r \not\sqsubseteq a$, this represents a violation of secure information flow.

So if E returns a $\mathsf{ref}_a\, B$, we can demote its reading level whenever $r \sqsubseteq a$, because any computation that wishes to make use of that return value would need a read level of at least r. In other words, a $\mathsf{ref}_a\, B$ is informative only to computations that may read at least at some security level (namely a) above r.

This observation suggests a new subsumption rule for expressions that alters the operation level:

$$\frac{\Sigma; \Gamma \vdash E \div_{(r,w)} A \quad \vdash A \nearrow r}{\Sigma; \Gamma \vdash E \div_{(\bot,w)} A}\ (35)$$

where the new *informativeness* judgment $\vdash A \nearrow r$ formalizes the idea that values of type A, if they are informative at all, are informative only at level r or above.[2]

In terms of this new judgment, our earlier observations are that $\vdash 1 \nearrow r$ for any r, and $\vdash \mathsf{ref}_a\, A \nearrow r$ whenever $r \sqsubseteq a$.

3.3 Informativeness

We now consider some properties of the new judgment $\vdash A \nearrow a$ (see Figure 4). Several structural rules (1,10,11) for the judgment are immediate. If A is informative at all, then it's informative only at $\bot$ or above. Also, if A is informative only at or above a and if $b \sqsubseteq a$, then A is informative only at or above b. That is, we may choose to discard some knowledge about when a type is informative. Finally, suppose A is informative only above a, and A is informative only above b. Then for any r if values of type A are informative to computations that read at r, we know that both $a \sqsubseteq r$ and $b \sqsubseteq r$. Therefore, for any such r, $a \sqcup b \sqsubseteq r$. So, A is informative only above $a \sqcup b$.

With the structural rules in place, we may consider each of the types in our language. We should keep in mind, that by adding rules to the judgment $\vdash A \nearrow a$ we increase the expressive power of the language by allowing more programs to be well-typed. It is always safe to add more restrictive rules in place of more liberal

[2] Informativeness is closely related to *protectedness* in DCC (Abadi *et al.*, 1999) and to the tampering levels of Honda & Yoshida (2002). We discuss the relationship in section 7.

ones. Below we take the most permissive rules that still maintain non-interference, although it is not clear in all cases that such flexibility is needed in practice.

A value of type bool is informative for any computation at all, since it may be trivially analyzed with a conditional. So aside from the structural axiom $\vdash A \nearrow \bot$, there should be no other rules for bool. We would give a similar account of other types that may be analyzed by branching (e.g. sum types $A + B$ or integers int).

Since functions are used by application, a value of type $A \to B$ is useful exactly when B is.

One straightforward rule (5) for references says that a store location is only informative if we can get at the value within it. There is an additional rule for references. Even if a computation can read from a store location of type $\mathsf{ref}_b\, A$ (i.e. its read level is above b), only if A is informative at its operation level, can $\mathsf{ref}_b\, A$ be informative.

Read-only store locations are useful only to computations that may read from them. Consequently, by an argument similar to the one for read-write store cells, they have analogous rules.

For write-only store cells $\mathsf{refw}_a\, A$, we have to consider aliasing. One way that a computation may learn whether two store locations are aliases is by writing a known value to one of them, and then reading out the value from the other. Because of subtyping, if a lower-security computation has a store location ℓ of type $\mathsf{refr}_a\, A$, a value of type $\mathsf{refw}_a\, A$ may be informative if the computation can read from (the seemingly unrelated) ℓ.

It is instructive to consider in detail the problem with write-only store locations $\mathsf{refw}_a\, A$. Suppose that instead of the rule (9), we had the following rule

$$\frac{}{\vdash \mathsf{refw}_a\, A \nearrow b} \ (\textit{incorrect})$$

That is, the same as the rule for unit: a value of type $\mathsf{refw}_a\, A$ is only informative above some security level b, for any b, i.e. not informative.

The following computation shows that with the incorrect rule, it is possible to leak high security information (whether the value of *secret*, a $\top$-security bool, is true) to a low security computation[3]:

$$\begin{array}{l}
\mathsf{let}\ x = \mathsf{ref}_\bot\,(\mathsf{false} : \mathsf{bool})\ \mathsf{in} \\
\mathsf{let}\ y = \mathsf{ref}_\bot\,(\mathsf{false} : \mathsf{bool})\ \mathsf{in} \\
\mathsf{let}\ z = (\mathsf{let}\ q = !\mathit{secret}\ \mathsf{in} \\
\qquad\qquad [\mathsf{if}\ q\ \mathsf{then}\ x\ \mathsf{else}\ y])\ \mathsf{in} \\
\mathsf{let}\ _ = z := \mathsf{true}\ \mathsf{in} \\
\mathsf{run}\ !x
\end{array}$$

The program lets z alias either x or y depending on the value of *secret*. The computation whose value is assigned to z may be subsumed to type $\mathsf{refw}_\bot\,\mathsf{bool}$, and by the incorrect rule, $\vdash \mathsf{refw}_\bot\,\mathsf{bool} \nearrow \top$, so the operation level of that computation can be dropped to $(\bot, \top)$ (and subsumed to $(\bot, \bot)$). Then by writing a known value

[3] Recall that $\mathsf{let}\ x = E\ \mathsf{in}\ F$ is syntactic sugar for $\mathsf{let\ val}\ x = \mathsf{val}\ E\ \mathsf{in}\ F$.

```
λc : ◯(⊤,⊤)bool.
 val let wref = ref⊤ (val [*] : ◯(⊥,⊤)1) in
     let w = [val (let b = run c in run (if b then val (let w′ = !wref in run w′) else val [*]))] in
     let _ = wref := w in
     run w
```

Fig. 5. Using rule (35), *untilFalse* has type $\bigcirc_{(\top,\top)}\mathsf{bool} \to \bigcirc_{(\bot,\top)}1$.

to z, whether we can observe a change in another alias of the same location is sufficient to learn about *secret*. We can give the entire computation the operation level $(\bot, \top)$ while it demonstrably returns the high-security value.

Finally, consider the type $\bigcirc_{(r,w)}A$. A value of this type is informative both to computations that may read at least security level w (that is, the level the suspended expression writes to), and to computations for which the type A is informative.

With informativeness in hand, many more useful terms become well-typed. Consider, for example, the term in Figure 5. The function *untilFalse* takes as argument a computation that reads and writes high before returning a boolean, and runs that computation repeatedly until it returns false. Recursion is accomplished using backpatching: a store location with a dummy value is allocated and is bound to *wref*, recursive calls in the body of the loop dereference *wref* and run the contents. The recursive knot is tied by overwriting the contents of *wref* with the real loop body w.

Interestingly, although *untilFalse* takes a high-security computation as an argument, our type system is able to give it the type $\bigcirc_{(\top,\top)}\mathsf{bool} \to \bigcirc_{(\bot,\top)}1$, that is its return type is a low-security computation. Intuitively, even if c is a high-security computation, *untilFalse c* does not leak any information to low-security since any information gained from c's return value is used only within the loop. To show that *untilFalse* is well-typed, observe that $\Gamma \vdash \mathsf{let}\ b = \mathsf{run}\ c\ \mathsf{in}\ \mathsf{run}\ (\ldots) \div_{(\top,\top)} 1$, and since $\vdash 1 \nearrow \top$, it can be given operation level $(\bot, \top)$. The rest of the typing derivation is straightforward.

4 Type safety

Our language enjoys the usual type safety property: well-typed computations do not become stuck. We may show type safety in the usual manner using Preservation and Progress lemmas.

Lemma 4.1 (*Preservation*)
If $\vdash S \div_o A$ and $S \to S'$ then $\vdash S' \div_o A$

Proof by induction on the evaluation relation. Proof is given in Appendix C.1.4.

Lemma 4.2 (*Progress*)
If $\vdash S \div_o A$ then either S is terminal, or there exists an S' such that $S \to S'$

Table 2. *Equivalence judgments*

Judgment	Meaning
$\Sigma_1;\Sigma_2;\Gamma \vdash M_1 \approx_\zeta M_2 : A$	Term Equivalence
$\Sigma_1;\Sigma_2;\Gamma \vdash E_1 \approx_\zeta E_2 \div_o A$	Expression Equivalence
$\vdash (H_1 : \Sigma_1) \approx^U_\zeta (H_2 : \Sigma_2)$	Store Equivalence
$\vdash S_1 \approx_\zeta S_2 \div_o A$	State Equivalence

Proof by induction on the typing derivation. Proof is given in Appendix C.1.4.

Theorem 4.3 (*Type Safety*)
If $\vdash S$ and $S \rightarrow^* S'$ then S' is not stuck.

Proof
By induction on the number of evaluation steps. If S takes zero steps, then by Progress, it is not stuck. If S takes $n+1$ steps, then by Preservation it takes a step to some well-typed state, and so by the induction hypothesis, S' is not stuck. □

5 Non-interference

Fix a security level ζ. We say that a security level is low if it is below ζ, and high otherwise. Informally, non-interference says that computations that have a low read level do not depend on values in high security store locations.

The proof is structured similarly to that of others (Zdancewic & Myers, 2002; Zdancewic & Myers, 2001b), and other proofs of non-intereference based on relating the operational behavior of pairs of computations, such as Pottier & Simonet (2003). However, by taking advantage of our informativeness judgment (see below), we can give a concise definition of the relation between computation states.

Operationally, the low security sub-computations of a program should behave identically irrespective of the values in the high security store locations. On the other hand, high security sub-computations may behave differently based on values in high security store locations. However once a high security sub-computation completes, the low security behavior should again be identical modulo the parts of the computation state that are "out of view" of the low security part of the program.

Formally, we define an equivalence property of computation states such that two states are equivalent whenever they agree on the "in view" parts of the computation state. Then, in the style of a confluence proof modulo an equivalence relation (Huet, 1980), we show that this property is preserved under evaluation.

5.1 Equivalence relation

We axiomatize the desired property as a collection of equivalence judgments (on states, stores, terms and expressions) that are summarized in Table 2.

Stores and States Certainly values in high security store locations are out of view. Less obviously, some values in the low security locations are out of view as well: if a low security store location appears only out of view, its value is also out of view. We parametrize the store equivalence judgment by a set U of in-view store locations. Two (well-typed) stores are equivalent only if their in view values are equivalent:

$$\frac{\begin{array}{l}\vdash H_1 : \Sigma_1 \quad \vdash H_2 : \Sigma_2 \\ \Sigma_1 \upharpoonright U = \Sigma_2 \upharpoonright U\end{array} \qquad \Sigma_1; \Sigma_2; \cdot \vdash H_1(\ell) \approx_\zeta H_2(\ell) : \Sigma_1(\ell) \text{ for } \ell \in U}{\vdash (H_1 : \Sigma_1) \approx_\zeta^U (H_2 : \Sigma_2)} \quad (58)$$

Where the notation $\Sigma \upharpoonright X$ is the restriction of Σ to just the locations in the set X.

For a pair of computation states, only low security locations that are common to both computations are in-view. Since allocation does not leak information, it is possible for two programs to allocate different low security locations while executing high security sub-computations. However such locations are out of view for the low security sub-computation.

Pairs of computation states are equivalent if their stores are equivalent on the in-view locations, and if they have equivalent expressions:

$$\frac{\vdash (H_1 : \Sigma_1) \approx_\zeta^{\mathrm{dom}(H_1) \cap \mathrm{dom}(H_2) \cap \downarrow(\zeta)} (H_2 : \Sigma_2) \quad \Sigma_1; \Sigma_2; \cdot \vdash E_1 \approx_\zeta E_2 \div_o A}{\vdash (H_1, \Sigma_1, E_1) \approx_\zeta (H_2, \Sigma_2, E_2) \div_o A} \quad (59)$$

Where $\downarrow(\zeta) = \{\ell \mid \mathsf{Level}(\ell) \sqsubseteq \zeta\}$ is the set of all low security locations.

Terms and Expressions High security sub-computations of a program may return different values to the low security sub-computations. However, by the upcall rule (35), the type of those values must be informative only at high security.

Values of a type that is informative only at high security are out of view. As a result, any two values of such a type are equivalent since two such values vacuously agree on their in view parts:

$$\frac{\Sigma_1; \Gamma \vdash V_1 : A \quad \Sigma_2; \Gamma \vdash V_2 : A \quad \vdash A \nearrow a \quad a \not\sqsubseteq \zeta}{\Sigma_1; \Sigma_2; \Gamma \vdash V_1 \approx_\zeta V_2 : A} \quad (39)$$

The remaining rules for term and expression equivalence are congruence rules that merely require corresponding sub-terms or sub-expressions to be equivalent. They are listed in Appendix A.3. Some useful structural properties (transitivity, inversion, functionality, etc) of the equivalence judgment are proved in Appendix C.2.

5.2 Hexagon lemmas

Non-interference will follow as a consequence of a pair of Hexagon Lemmas: one for terms and one for expressions. We show that by starting with some two related terms (or expressions) that both take a step, we can find zero or more steps that each of them could take so that we get back to related states (respectively, expression).

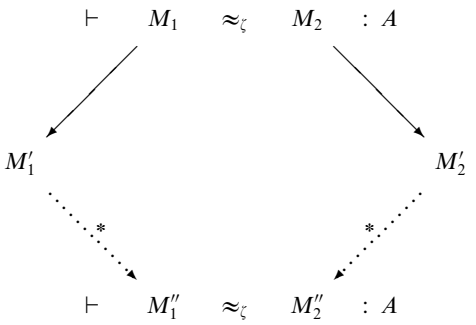

Fig. 6. Informal statement of the Term Hexagon Lemma.

The lemma for terms is summarized in Figure 6, the name "Hexagon Lemma" is motivated by the shape of this diagram.

Lemma 5.1 (*Term Hexagon Lemma*)
For all ζ, if $\Sigma_1;\Sigma_2;\cdot \vdash M_1 \approx_\zeta M_2 : A$ and $M_1 \to M_1'$ and $M_2 \to M_2'$ and $M_1' \downarrow$ and $M_2' \downarrow$, then there exist M_1'', M_2'' such that $M_1' \to^* M_1''$, $M_2' \to^* M_2''$, $\Sigma_1;\Sigma_2;\cdot \vdash M_1'' \approx_\zeta M_2'' : A$

The proof is by induction on the given derivation. Most cases are vacuous. In the cases of function application and if-then-else, proceed by subcases on $M_1 \to M_1'$. The full proof is given in Appendix C.3.

Roughly speaking, the proof of the Hexagon Lemma for expressions is divided into two cases depending on whether the sub-expressions of the current pair of states depend on in-view ($\sqsubseteq \zeta$) or out-of-view ($\not\sqsubseteq \zeta$) locations. In the former case, the two states execute in lock-step and after each computation step we can show the resulting states are equivalent. In the latter case, each computation state may execture arbitrarily many high security steps before continuing on with low-security computation that is in view of the observer. The following High Security Step lemma shows that starting with two equivalent stores and executing arbitrary high-security expressions, the resulting stores are still equivalent for the low-security observer.

One complication in this lemma is that evaluation of two distinct computation states S_1, S_2 may inadvertently allocate the same store location ℓ for distinct purposes. However we will show that for each such ℓ, we may choose an element of the α-equivalence class of S_1 or S_2 such that all such accidental sharing is eliminated.

Lemma 5.2 (*High Security Step (HSS)*)
Given (H_1, Σ_1, E_1) and (H_2, Σ_2, E_2) such that

- $\vdash (H_1 : \Sigma_1) \approx_\zeta^U (H_2 : \Sigma_2)$ where $U = \mathrm{dom}(\Sigma_1) \cap \mathrm{dom}(\Sigma_2) \cap \downarrow(\zeta)$,
- $\Sigma_i;\cdot \vdash E_i \div_{o_i} C_i$ for some $o_i = (r_i, w_i), C_i$ with $w_i \not\sqsubseteq \zeta$ for $i = 1, 2$,

and if $(H_1, \Sigma_1, E_1) \to^* (H_1', \Sigma_1', E_1')$ and $(H_2, \Sigma_2, E_2) \to^* (H_2', \Sigma_2', E_2')$ then

- $\vdash (H_1' : \Sigma_1') \approx_\zeta^U (H_2' : \Sigma_2')$
- and moreover $U = \mathrm{dom}(\Sigma_1') \cap \mathrm{dom}(\Sigma_2') \cap \downarrow(\zeta)$

Proof

1. By Regularity of Equivalence, $\vdash (H_i : \Sigma_i)$ for $i = 1, 2$
2. By Lemma C.21, for $i = 1, 2$:
   - $\vdash (H_i : \Sigma_i) \approx_\zeta^{U_i} (H_i' : \Sigma_i')$ where $U_i = \mathrm{dom}(\Sigma_i) \cap {\downarrow}(\zeta)$
3. By Regularity, $\vdash H_i' : \Sigma_i'$ for $i = 1, 2$
4. Note also that $U \subseteq U_i$ for $i = 1, 2$
5. Consider an arbitrary $\ell \in U$
   (a) Since $\vdash (H_1 : \Sigma_1) \approx_\zeta^U (H_2 : \Sigma_2)$, $\Sigma_1; \Sigma_2; \cdot \vdash H_1(\ell) \approx_\zeta H_2(\ell) : A$,
   (b) Evidently, also $\ell \in U_i$
   (c) And since, for $i = 1, 2$, $\vdash (H_i : \Sigma_i) \approx_\zeta^{U_i} (H_i' : \Sigma_i')$, we have $\Sigma_i; \Sigma_i'; \cdot \vdash H_i(\ell) \approx_\zeta H_i'(\ell) : A$.
   (d) Therefore, by symmetry and transitivity, $\Sigma_1'; \Sigma_2'; \cdot \vdash H_1'(\ell) \approx_\zeta H_2'(\ell) : A$
6. So, by rule (58), $\vdash (H_1' : \Sigma_1') \approx_\zeta^U (H_2' : \Sigma_2')$
7. Now consider a new set $U' = \mathrm{dom}(\Sigma_1') \cap \mathrm{dom}(\Sigma_2') \cap {\downarrow}(\zeta)$
8. Since $\Sigma_i' \supseteq \Sigma_i$, so $U' \supseteq U$
9. Suppose $\ell \in U' \setminus U$
   (a) Since $\ell \in U'$, $\ell \in \mathrm{dom}(\Sigma_i')$ for $i = 1, 2$
   (b) Since $\ell \notin U$, then $\ell \notin \mathrm{dom}(\Sigma_i)$ for at least one of $i = 1$ or $i = 2$
   (c) Suppose $\ell \notin \mathrm{dom}(\Sigma_1)$ (the other case is similar)
   (d) Choose a fresh store location $\ell' \notin \mathrm{dom}(\Sigma_1') \cup \mathrm{dom}(\Sigma_2')$ with $\mathsf{Level}(\ell') = \mathsf{Level}(\ell)$, and systematically rename ℓ with ℓ' in (H_1', Σ_1', E_1').
   (e) Evidently we have an element of the α-equivalence class of (H_1', Σ_1', E_1') where $\ell \notin \mathrm{dom}(H_1')$
10. So $U' = U$, and the conclusion of the lemma follows.

□

We may now show a hexagon lemma for expressions.

Lemma 5.3 (*Hexagon Lemma*)
For all ζ, if $o = (r, w)$ with $r \sqsubseteq \zeta$, and if

- $\vdash S_1 \approx_\zeta S_2 \div_o C$
- $S_1 \to S_1'$, $S_2 \to S_2'$
- $S_1' \downarrow$, $S_2' \downarrow$

then there exist S_1'', S_2'' such that

- $S_1' \to^* S_1''$, $S_2' \to^* S_2''$
- $\cdot \vdash S_1'' \approx_\zeta S_2'' \div_o C$

Proof
By Inversion on $\vdash S_1 \approx_\zeta S_2 \div_o C$, we have

- $S_1 = (H_1, \Sigma_1, E_1)$, $S_2 = (H_2, \Sigma_2, E_2)$
- $\vdash (H_1 : \Sigma_1) \approx_\zeta^U (H_2 : \Sigma_2)$ where $U = \mathrm{dom}(\Sigma_1) \cap \mathrm{dom}(\Sigma_2) \cap {\downarrow}(\zeta)$
- $\Sigma_1; \Sigma_2; \cdot \vdash E_1 \approx_\zeta E_2 \div_o C$

Now we proceeed, by induction on the derivation of $\Sigma_1; \Sigma_2; \cdot \vdash E_1 \approx_\zeta E_2 \div_o C$, by cases on the last rule in the derivation. In each case we exhibit the appropriate $S_i'' = (H_1'', \Sigma_1'', E_1''), S_2'' = (H_2'', \Sigma_2'', E_2'')$. We show several representative cases below.

- Case

$$\frac{\Sigma_1;\Sigma_2;\Gamma \vdash E_1 \approx_\zeta E_2 \div_{(r',w)} C \quad \vdash C \nearrow r'}{\Sigma_1;\Sigma_2;\Gamma \vdash E_1 \approx_\zeta E_2 \div_{(\bot,w)} C} \quad (51)$$

By pattern matching, $r = \bot$
Consider two subcases: either $r' \sqsubseteq \zeta$ or $r' \not\sqsubseteq \zeta$. The former case follows by the induction hypothesis. In the latter case, we appeal to the HSS lemma:

1. Since $r' \sqsubseteq w$, then $w \not\sqsubseteq \zeta$
2. Since $S_i' \downarrow$, $(H_i, \Sigma_i, E_i) \rightarrow^+ (H_i'', \Sigma_i'', [V_i])$ for some $S_i'' = (H_i'', \Sigma_i'', [V_i])$ for $i = 1, 2$
3. Therefore we can apply HSS to get
   — $\vdash (H_1'' : \Sigma_1'') \approx_\zeta^U (H_2'' : \Sigma_2'')$
   — $U = \mathrm{dom}(H_1'') \cap \mathrm{dom}(H_2'') \cap \downarrow(\zeta)$
4. By repeatedly applying Preservation, $\Sigma_i'';\cdot \vdash [V_i] \div_{(r',w)} C$ for $i = 1, 2$
5. And by various typing rules, $\Sigma_1'';\Sigma_2'';\cdot \vdash [V_1] \approx_\zeta [V_2] \div_o C$

- Case

$$\frac{\Sigma_1;\Sigma_2;\cdot \vdash M_1 \approx_\zeta M_2 : A}{\Sigma_1;\Sigma_2;\cdot \vdash \mathsf{ref}_a\,(M_1 : A) \approx_\zeta \mathsf{ref}_a\,(M_2 : A) \div_{(\bot,\top)} \mathsf{ref}_a\,A} \quad (55)$$

By pattern matching, $E_i = \mathsf{ref}_a\,(M_i : A)$, $o = (\bot, \top)$, $C = \mathsf{ref}_a\,A$
There are two possible evaluation rules for $(H_1, \Sigma_1, E_1) \rightarrow (H_1', \Sigma_1', E_1')$

— Subcase REF1 follows eventually from the Term Hexagon Lemma.
— Subcase REF: M_1 value, $H_1' = H_1\{\ell_1 \mapsto M_1\}$, $\Sigma_1' = \Sigma_1\{\ell_1 : A\}$, $E_1' = [\ell_1]$, where $\ell_1 \notin \mathrm{dom}(H_1)$, $\mathsf{Level}(\ell_1) = a$

1. By Equivalent Values (Lemma C.19), M_2 is a value since M_1 is
2. Only REF rule is applicable to $(H_2, \Sigma_2, E_2) \rightarrow (H_2', \Sigma_2', E_2')$: $H_2' = H_2\{\ell_2 \mapsto M_2\}$, $\Sigma_2' = \Sigma_2\{\ell_2 : A\}$, $E_2' = [\ell_2]$, where $\ell_2 \notin \mathrm{dom}(H_2)$, $\mathsf{Level}(\ell_2) = a$
3. Consider two subcases now, either $a \sqsubseteq \zeta$ or $a \not\sqsubseteq \zeta$. In the former case we want both states to allocate the same fresh location (that will be in-view to the observer), in the latter case we want the locations to be distinct (and thus out of view):
   - Subcase $a \sqsubseteq \zeta$
     (a) Since in both S_1' and S_2', ℓ_1 and ℓ_2 are freshly allocated, we may α-vary S_1', S_2' such that $\ell_1 = \ell_2 = \ell$ for an appropriate ℓ
     (b) Then $\mathsf{Level}(\ell) = a$, $\ell \notin \mathrm{dom}(H_1) \cup \mathrm{dom}(H_2)$
     (c) Let $S_i'' = (H_i\{\ell \mapsto M_i\}, \Sigma_i\{\ell : A\}, [\ell])$ for $i = 1, 2$
     (d) The result follows since the freshly allocated location is (by construction) in the set $U'' = U \cup \{\ell\}$ of common locations between S_1'' and S_2'', and it contains equivalent values.
   - Subcase $a \not\sqsubseteq \zeta$
     In this case the newly allocated locations ℓ_1, ℓ_2 are not in the common set of S_1'' and S_2'' since they have high security levels.

Furthermore $\Sigma_1\{\ell_1 : A\}; \Sigma_2\{\ell_2 : A\}; \cdot \vdash \ell_1 \approx_\zeta \ell_2 : \mathsf{ref}_a A$, since $\vdash \mathsf{ref}_a A \nearrow a$ and $a \not\sqsubseteq \zeta$. The result follows.

- Case

$$\frac{\Sigma_1; \Sigma_2; \cdot \vdash M_1 \approx_\zeta M_2 : \mathsf{refr}_a C}{\Sigma_1; \Sigma_2; \cdot \vdash !M_1 \approx_\zeta !M_2 \div_{(a,\top)} C} \quad (56)$$

By pattern matching, $E_i = !M_i$, $o = (r, w) = (a, \top)$. Recall that $a = r \sqsubseteq \zeta$
There are two applicable rules for $(H_1, \Sigma_1, E_1) \to (H_1', \Sigma_1', E_1')$

— Subcase BANG1 follows by the Term Hexagon Lemma
— Subcase BANG: $M_1 = \ell_1$, $H_1' = H_1$, $\Sigma_1' = \Sigma_1$, $E_1' = [H_1(\ell_1)]$

1. By Equivalent Values (Lemma C.19), M_2 is a value since M_1 is.
2. The single applicable evaluation rule for $(H_2, \Sigma_2, E_2) \to (H_2', \Sigma_2', E_2')$ is BANG: $M_2 = \ell_2$, $H_2' = H_2$, $\Sigma_2' = \Sigma_2$, $E_2' = [H_2(\ell_2)]$
3. Let $S_i'' = (H_i, \Sigma_i, [H_i(\ell_i)])$ for $i = 1, 2$
4. So it only remains to show that $\Sigma_1; \Sigma_2; \cdot \vdash H_i(\ell_1) \approx_\zeta H_2(\ell_2) : C$
5. By Equivalent Term Inversion on $\Sigma_1; \Sigma_2; \cdot \vdash \ell_1 \approx_\zeta \ell_2 : \mathsf{refr}_a C$, there are two possibilities:
   - Either $\Sigma_i; \cdot \vdash \ell_i : B$ and $\vdash B \leqslant \mathsf{refr}_a C$ and $\vdash B \nearrow b$ and $b \not\sqsubseteq \zeta$
     It follows by a series of inversions that B is either $\mathsf{refr}_{b'} B'$ or $\mathsf{ref}_{b'} B'$ and in either case $\vdash B' \nearrow c$ for some $c \not\sqsubseteq \zeta$. That is, the computations are dereferencing locations whose contents are not informative to a ζ-observer. The result follows.
   - Or $\ell_1 = \ell_2 = \ell$ where $\mathsf{Level}(\ell) = b \sqsubseteq \zeta$,
     and $\vdash \mathsf{ref}_b \Sigma_1(\ell) \leqslant \mathsf{refr}_a C$ and $\Sigma_1(\ell) = \Sigma_2(\ell)$. This case follows since ℓ is in the common set of Σ_1, Σ_2 and since the stores are equivalent.

- Case

$$\frac{\Sigma_1; \Sigma_2; \cdot \vdash M_1 \approx_\zeta M_2 : \mathsf{refw}_a A \quad \Sigma_1; \Sigma_2; \cdot \vdash N_1 \approx_\zeta N_2 : A}{\Sigma_1; \Sigma_2; \cdot \vdash M_1 := N_1 \approx_\zeta M_2 := N_2 \div_{(\bot,a)} 1} \quad (57)$$

By pattern matching, $E_i = M_i := N_i$, $o = (r, w) = (\bot, a)$, $C = 1$
There are three applicable rules for $(H_1, \Sigma_1, E_1) \to (H_1', \Sigma_1', E_1')$. If the rule was ASSN1 or ASSN2, the result follows from the Term Hexagon Lemma.
Otherwise, the rule was ASSN, and we have: $M_1 = \ell_1$, N_1 value, $H_1' = H_1\{\ell_1 \mapsto N_1\}$, $\Sigma_1' = \Sigma_1$, $E_1' = [*]$

1. By Equivalent Values (Lemma C.19), M_2, N_2 are values since M_1, N_1 are.
2. The only applicable evaluation rule for $(H_2, \Sigma_2, E_2) \to (H_2', \Sigma_2', E_2')$ is ASSN, and we have: $M_2 = \ell_2$, $H_2' = H_2\{\ell_2 \mapsto N_2\}$, $\Sigma_2' = \Sigma_2$, $E_2' = [*]$
3. Let $S_i'' = (H_i\{\ell_i \mapsto N_i\}, \Sigma_i, [*])$. It suffices to show that the updated stores are still equivalent.
4. By Equivalent Term Inversion on $\Sigma_1; \Sigma_2; \cdot \vdash \ell_1 \approx_\zeta \ell_2 : \mathsf{refw}_a A$, there are two possibilities:

   — Either $\Sigma_i; \cdot \vdash \ell_i : B$ and $\vdash B \leqslant \mathsf{refw}_a A$, $\vdash B \nearrow b$ and $b \not\sqsubseteq \zeta$

By Subtyping Inversion, either $B = \mathsf{refw}_{b'}\, B'$ or $B = \mathsf{ref}_{b'}\, B'$ and in either case $\vdash A \leqslant B'$ and $a \sqsubseteq b'$

- If $B = \mathsf{refw}_{b'}\, B'$, then it eventually follows from inversions that $\mathsf{Level}(\ell_i) \not\sqsubseteq \zeta$, and so the ℓ_i are not in the common set U of locations, and the result follows.
- If $B = \mathsf{ref}_{b'}\, B'$
  (a) By Subtyping Inversion, $B' = \Sigma_i(\ell_i)$ and $b' = \mathsf{Level}(\ell_i)$ for $i = 1, 2$
  (b) By Informativeness Inversion, $b \sqsubseteq b' \sqcup c$ and $\vdash B' \nearrow c$ for some c
  (c) Since $b \not\sqsubseteq \zeta$, either $b' \not\sqsubseteq \zeta$ or $c \not\sqsubseteq \zeta$
  (d) If $b' \not\sqsubseteq \zeta$, we can use the same argument as the previous subcase: $B = \mathsf{refw}_{b'}\, B'$.
  (e) So instead suppose $b' \sqsubseteq \zeta$; it must be the case that $c \not\sqsubseteq \zeta$.
  (f) Consider ℓ_1 (the argument for ℓ_2 is symmetric)
  (g) Evidently $\mathsf{Level}(\ell_1) = b' \sqsubseteq \zeta$, so suppose $\ell_1 \in U$ (if not, same argument as previous subcase)
  (h) If $\ell_1 = \ell_2$ then the situation is the same as the next subcase ($\ell_1 = \ell_2 = \ell, \ldots$) below; so suppose ℓ_1 differs from ℓ_2
  (i) So $\ell_1 \in \mathrm{dom}(\Sigma_2) = \mathrm{dom}(H_2)$
  (j) By heap typing inversion, $\Sigma_2; \cdot \vdash H_2(\ell_1) : \Sigma_2(\ell_1)$
  (k) Since $\ell_1 \in U$, $\Sigma_2(\ell_1) = \Sigma_1(\ell_1) = B'$
  (l) By rule (39), $\Sigma_1; \Sigma_2; \cdot \vdash N_1 \approx_\zeta H_2(\ell_1) : \Sigma_1(\ell_1)$
  (m) Therefore for all $\ell \in U$, $\Sigma_1''; \Sigma_2''; \cdot \vdash H_1''(\ell) \approx_\zeta H_2''(\ell) : \Sigma_1''(\ell)$
  (n) So by rule (58), $\vdash (H_1'' : \Sigma_1'') \approx_\zeta^U (H_2'' : \Sigma_2'')$

— Or $\ell_1 = \ell_2 = \ell$ and $\mathsf{Level}(\ell) \sqsubseteq \zeta$ and $\Sigma_1(\ell) = \Sigma_2(\ell)$, and $\vdash \mathsf{ref}_{\mathsf{Level}(\ell)}\, \Sigma_1(\ell) \leqslant \mathsf{refw}_a\, A$
We can show that ℓ is in the common set U of locations; the result follows by a straightforward derivation.

□

5.3 Non-interference theorem

By repeatedly applying the Hexagon Lemma, we can prove a non-interference result. We show that starting with some initial store H (well-typed with store type Σ) and an expression to execute E with a free variable x, if we plug in different values V_1, V_2 for x, then provided that the in-view parts of V_1, V_2 are equivalent, we expect that if the resulting programs $(H, \Sigma, E[V_1/x])$, $(H, \Sigma, E[V_2/x])$ run to termination, the resulting terminal states will be equivalent on their in view parts.

Theorem 5.4 (*Non-interference*)
If $\vdash H : \Sigma$ and $\Sigma; x : A \vdash E \div_{(r,w)} B$ and if $\Sigma; \Sigma; \cdot \vdash V_1 \approx_r V_2 : A$ then if $(H, \Sigma, E[V_1/x]) \to^* S_1$ and $(H, \Sigma, E[V_2/x]) \to^* S_2$ and both S_1, S_2 are terminal, then $\vdash S_1 \approx_r S_2 \div_{(r,w)} B$

Proof
By reflexivity and Functionality (see Appendix C.2), we can show that

$$\Sigma;\Sigma;\cdot \vdash E[V_1/x] \approx_r E[V_2/x] \div_{(r,w)} B$$

By repeated application of the Hexagon Lemma, the two computations evaluate to equivalent terminal states. Since the operational semantics are deterministic, those terminal states are S_1 and S_2, respectively. □

6 Encoding value-oriented secure languages

Our account differs substantially from prior secure programming languages where each value has a security level. In such languages, terms are classified by security types: pairs of an ordinary type and a security level. The type system ensures that each term is assigned a security level at least as high as the security level of the terms contributing to it. In our account only the store provides security. A natural question is whether we sacrifice expressive power in comparison to value-oriented secure languages.

We will show that our language is at least as expressive by showing how to embed several value-oriented secure languages in our account. The embeddings are not only type correct, but also preserve security properties of the source languages.

When a computation analyzes a value of a datatype by cases, each arm – by virtue of control flow – gains information about the subject of the case expression. In a purely functional setting, that additional information may only be used to compute the return value of the expression. Thus it suffices to require the return type of each arm (and thus the entire case expression) to be at least as secure as the case subject.

On the other hand, in an imperative setting, information gained via control-flow may leave an expression non-locally (e.g. via a write to the store). As a result, it becomes necessary to track such *implicit flows* of information. Secure imperative languages use a so-called *program counter security level*, pc, as a lower bound on the information that a computation may gain via control flow. Consequently, the results and effects of each expression must be at least as secure as any information gained via control flow.

In contrast to value-oriented secure programming languages, in our account we expect that case analysis is at the term level, and thus the arms of the case term do not have side-effects. We show that our approach is at least as expressive as imperative value-oriented secure languages.

We consider the language $\lambda_{\mathrm{SEC}}^{\mathrm{REF}}$ (summarized in Figure 7) of Zdancewic (2002). In addition to unit and boolean types, it has function types that are annotated with a lower bound on the write effects of the function body, and store locations. The base values of $\lambda_{\mathrm{SEC}}^{\mathrm{REF}}$ are annotated with a security level inside expressions.

The typing rules for $\lambda_{\mathrm{SEC}}^{\mathrm{REF}}$ are given by a pair of mutually recursive judgments for base values and expressions, given in Figures 8 and 9.

The following key property is maintained by the $\lambda_{\mathrm{SEC}}^{\mathrm{REF}}$ typing judgment. Intuitively, it captures the idea that the value of an expression is at least as secure as the information that the expression gains via implicit information flow.

$$
\begin{array}{rcll}
t & \in & \textit{types} & ::= \quad 1 \mid \mathsf{bool} \mid s_1 \xrightarrow{\mathsf{pc}} s_2 \mid \mathsf{ref}\ s \\
s & \in & \textit{security types} & ::= \quad (t, a) \\
bv & \in & \textit{base values} & ::= \quad * \mid \mathsf{true} \mid \mathsf{false} \mid \ell \mid \lambda[\mathsf{pc}]x : s.e \\
e & \in & \textit{expressions} & ::= \quad x \mid bv_a \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid e_1 e_2 \mid \mathsf{ref}\ (e : s)) \mid\ !e \mid e := e'
\end{array}
$$

Fig. 7. $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ syntax.

$$\boxed{\Sigma; \Gamma \vdash bv : t}$$

$$\frac{}{\Sigma; \Gamma \vdash * : 1} \quad \frac{}{\Sigma; \Gamma \vdash \mathsf{true} : \mathsf{bool}} \quad \frac{}{\Sigma; \Gamma \vdash \mathsf{false} : \mathsf{bool}} \quad \frac{}{\Sigma; \Gamma \vdash \ell : \Sigma(\ell)}$$

$$\frac{\Sigma; \Gamma, x : s[\mathsf{pc}] \vdash e : s'}{\Sigma; \Gamma \vdash \lambda[\mathsf{pc}]x : s.e \ : s \xrightarrow{\mathsf{pc}} s'} \quad \frac{\Sigma; \Gamma \vdash bv : t' \quad \vdash t' \leqslant t}{\Sigma; \Gamma \vdash bv : t}$$

Fig. 8. $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ base value typing.

$$\boxed{\Sigma; \Gamma[\mathsf{pc}] \vdash e : s}$$

$$\frac{}{\Sigma; \Gamma, x : s[\mathsf{pc}] \vdash x : s \sqcup \mathsf{pc}} \quad \frac{\Sigma; \Gamma \vdash bv : t}{\Sigma; \Gamma[\mathsf{pc}] \vdash bv_a : (t, a \sqcup \mathsf{pc})} \quad \frac{\Sigma; \Gamma[\mathsf{pc}] \vdash e : s' \quad \vdash s' \leqslant s}{\Sigma; \Gamma[\mathsf{pc}] \vdash e : s}$$

$$\frac{\Sigma; \Gamma[\mathsf{pc}] \vdash e_1 : (\mathsf{bool}, a) \quad \begin{array}{l}\Sigma; \Gamma[\mathsf{pc} \sqcup a] \vdash e_2 : s \\ \Sigma; \Gamma[\mathsf{pc} \sqcup a] \vdash e_3 : s\end{array}}{\Sigma; \Gamma[\mathsf{pc}] \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : s} \quad \frac{\Sigma; \Gamma[\mathsf{pc}] \vdash e_1 : (s' \xrightarrow{\mathsf{pc}'} s, a) \quad \begin{array}{l}\Sigma; \Gamma[\mathsf{pc}] \vdash e_2 : s' \\ \mathsf{pc} \sqcup a \sqsubseteq \mathsf{pc}'\end{array}}{\Sigma; \Gamma[\mathsf{pc}] \vdash e_1 e_2 : s \sqcup a}$$

$$\frac{\Sigma; \Gamma[\mathsf{pc}] \vdash e : s}{\Sigma; \Gamma[\mathsf{pc}] \vdash (\mathsf{ref}\ (e : s)) : (\mathsf{ref}\ s, \mathsf{pc})} \quad \frac{\Sigma; \Gamma[\mathsf{pc}] \vdash e : (\mathsf{ref}\ s, a)}{\Sigma; \Gamma[\mathsf{pc}] \vdash\ !e : s \sqcup a}$$

$$\frac{\Sigma; \Gamma[\mathsf{pc}] \vdash e_1 : (\mathsf{ref}\ (t, b), a) \quad \Sigma; \Gamma[\mathsf{pc}] \vdash e_2 : (t, b) \quad a \sqsubseteq b}{\Sigma; \Gamma[\mathsf{pc}] \vdash e_1 := e_2 : (1, \mathsf{pc})}$$

Fig. 9. $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ expression typing.

Lemma 6.1
If $\Sigma; \Gamma[\mathsf{pc}] \vdash e : (t, a)$ then $\mathsf{pc} \sqsubseteq a$.

The proof of this fact appears as Lemma 3.2.1 in Zdancewic (2002).

Encoding To emulate the sealing behavior of value-oriented languages in our store-oriented discipline, we embed source-language values of security type $s = (t, a)$ into read-only refs in our language $\overline{s} = \mathsf{refr}_a\, \overline{t}$.

A slight complication arises in the translation of ref types $\mathsf{ref}\ s$ since our language associates a security level with ref cells, but $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ does not. In value-oriented security languages, the contents of ref cells have a security level, however. So we use the security level a of the contents t as the security level of the ref cell itself in our translation: $\overline{\mathsf{ref}\ (t, a)} = \mathsf{ref}_a\, \overline{(t, a)}$.

$$\boxed{\Sigma;\Gamma \vdash bv : t \Rightarrow M}$$

$$\frac{}{\Sigma;\Gamma \vdash * : 1 \Rightarrow *} \quad \frac{}{\Sigma;\Gamma \vdash \mathsf{true} : \mathsf{bool} \Rightarrow \mathsf{true}} \quad \frac{}{\Sigma;\Gamma \vdash \mathsf{false} : \mathsf{bool} \Rightarrow \mathsf{false}}$$

$$\frac{}{\Sigma;\Gamma \vdash \ell : \Sigma(\ell) \Rightarrow \ell} \quad \frac{\Sigma;\Gamma, x : s_1[\mathsf{pc}] \vdash e : s_2 \Rightarrow E}{\Sigma;\Gamma \vdash \lambda[\mathsf{pc}]x : s_1.e : s_1 \xrightarrow{\mathsf{pc}} s_2 \Rightarrow \lambda x : \overline{s_1}.\mathsf{val}\ E}$$

$$\frac{\Sigma;\Gamma \vdash bv : t' \Rightarrow E \quad \vdash t' \leqslant t}{\Sigma;\Gamma \vdash bv : t \Rightarrow E}$$

Fig. 10. $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ base value encoding.

$$\boxed{\Sigma;\Gamma[\mathsf{pc}] \vdash e : s \Rightarrow E}$$

$$\frac{}{\Sigma;\Gamma, x : s[\mathsf{pc}] \vdash x : s \sqcup \mathsf{pc} \Rightarrow [x]}$$

$$\frac{\Sigma;\Gamma \vdash bv : t \Rightarrow M}{\Sigma;\Gamma[\mathsf{pc}] \vdash bv_a : (t, a \sqcup \mathsf{pc}) \Rightarrow \mathsf{ref}_{a \sqcup \mathsf{pc}}\,(M : \overline{t})}$$

$$\frac{\Sigma;\Gamma[\mathsf{pc}] \vdash e_1 : (\mathsf{bool}, a) \Rightarrow E_1 \quad \Sigma;\Gamma[\mathsf{pc} \sqcup a] \vdash e_2 : s \Rightarrow E_2 \quad \Sigma;\Gamma[\mathsf{pc} \sqcup a] \vdash e_3 : s \Rightarrow E_3}{\Sigma;\Gamma[\mathsf{pc}] \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : s \Rightarrow \begin{array}{l}\mathsf{let}\ y = E_1\ \mathsf{in}\\ \mathsf{let}\ y' = !y\ \mathsf{in}\\ \mathsf{run}(\mathsf{if}\ y'\ \mathsf{then}\ \mathsf{val}\ E_2\ \mathsf{else}\ \mathsf{val}\ E_3)\end{array}}$$

$$\frac{\Sigma;\Gamma[\mathsf{pc}] \vdash e_1 : (s' \xrightarrow{\mathsf{pc}'} s, a) \Rightarrow E_1 \quad \Sigma;\Gamma[\mathsf{pc}] \vdash e_2 : s' \Rightarrow E_2 \quad \mathsf{pc} \sqcup a \sqsubseteq \mathsf{pc}'}{\Sigma;\Gamma[\mathsf{pc}] \vdash e_1 e_2 : s \sqcup a \Rightarrow \begin{array}{l}\mathsf{let}\ y_1 = E_1\ \mathsf{in}\\ \mathsf{let}\ y_2 = E_2\ \mathsf{in}\\ \mathsf{let}\ y_1' = !y_1\ \mathsf{in}\\ \mathsf{run}\ (y_1' y_2)\end{array}}$$

$$\frac{\Sigma;\Gamma[\mathsf{pc}] \vdash e : s \Rightarrow E}{\Sigma;\Gamma[\mathsf{pc}] \vdash \mathsf{ref}\ (e : (t, a)) : (\mathsf{ref}\ (t, a), \mathsf{pc}) \Rightarrow \begin{array}{l}\mathsf{let}\ y = E\ \mathsf{in}\\ \mathsf{ref}_a\,(y : \overline{(t, a)})\end{array}}$$

$$\frac{\Sigma;\Gamma[\mathsf{pc}] \vdash e : (\mathsf{ref}\ s, a) \Rightarrow E}{\Sigma;\Gamma[\mathsf{pc}] \vdash !e : s \sqcup a \Rightarrow \mathsf{let}\ y = E\ \mathsf{in}\ \mathsf{let}\ y' = !y\ \mathsf{in}\ !y'}$$

$$\frac{\Sigma;\Gamma[\mathsf{pc}] \vdash e_1 : (\mathsf{ref}\ (t, b), a) \Rightarrow E_1 \quad \Sigma;\Gamma[\mathsf{pc}] \vdash e_2 : (t, b) \Rightarrow E_2 \quad a \sqsubseteq b}{\Sigma;\Gamma[\mathsf{pc}] \vdash e_1 := e_2 : (1, \mathsf{pc}) \Rightarrow \begin{array}{l}\mathsf{let}\ y_1 = E_1\ \mathsf{in}\\ \mathsf{let}\ y_2 = E_2\ \mathsf{in}\\ \mathsf{let}\ y_1' = !y_1\ \mathsf{in}\\ \mathsf{let}\ \_ = y_1' := y_2\ \mathsf{in}\\ \mathsf{ref}_{\mathsf{pc}}\,(* : 1)\end{array}}$$

$$\frac{\Sigma;\Gamma[\mathsf{pc}] \vdash e : s_1 \Rightarrow E \quad \vdash s_1 \leqslant s_2}{\Sigma;\Gamma[\mathsf{pc}] \vdash e : s_2 \Rightarrow E}$$

Fig. 11. $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ expression encoding.

In a $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ function of type $s \xrightarrow{\mathsf{pc}} s'$ the program counter annotation pc is a conservative approximation of the information gained by the body of the function. Therefore, values written by the body must have security level at least pc. Thus, the corresponding writes in the translation must have write level at least pc. Consequently, the corresponding translated type for a function is $\overline{s} \to \bigcirc_{(\bot,\mathsf{pc})}\ \overline{s'}$.

The encoding for $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ expressions is given by a pair of judgments $\Sigma;\Gamma \vdash bv : t \Rightarrow M$ and $\Sigma;\Gamma[\mathsf{pc}] \vdash e : s \Rightarrow E$, shown in Figures 10 and 11. We assume that

the metavariable y stands for variables in our calculus that do not appear in $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ programs.

Type-correctness To show that our proposed encoding preserves typing, we first have to establish the following facts. The first shows that our encoding judgments agree with $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ typing judgments; the second shows that the encoding preserves subtyping.

Lemma 6.2

1. If $\Sigma;\Gamma \vdash bv : t \Rightarrow M$ then $\Sigma;\Gamma \vdash bv : t$
2. If $\Sigma;\Gamma \vdash e : s \Rightarrow E$ then $\Sigma;\Gamma \vdash e : s$

Proof
By induction on the given derivations. Observe that in each case, the rules of the encoding judgment have the same premises as the corresponding typing rules. □

Lemma 6.3 (*Subtyping Translation*)

1. If $\vdash t' \leqslant t$ then $\vdash \overline{t'} \leqslant \overline{t}$
2. If $\vdash s' \leqslant s$ then $\vdash \overline{s'} \leqslant \overline{s}$

Proof
Both parts simultaneously, by induction on the given derivation. □

Finally, we need to extend our type-translation to store types

$$\overline{\Sigma, \ell : s} = \overline{\Sigma}, \ell : \overline{s}$$

We are now ready to show type-correctness.

Theorem 6.4 (*Well-typed Translation*)

1. If $\Sigma;\Gamma \vdash bv : t \Rightarrow M$ then $\overline{\Sigma};\overline{\Gamma} \vdash M : \overline{t}$
2. If $\Sigma;\Gamma[\mathsf{pc}] \vdash e : s \Rightarrow E$ then $\overline{\Sigma};\overline{\Gamma} \vdash E \div_{(\bot,\mathsf{pc})} \overline{s}$

The proof is by simultaneous induction on the given derivations. The full proof is available in Appendix D.

Non-interference Of course a type correct (but insecure) embedding could be constructed by ignoring the security levels of the source and placing everything at level $\bot$. We wish to show that the embedding is actually secure. To do so, we show that an instance of non-interference for $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ is preserved by our translation.

Theorem 6.5 ($\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ *non-interference*)
Suppose $\Sigma_0; x : (t,a)[b] \vdash f : (\mathsf{bool}, b) \Rightarrow F$ where $a \not\sqsubseteq b$, and suppose that H, Σ are such that $\Sigma \supseteq \overline{\Sigma_0}$, and $\vdash H : \Sigma$. If $\Sigma;\cdot \vdash \ell_i : \mathsf{refr}_a\, \overline{t}$ for $i = 1,2$ and if there exist $H_1, H_2, \Sigma_1, \Sigma_2, V_1, V_2$ such that

$$(H', \Sigma', F[\ell_i/x]) \rightarrow^* (H_i, \Sigma_i, [V_i])$$

for $i = 1,2$, then $V_i = \ell'_i$ and $H_1(\ell'_1) = H_2(\ell'_2)$ as booleans.

Proof

1. From the type-correctness of the translation, and since the argument locations ℓ_i are out of view, by the non-interference theorem we conclude that $\vdash (H_1, \Sigma_1, [V_1]) \approx_b (H_2, \Sigma_2, [V_2]) \div_{(b,b)} \mathsf{refr}_b\, \mathsf{bool}$
2. By inversion and by Regularity (Lemma C.14) and Canonical Forms (Lemma C.5), each V_i must be some store location $\ell'_i \in \mathrm{dom}(\Sigma_i)$ and furthermore $\Sigma_1; \Sigma_2; \cdot \vdash V_1 \approx_b V_2 : \mathsf{refr}_b\, \mathsf{bool}$
3. By inversion on the latter equivalence(Lemma C.15), each $\Sigma_i(\ell'_i)$ must either be out of view, or $\ell'_1 = \ell'_2$ with $\mathsf{Level}(\ell'_i) \sqsubseteq b$. But since $\Sigma_i(\ell'_i)$ must be a subtype of $\mathsf{refr}_b\, \mathsf{bool}$, it cannot be out of view for a b-observer.
4. Therefore, $\ell'_1 = \ell'_2$ are in the set of in-view locations $U = \mathrm{dom}(\Sigma_1) \cap \mathrm{dom}(\Sigma_2) \cap \downarrow(b)$, and by inversion on the store equivalence $\vdash (H_1 : \Sigma_1) \approx_\zeta^U (H_2 : \Sigma_2)$, the values in the respective stores must, in turn, be equivalent $\Sigma_1; \Sigma_2; \cdot \vdash H_1(\ell'_1) \approx_b H_2(\ell'_2) : \mathsf{bool}$
5. Since bool is informative at any security level, by inversion, it must be the case that $H_1(\ell'_1) = H_2(\ell'_2)$.

□

7 Related work

There is a large body of existing work on type systems for secure information flow. Volpano *et al.* (1996) first showed how to formulate an information flow analysis as a type system. An excellent survey by Sabelfeld & Myers (2003) outlines the key ideas in the design of secure programming languages.

Our account is most related to the Dependency Core Calculus (Abadi *et al.*, 1999). Like our language, DCC uses a family of monads to reason about information flow. However in DCC, terms of monadic type are used to seal up values at a security level. In our account, monads are used in a more traditional role as a means of threading state through a program. Central to DCC is the notion of *protectedness* of a type at a security level. If T is protected at a then T is at least as secure as a. This is closely related to our notion of informativeness.

When viewed through the lens of the encoding of (a pure subset of) $\lambda_{\mathrm{SEC}}^{\mathrm{REF}}$, the two relations serve the same purpose, ensuring that a computation's output is at least as secure as its inputs. In DCC, this is done directly. In our account, this occurs indirectly: to access a value carrying information only at a particular level, a computation must adopt a read level at least as high. (However, our account also offers the facility – not employed in the $\lambda_{\mathrm{SEC}}^{\mathrm{REF}}$ embedding – not to seal all computations' return values in order to obtain a $\bot$ effective read level).

The definitions of protectedness and informativeness are the same on the standard type operators, but do not include the idiosyncratic cases: our language has no analog of DCC's monad, nor does DCC contain references or a traditional (i.e. effects-oriented) monad. Moreover, if it did, we conjecture that DCC's definition for these would be somewhat different from ours. Nevertheless, the similarity between the two suggests that our account might be profitably combined with DCC to

produce a language capable of expressing security in both value-oriented and store-oriented fashions.

A further similarity exists between the *tampering levels* of Honda and Yoshida (2002) and informativeness. They work in a concurrent setting of a typed π-calculus, and the tampering level of a process represents the least security level that may observe the effects of a process of a given type. They present a calculus in the style of Smith & Volpano (1998) extended with local variables, reference types and higher-order procedures and a translation of it into their typed process calculus. Much of the complexity of their language stems from tracking the action set of a command, that is, the references (conflated with program variables) that a command may read or write. Our language may be seen as a restatement of their language in a more conventional monadic style. In the setting of Honda & Yoshida (2002), our upcall rule (exploiting the informativeness judgment) would correspond to leaving out the information that a command read from some variables from its action set whenever the command does not tamper below a certain security level.

Harrison *et al.* (2003) observed that monads and monad transformers may be used to separate pieces of the state with different security levels, thus ensuring a kind of non-interference via properties of the state monad transformer. However their system does not statically rule out insecure flows when computations at different security levels are combined. Instead, the system dynamically prevents security leaks by channeling communication between computations at different security levels through a trusted kernel.

8 Conclusion

We give an account of secure information flow in the context of a higher-order language with mutable state. Moreover, motivated by a low-level store-oriented view of computation, we arrive at a view of security based on lax logic. Rather than sealing values at a security level, we instead associate security with the store. A family of monadic types is used to keep track of the effects of computations. To account for upcalls, we classify the informativeness of types at particular security levels.

Since we treat terms apart from the effectful expressions, our approach can straightforwardly encompass additional type constructors. The question of how to account for additional effects requires further work. From the point of view of non-interference, effects introduce the possibility of different behavior from seemingly related expressions. We expect that by further refining the monadic type to restrict the behavior of related terms, we may be able to account for effects such as I/O or non-local control transfers.

Certain complications beyond those discussed in this paper remain in developing a typed assembly language that tracks information flow. One problem to be dealt with is the re-use of registers between low-security and high-security computations. Any mutation of a register by a high security computation could potentially be observed once it returns to a low-security caller. As a result it is necessary to exploit

informativeness to ensure that the contents of registers are not informative to the caller. We conjecture that informativeness in conjunction with linear continuations (2002) will prove invaluable to the design of a secure TAL.

Our formulation of the monadic language is in the style of Pfenning & Davies (2001). One avenue of future work is to study whether there is a formulation of information flow in a modal logic that decomposed our monad into the possibility and necessity modalities.

Incorporating concurrency is another direction for future work. Smith *et al.* (1998) show that in a language with parallel composition, allowing loops to depend on high-security locations leads to security leaks. Their solution is to disallow such loops outright. Since looping can be simulated in our account via back-patching in combination with informativeness (see section 3.3), it is not clear how to adopt their solution to (a concurrent extension of) our approach. Zdancewic (2002) observes that insecure concurrent programs exhibit race conditions on low-security locations. He shows that if alias information is used to disallow such data races, a non-interference result can be established. We expect that his approach may be adopted to our setting.

A general open problem in the area of secure programming languages is how to devise a type system for a language with *declassification* operations. Declassification occurs when a low-security computation makes use of a high-security value, but in a way such that the information gained from the high-security value is deemed an acceptable leak. Recently, Zdancewic & Myers (2001a) showed how to characterize so-called *robust declassification* in programs such that an attacker may observe the declassified values, but may not exploit them to gain additional high-security information. Zdancewic (2003) then gives a type system for robust declassification. Since declassification is fundamentally an operation, we conjecture that our store-oriented viewpoint could be meshed with Zdancewic and Myers to provide a logic of declassification.

A Judgments

$\boxed{\vdash A \nearrow a}$

A.1 Informativeness judgment rules

$$\frac{}{\vdash A \nearrow \bot}\ (1) \qquad \frac{}{\vdash 1 \nearrow a}\ (2) \qquad \frac{\vdash B \nearrow a}{\vdash A \to B \nearrow a}\ (3)$$

$$\frac{\vdash A \nearrow a}{\vdash \bigcirc_{(r,w)} A \nearrow w \sqcap a}\ (4) \qquad \frac{}{\vdash \mathsf{ref}_b\, A \nearrow b}\ (5) \qquad \frac{\vdash A \nearrow a}{\vdash \mathsf{ref}_b\, A \nearrow a}\ (6)$$

$$\frac{\vdash A \nearrow a}{\vdash \mathsf{refr}_b\, A \nearrow a}\ (7) \qquad \frac{}{\vdash \mathsf{refr}_b\, A \nearrow b}\ (8) \qquad \frac{}{\vdash \mathsf{refw}_a\, A \nearrow a}\ (9)$$

$$\frac{\vdash A \nearrow a \quad b \sqsubseteq a}{\vdash A \nearrow b}\ (10) \qquad \frac{\vdash A \nearrow a \quad \vdash A \nearrow b}{\vdash A \nearrow a \sqcup b}\ (11)$$

A.2 Typing judgment rules

$\boxed{\vdash A \leqslant B}$

$$\frac{}{\vdash A \leqslant A}\ (12)$$

$$\frac{\vdash A \leqslant A' \quad \vdash B' \leqslant B}{\vdash A' \to B' \leqslant A \to B}\ (13) \qquad \frac{\vdash A \leqslant B \quad o \preceq o'}{\vdash \bigcirc_o A \leqslant \bigcirc_{o'} B}\ (14)$$

$$\frac{\vdash A \leqslant B \quad a \sqsubseteq b}{\vdash \mathsf{ref}_a\, A \leqslant \mathsf{refr}_b\, B}\ (15) \qquad \frac{\vdash B \leqslant A \quad b \sqsubseteq a}{\vdash \mathsf{ref}_a\, A \leqslant \mathsf{refw}_b\, B}\ (16)$$

$$\frac{\vdash A \leqslant B \quad a \sqsubseteq b}{\vdash \mathsf{refr}_a\, A \leqslant \mathsf{refr}_b\, B}\ (17) \qquad \frac{\vdash B \leqslant A \quad b \sqsubseteq a}{\vdash \mathsf{refw}_a\, A \leqslant \mathsf{refw}_b\, B}\ (18)$$

$\boxed{\Sigma;\Gamma \vdash M : A}$

$$\frac{}{\Sigma;\Gamma \vdash x : \Gamma(x)}\ (19) \qquad \frac{}{\Sigma;\Gamma \vdash \ell : \mathsf{ref}_{\mathsf{Level}(\ell)}\, \Sigma(\ell)}\ (20)$$

$$\frac{}{\Sigma;\Gamma \vdash * : 1}\ (21) \qquad \frac{}{\Sigma;\Gamma \vdash \mathsf{true} : \mathsf{bool}}\ (22) \qquad \frac{}{\Sigma;\Gamma \vdash \mathsf{false} : \mathsf{bool}}\ (23)$$

$$\frac{\Sigma;\Gamma \vdash M : \mathsf{bool} \quad \Sigma;\Gamma \vdash N_1 : A \quad \Sigma;\Gamma \vdash N_2 : A}{\Sigma;\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : A}\ (24)$$

$$\frac{\Sigma;\Gamma, x : A \vdash M : B}{\Sigma;\Gamma \vdash \lambda x : A.M : A \to B}\ (25) \qquad \frac{\Sigma;\Gamma \vdash M : A \to B \quad \Sigma;\Gamma \vdash N : A}{\Sigma;\Gamma \vdash M\ N : B}\ (26)$$

$$\frac{\Sigma;\Gamma \vdash E \div_o A}{\Sigma;\Gamma \vdash \mathsf{val}\ E : \bigcirc_o A}\ (27) \qquad \frac{\Sigma;\Gamma \vdash M : A \quad \vdash A \leqslant B}{\Sigma;\Gamma \vdash M : B}\ (28)$$

$\boxed{\Sigma;\Gamma \vdash E \div_o A}$

$$\frac{\Sigma;\Gamma \vdash M : A}{\Sigma;\Gamma \vdash [M] \div_{(\bot,\top)} A}\ (29) \qquad \frac{\Sigma;\Gamma \vdash M : \bigcirc_o A \quad \Sigma;\Gamma, x : A \vdash E \div_o B}{\Sigma;\Gamma \vdash \mathsf{let\ val}\ x = M\ \mathsf{in}\ E \div_o B}\ (30)$$

$$\frac{\Sigma;\Gamma \vdash M : A}{\Sigma;\Gamma \vdash \mathsf{ref}_a\, (M : A) \div_{(\bot,\top)} \mathsf{ref}_a\, A}\ (31)$$

$$\frac{\Sigma;\Gamma \vdash M : \mathsf{refr}_a\, A}{\Sigma;\Gamma \vdash !M \div_{(a,\top)} A}\ (32) \qquad \frac{\Sigma;\Gamma \vdash M : \mathsf{refw}_a\, A \quad \Sigma;\Gamma \vdash N : A}{\Sigma;\Gamma \vdash M := N \div_{(\bot,a)} 1}\ (33)$$

$$\frac{\Sigma;\Gamma \vdash E \div_{o'} A \quad o' \preceq o}{\Sigma;\Gamma \vdash E \div_o A}\ (34) \qquad \frac{\Sigma;\Gamma \vdash E \div_{(r,w)} A \quad \vdash A \nearrow r}{\Sigma;\Gamma \vdash E \div_{(\bot,w)} A}\ (35)$$

$$\frac{\Sigma;\Gamma \vdash E \div_o B \quad \vdash B \leqslant C}{\Sigma;\Gamma \vdash E \div_o C}\ (36)$$

$\boxed{\vdash H : \Sigma}$

$$\frac{\mathrm{dom}(\Sigma) = \{\ell_1, \ldots, \ell_n\} \quad \Sigma;\cdot \vdash V_i : \Sigma(\ell_i)\ \text{ for } 1 \leqslant i \leqslant n}{\vdash \{\ell_1 \mapsto V_1, \ldots, \ell_n \mapsto V_n\} : \Sigma}\ (37)$$

$\boxed{\vdash S \div_o A}$

$$\frac{\vdash H : \Sigma \quad \Sigma;\cdot \vdash E \div_o A}{\vdash (H, \Sigma, E) \div_o A}\ (38)$$

Derived typing rules for syntactic sugar

$$\frac{\Sigma;\Gamma \vdash E \div_o A \quad \Sigma;\Gamma, x : A \vdash F \div_o C}{\Sigma;\Gamma \vdash \mathsf{let}\ x = E\ \mathsf{in}\ F \div_o C} \qquad \frac{\Sigma;\Gamma \vdash M : \bigcirc_o C}{\Sigma;\Gamma \vdash \mathsf{run}\ M \div_o C}$$

A.3 Equivalent view judgments rules

$\boxed{\Sigma_1;\Sigma_2;\Gamma \vdash M_1 \approx_\zeta M_2 : A}$

$$\frac{\vdash A \nearrow a \quad a \not\sqsubseteq \zeta \quad \Sigma_1;\Gamma \vdash V_1 : A \quad \Sigma_2;\Gamma \vdash V_2 : A}{\Sigma_1;\Sigma_2;\Gamma \vdash V_1 \approx_\zeta V_2 : A}\ (39)$$

$$\frac{}{\Sigma_1;\Sigma_2;\Gamma \vdash * \approx_\zeta * : 1}\ (40) \qquad \frac{}{\Sigma_1;\Sigma_2;\Gamma \vdash x \approx_\zeta x : \Gamma(x)}\ (41)$$

$$\frac{}{\Sigma_1;\Sigma_2;\Gamma \vdash \mathsf{true} \approx_\zeta \mathsf{true} : \mathsf{bool}}\ (42) \qquad \frac{}{\Sigma_1;\Sigma_2;\Gamma \vdash \mathsf{false} \approx_\zeta \mathsf{false} : \mathsf{bool}}\ (43)$$

$$\frac{\begin{array}{l}\Sigma_1;\Sigma_2;\Gamma \vdash M_1 \approx_\zeta M_2 : \mathsf{bool} \\ \Sigma_1;\Sigma_2;\Gamma \vdash N_1 \approx_\zeta N_2 : A \\ \Sigma_1;\Sigma_2;\Gamma \vdash P_1 \approx_\zeta P_2 : A\end{array}}{\Sigma_1;\Sigma_2;\Gamma \vdash \begin{array}{ll}\mathsf{if}\ M_1\ \mathsf{then}\ N_1\ \mathsf{else}\ P_1 & \approx_\zeta \\ \mathsf{if}\ M_2\ \mathsf{then}\ N_2\ \mathsf{else}\ P_2 & : A\end{array}}\ (44)$$

$$\frac{\Sigma_1;\Sigma_2;\Gamma, x : A \vdash M_1 \approx_\zeta M_2 : B}{\Sigma_1;\Sigma_2;\Gamma \vdash \lambda x : A.M_1 \approx_\zeta \lambda x : A.M_2 : A \to B}\ (45)$$

$$\frac{\begin{array}{l}\Sigma_1;\Sigma_2;\Gamma \vdash M_1 \approx_\zeta M_2 : A \to B \\ \Sigma_1;\Sigma_2;\Gamma \vdash N_1 \approx_\zeta N_2 : A\end{array}}{\Sigma_1;\Sigma_2;\Gamma \vdash M_1 N_1 \approx_\zeta M_2 N_2 : B}\ (46)$$

$$\frac{\mathsf{Level}(\ell) \sqsubseteq \zeta \quad \Sigma_1(\ell) = \Sigma_2(\ell)}{\Sigma_1;\Sigma_2;\Gamma \vdash \ell \approx_\zeta \ell : \mathsf{ref}_{\mathsf{Level}(\ell)}\, \Sigma_1(\ell)}\ (47) \qquad \frac{\Sigma_1;\Sigma_2;\Gamma \vdash E_1 \approx_\zeta E_2 \div_o A}{\Sigma_1;\Sigma_2;\Gamma \vdash \mathsf{val}\ E_1 \approx_\zeta \mathsf{val}\ E_2 : \bigcirc_o A}\ (48)$$

$$\frac{\Sigma_1;\Sigma_2;\Gamma \vdash M_1 \approx_\zeta M_2 : A \quad \vdash A \leqslant B}{\Sigma_1;\Sigma_2;\Gamma \vdash M_1 \approx_\zeta M_2 : B}\ (49)$$

$\boxed{\Sigma_1;\Sigma_2;\Gamma \vdash E_1 \approx_\zeta E_2 \div_o C}$

$$\frac{\Sigma_1;\Sigma_2;\Gamma \vdash E_1 \approx_\zeta E_2 \div_{o'} C \quad o' \preceq o}{\Sigma_1;\Sigma_2;\Gamma \vdash E_1 \approx_\zeta E_2 \div_o C}\ (50)$$

$$\frac{\Sigma_1;\Sigma_2;\Gamma \vdash E_1 \approx_\zeta E_2 \div_{(r,w)} C \quad \vdash C \nearrow r}{\Sigma_1;\Sigma_2;\Gamma \vdash E_1 \approx_\zeta E_2 \div_{(\bot,w)} C}\ (51)$$

$$\frac{\Sigma_1;\Sigma_2;\Gamma \vdash E_1 \approx_\zeta E_2 \div_o B \quad \vdash B \leqslant C}{\Sigma_1;\Sigma_2;\Gamma \vdash E_1 \approx_\zeta E_2 \div_o C}\ (52) \qquad \frac{\Sigma_1;\Sigma_2;\Gamma \vdash M_1 \approx_\zeta M_2 : C}{\Sigma_1;\Sigma_2;\Gamma \vdash [M_1] \approx_\zeta [M_2] \div_{(\bot,\top)} C}\ (53)$$

$$\frac{\begin{array}{l}\Sigma_1;\Sigma_2;\Gamma \vdash M_1 \approx_\zeta M_2 : \bigcirc_o A \\ \Sigma_1;\Sigma_2;\Gamma, x : A \vdash E_1 \approx_\zeta E_2 \div_o C\end{array}}{\Sigma_1;\Sigma_2;\Gamma \vdash \begin{array}{ll}\mathsf{let\ val}\ x = M_1\ \mathsf{in}\ E_1 & \approx_\zeta \\ \mathsf{let\ val}\ x = M_2\ \mathsf{in}\ E_2 & \div_o C\end{array}}\ (54)$$

$$\frac{\Sigma_1;\Sigma_2;\Gamma \vdash M_1 \approx_\zeta M_2 : A}{\Sigma_1;\Sigma_2;\Gamma \vdash \begin{array}{ll}\mathsf{ref}_a\,(M_1 : A) & \approx_\zeta \\ \mathsf{ref}_a\,(M_2 : A) & \div_{(\bot,\top)} \mathsf{ref}_a\,A\end{array}} \quad (55) \qquad \frac{\Sigma_1;\Sigma_2;\Gamma \vdash M_1 \approx_\zeta M_2 : \mathsf{refr}_a\,A}{\Sigma_1;\Sigma_2;\Gamma \vdash !M_1 \approx_\zeta !M_2 \div_{(a,\top)} A} \quad (56)$$

$$\frac{\begin{array}{c}\Sigma_1;\Sigma_2;\Gamma \vdash M_1 \approx_\zeta M_2 : \mathsf{refw}_a\,A \\ \Sigma_1;\Sigma_2;\Gamma \vdash N_1 \approx_\zeta N_2 : A\end{array}}{\Sigma_1;\Sigma_2;\Gamma \vdash M_1 := N_1 \approx_\zeta M_2 := N_2 \div_{(\bot,a)} 1} \quad (57)$$

$\boxed{\vdash (H_1 : \Sigma_1) \approx^U_\zeta (H_2 : \Sigma_2)}$

$$\frac{\begin{array}{l}\vdash H_i : \Sigma_i \text{ for } i = 1,2 \\ \Sigma_1 \restriction U = \Sigma_2 \restriction U \\ \Sigma_1;\Sigma_2;\cdot \vdash H_1(\ell) \approx_\zeta H_2(\ell) : \Sigma_1(\ell) \text{ for all } \ell \in U\end{array}}{\vdash (H_1 : \Sigma_1) \approx^U_\zeta (H_2 : \Sigma_2)} \quad (58)$$

$\boxed{\vdash S_1 \approx_\zeta S_2 \div_o C}$

$$\frac{\begin{array}{l}\vdash (H_1 : \Sigma_1) \approx_\zeta^{\mathrm{dom}(\Sigma_1)\cap\mathrm{dom}(\Sigma_2)\cap\downarrow(\zeta)} (H_2 : \Sigma_2) \\ \Sigma_1;\Sigma_2;\cdot \vdash E_1 \approx_\zeta E_2 \div_o C\end{array}}{\vdash (H_1,\Sigma_1,E_1) \approx_\zeta (H_2,\Sigma_2,E_2) \div_o C} \quad (59)$$

B Evaluation rules

$\boxed{M \to M'}$

$$\frac{M \to M'}{\mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 \to \mathsf{if}\ M'\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2}\ \textsc{If1}$$

$$\frac{}{\mathsf{if\ true\ then}\ N_1\ \mathsf{else}\ N_2 \to N_1}\ \textsc{IfTrue} \qquad \frac{}{\mathsf{if\ false\ then}\ N_1\ \mathsf{else}\ N_2 \to N_2}\ \textsc{IfFalse}$$

$$\frac{M \to M'}{MN \to M'N}\ \textsc{App1} \qquad \frac{N \to N'}{VN \to VN'}\ \textsc{App2}$$

$$\frac{}{(\lambda x : A.M)V \to M[V/x]}\ \textsc{App}$$

$\boxed{S \to S'}$

$$\frac{M \to M'}{(H,\Sigma,[M]) \to (H,\Sigma,[M'])}\ \textsc{Ret1}$$

$$\frac{M \to M'}{\begin{array}{l}(H,\Sigma,\mathsf{let\ val}\ x = M\ \mathsf{in}\ E) \to \\ (H,\Sigma,\mathsf{let\ val}\ x = M'\ \mathsf{in}\ E)\end{array}}\ \textsc{Letval1} \qquad \frac{(H,\Sigma,E) \to (H',\Sigma',E')}{\begin{array}{l}(H,\Sigma,\mathsf{let\ val}\ x = \mathsf{val}\ E\ \mathsf{in}\ F) \to \\ (H',\Sigma',\mathsf{let\ val}\ x = \mathsf{val}\ E'\ \mathsf{in}\ F)\end{array}}\ \textsc{LetvalVal}$$

$$\frac{}{(H,\Sigma,\mathsf{let\ val}\ x = \mathsf{val}\ [V]\ \mathsf{in}\ E) \to (H,\Sigma,E[V/x])}\ \textsc{Letval}$$

$$\frac{M \to M'}{\begin{array}{l}(H,\Sigma,\mathsf{ref}_a\,(M : A)) \to \\ (H,\Sigma,\mathsf{ref}_a\,(M' : A))\end{array}}\ \textsc{Ref1} \qquad \frac{\ell \notin \mathrm{dom}(H) \quad \mathsf{Level}(\ell) = a}{\begin{array}{l}(H,\Sigma,\mathsf{ref}_a\,(V : A)) \to \\ (H\{\ell \mapsto V\},\Sigma\{\ell : A\},[\ell])\end{array}}\ \textsc{Ref}$$

$$\frac{M \to M'}{(H, \Sigma, !M) \to (H, \Sigma, !M')}\ \textsc{Bang1} \qquad \frac{}{(H, \Sigma, !\ell) \to (H, \Sigma, [H(\ell)])}\ \textsc{Bang}$$

$$\frac{M \to M'}{(H, \Sigma, M := N) \to (H, \Sigma, M' := N)}\ \textsc{Assn1} \qquad \frac{N \to N'}{(H, \Sigma, V := N) \to (H, \Sigma, V := N')}\ \textsc{Assn2}$$

$$\frac{\ell \in \mathrm{dom}(H)}{(H, \Sigma, \ell := V) \to (H\{\ell \mapsto V\}, \Sigma, [*])}\ \textsc{Assn}$$

C Proofs

C.1 Type safety proof

C.1.1 Properties of informativeness and subtyping

Before we go on to prove type safety and non-interference, we take the time to prove several (standard) lemmas.

Lemma C.1 (*Informativeness Inversion*)
If $\vdash A \nearrow a$ and if

- $A = \mathsf{bool}$ then $a = \bot$
- $A = B \to C$ then $\vdash C \nearrow a$
- $A = \bigcirc_{(r,w)} B$ then $a \sqsubseteq w \sqcap b$ and $\vdash B \nearrow b$
- $A = \mathsf{ref}_b\, B$ then $\vdash B \nearrow c$ and $a \sqsubseteq c \sqcup b$
- $A = \mathsf{refr}_b\, B$ then $\vdash B \nearrow c$ and $a \sqsubseteq c \sqcup b$
- $A = \mathsf{refw}_b\, B$ then $a \sqsubseteq b$

Proof
by induction on the given derivation. By cases on the last rule used. □

Lemma C.2 (*Subtyping Inversion*)
If $\vdash A' \leqslant A$ and if

- $A = 1$ then $A' = 1$
- $A = \mathsf{bool}$ then $A' = \mathsf{bool}$
- $A = B \to C$ then $A' = B' \to C'$ and $\vdash B \leqslant B'$ and $\vdash C' \leqslant C$
- $A = \mathsf{ref}_a\, B$ then $A' = \mathsf{ref}_a\, B$
- $A = \mathsf{refr}_a\, B$ then
  - either $A' = \mathsf{refr}_{a'}\, B'$ with $\vdash B' \leqslant B$ and $a' \sqsubseteq a$
  - or $A' = \mathsf{ref}_{a'}\, B'$ with $\vdash B' \leqslant B$ and $a' \sqsubseteq a$
- $A = \mathsf{refw}_a\, B$ then
  - either $A' = \mathsf{refw}_{a'}\, B'$ with $\vdash B \leqslant B'$ and $a \sqsubseteq a'$
  - or $A' = \mathsf{ref}_{a'}\, B'$ with $\vdash B \leqslant B'$ and $a \sqsubseteq a'$
- $A = \bigcirc_o B$ then $A' = \bigcirc_{o'} B'$ and $\vdash B' \leqslant B$ and $o' \preceq o$

and moreover, all the result derivations are subderivations of the given derivation.

Proof
by cases on the last rule used in the given derivation. Each case follows immediately from the rules. □

C.1.2 Typing judgment properties

Lemma C.3 (*Substitution*)
If $\Sigma; \Gamma, \Gamma' \vdash M : A$ and

1. if $\Sigma; \Gamma, x : A, \Gamma' \vdash N : B$ then $\Sigma; \Gamma, \Gamma' \vdash N[M/x] : B$
2. if $\Sigma; \Gamma, x : A, \Gamma' \vdash E \div_o B$ then $\Sigma; \Gamma, \Gamma' \vdash E[M/x] \div_o B$

Proof
Parts (1) and (2) simultaneously by induction on $\Sigma; \Gamma, x : A, \Gamma' \vdash N : B$ (or $\Sigma; \Gamma, x : A, \Gamma' \vdash E \div_o B$). By cases on the last rule used. □

Lemma C.4 (*Inversion*)
Two parts:

- If $\Sigma; \Gamma \vdash M : A$ and
  1. if $M = x$ then $\vdash \Gamma(x) \leqslant A$
  2. if $M = *$ then $\vdash 1 \leqslant A$
  3. if $M = \mathsf{true}$ or $M = \mathsf{false}$ then $\vdash \mathsf{bool} \leqslant A$
  4. if $M = \mathsf{if}\ N_1\ \mathsf{then}\ N_2\ \mathsf{else}\ N_3$ then $\Sigma; \Gamma \vdash N_1 : \mathsf{bool}$, $\Sigma; \Gamma \vdash N_2 : B$, $\Sigma; \Gamma \vdash N_3 : B'$, and $\vdash B \leqslant A$, $\vdash B' \leqslant A$
  5. if $M = \lambda x : B.N$ then $\Sigma; \Gamma, x : B \vdash N : C$ and $\vdash B \to C \leqslant A$
  6. if $M = NP$ then $\Sigma; \Gamma \vdash N : B \to C$ and $\Sigma; \Gamma \vdash P : B$ and $\vdash C \leqslant A$
  7. if $M = \ell$ then $\vdash \mathsf{ref}_{\mathsf{Level}(\ell)}\, \Sigma(\ell) \leqslant A$
  8. if $M = \mathsf{val}\ E$ then $\Sigma; \Gamma \vdash E \div_o B$ and $\vdash \bigcirc_o B \leqslant A$
- If $\Sigma; \Gamma \vdash E \div_o A$ and
  1. if $E = [M]$ then $\Sigma; \Gamma \vdash M : A$
  2. if $E = \mathsf{let\ val}\ x = M\ \mathsf{in}\ F$ then $\Sigma; \Gamma \vdash M : \bigcirc_{o'} B$ and $\Sigma; \Gamma, x : B \vdash F \div_{o'} C$, $\vdash C \leqslant A$ and $o' = (r', w')$ with either $o' \preceq o$ or $\vdash C \nearrow r'$ and $(\bot, w') \preceq o$
  3. if $E = \mathsf{ref}_a\,(M : B)$ then $\vdash \mathsf{ref}_a\, B \leqslant A$ and $\Sigma; \Gamma \vdash M : B$
  4. if $E = !M$ then $\Sigma; \Gamma \vdash M : \mathsf{refr}_a\, B$ and $\vdash B \leqslant C$, $\vdash C \leqslant A$ and either $(a, \top) \preceq o$ or $\vdash C \nearrow a$
  5. if $E = M := N$ then $\Sigma; \Gamma \vdash M : \mathsf{refw}_a\, B$, $\Sigma; \Gamma \vdash N : B$, $\vdash 1 \leqslant A$, and $(\bot, a) \preceq o$

Proof
by induction on the given derivation. By cases on the last rule used.

For part (1), in cases of rules (21) − −(27) the result is immediate, by rule (12). In case of rule (28), the result follows by IH, and transitivity of subtyping (which can be shown to be admissible).

For part (2), the cases for rules (29), (30), (31), (32), (33) are immediate. The cases for rules (34), (35) and (36) follow by IH, by subcases on E. □

Lemma C.5 (*Canonical Forms*)
If $\Sigma; \cdot \vdash V : A$ and

1. if $A = 1$ then $V = *$
2. if $A = \mathsf{bool}$ then $V = \mathsf{true}$ or $V = \mathsf{false}$

3. if $A = B \to C$ then $V = \lambda x : B'.M$
4. if $A = \mathsf{ref}_a\, B$ then $V = \ell$ and $\ell \in \mathrm{dom}(\Sigma)$
5. if $A = \mathsf{refr}_a\, B$ then $V = \ell$ and $\ell \in \mathrm{dom}(\Sigma)$
6. if $A = \mathsf{refw}_a\, B$ then $V = \ell$ and $\ell \in \mathrm{dom}(\Sigma)$
7. if $A = \bigcirc_o B$ then $V = \mathsf{val}\ E$

Proof
by induction on the typing derivation; by inspection of the last typing rule used. □

C.1.3 Store properties

Lemma C.6 (*Store Weakening*)
If $\Sigma' \supseteq \Sigma$ and Σ' well-formed, and

- if $\Sigma; \Gamma \vdash M : A$ then $\Sigma'; \Gamma \vdash M : A$
- if $\Sigma; \Gamma \vdash E \div_o C$ then $\Sigma'; \Gamma \vdash E \div_o C$

Proof
by simultaneous induction on the given derivations. By cases on the last rule used.

- Case
$$\frac{}{\Sigma; \Gamma \vdash \ell : \mathsf{ref}_{\mathsf{Level}(\ell)}\, \Sigma(\ell)} \quad (20)$$
  1. Since Σ' is well-formed, there is at most one occurrence of ℓ in Σ'
  2. Evidently $\ell \in \mathrm{dom}(\Sigma)$, therefore $\ell \in \mathrm{dom}(\Sigma')$.
  3. Since $\Sigma' \supseteq \Sigma$, $\Sigma'(\ell) = \Sigma(\ell)$.
  4. By rule (20), $\Sigma'; \Gamma \vdash \ell : \mathsf{ref}_{\mathsf{Level}(\ell)}\, \Sigma'(\ell)$.
- All the remaining cases are straightforward by IH.

□

Corollary C.7 (*Allocation Safety*)
If $\Sigma; \cdot \vdash V : A$, $\vdash H : \Sigma$ and if $\ell \notin \mathrm{dom}(H)$ then $\vdash H\{\ell \mapsto V\} : \Sigma\{\ell : A\}$

Proof
Directly. Using the Store Weakening lemma. □

Lemma C.8 (*Store Update*)
If $\vdash H : \Sigma$ and if $\ell \in \mathrm{dom}(\Sigma)$ and $\Sigma; \cdot \vdash V : \Sigma(\ell)$ then $\vdash H\{\ell \mapsto V\} : \Sigma$

Proof
Directly. □

C.1.4 Preservation, progress and type safety

Lemma C.9 (*Term Preservation*)
If $\Sigma; \cdot \vdash M : A$ and $M \to M'$ then $\Sigma; \cdot \vdash M' : A$

Proof
by induction on the evaluation relation. By cases on the last rule used. Since terms are pure, the proof is particularly straightforward. □

Preservation If $\vdash S \div_o A$ and $S \to S'$ then $\vdash S' \div_o A$

Proof
by induction on the evaluation relation.

By pattern matching, $S = (H, \Sigma, E)$, $S' = (H', \Sigma', E')$, $o = (r, w)$

By Inversion,

- $\vdash H : \Sigma$
- $\Sigma; \cdot \vdash E \div_o A$

Now proceed by cases on the last rule used in $S \to S'$. The proof is straightforward, using Inversion, Term Preservation, Store Weakening, Allocation Safety, and Store Update. □

Lemma C.10 (*Term Progress*)
If $\Sigma; \cdot \vdash M : A$ then either M is a value, or $\exists M'$ such that $M \to M'$

Proof
by induction on the given derivation. By cases on the last rule used. The proof is straightforward, using the Canonical Forms lemma. □

Progress If $\vdash S \div_o A$ then either S is terminal, or $\exists S'$ such that $S \to S'$

Proof
By pattern matching, $S = (H, \Sigma, E)$.

By Inversion, $\vdash H : \Sigma$, and $\Sigma; \cdot \vdash E \div_o A$.

Proceed by induction on the typing derivation, by cases on the last rule used. In each case the result is either immediate by IH, or follows from Term Progress, Canonical Forms and the IH. □

C.2 Structural properties of equivalence

We show that the judgments for $\approx_\zeta$ admit reflexivity (for well-typed computations), symmetry, and transitivity rules, that is they are equivalence relations on well-typed computation states.

Lemma C.11 (*Reflexivity*)

1. If $\Sigma; \Gamma \vdash M : A$ then $\Sigma; \Sigma; \Gamma \vdash M \approx_\zeta M : A$.
2. If $\Sigma; \Sigma; \Gamma \vdash E \div_o C$ then $\Sigma; \Sigma; \Gamma \vdash E \approx_\zeta E \div_o C$
3. If $\vdash H : \Sigma$ then $\vdash (H : \Sigma) \approx_\zeta^U (H : \Sigma)$ for all $U \subseteq \mathrm{dom}(H)$
4. If $\vdash S \div_o C$ then $\vdash S \approx_\zeta S \div_o C$

Proof
Parts (1) and (2) simultaneously by induction on the given derivation, by cases on the last rule used. Parts (3) and (4) follow by inversion on the single rule for the given derivation, and then using parts (1) and (2).

In part (1), the case of store locations l is not immediate. There are two cases depending on whether $\mathsf{Level}(\ell)$ is below ζ or not. When ℓ is low-security, the result is straightforward. Otherwise, note that store locations are values and that since $\mathsf{Level}(\ell) \not\sqsubseteq \zeta$, $\vdash \mathsf{ref}_{\mathsf{Level}(\ell)}\, \Sigma(\ell) \nearrow \mathsf{Level}(\ell)$, and the result follows using rule (39). The remaining cases follow by induction. □

Lemma C.12 (*Symmetry*)

1. If $\Sigma_1;\Sigma_2;\Gamma \vdash M_1 \approx_\zeta M_2 : A$ then $\Sigma_2;\Sigma_1;\Gamma \vdash M_2 \approx_\zeta M_1 : A$.
2. If $\Sigma_1;\Sigma_2;\Gamma \vdash E_1 \approx_\zeta E_2 \div_o C$ then $\Sigma_2;\Sigma_1;\Gamma \vdash E_2 \approx_\zeta E_1 \div_o C$
3. If $\vdash (H_1 : \Sigma_1) \approx_\zeta^U (H_2 : \Sigma_2)$ then $\vdash (H_2 : \Sigma_2) \approx_\zeta^U (H_1 : \Sigma_1)$
4. If $\vdash S_1 \approx_\zeta S_2 \div_o C$ then $\vdash S_2 \approx_\zeta S_1 \div_o C$

Proof
by induction on derivations. Evident as all the judgments are symmetric. □

Lemma C.13 (*Transitivity*)
Four parts:

1. If $\Sigma_1;\Sigma_2;\Gamma \vdash M_1 \approx_\zeta M_2 : A$ and $\Sigma_2;\Sigma_3;\Gamma \vdash M_2 \approx_\zeta M_3 : A$ then $\Sigma_1;\Sigma_3;\Gamma \vdash M_1 \approx_\zeta M_3 : A$
2. If $\Sigma_1;\Sigma_2;\Gamma \vdash E_1 \approx_\zeta E_2 \div_o C$ and $\Sigma_2;\Sigma_3;\Gamma \vdash E_2 \approx_\zeta E_3 \div_o C$ then $\Sigma_1;\Sigma_1;\Gamma \vdash E_1 \approx_\zeta E_3 \div_o C$
3. If $\vdash (H_1 : \Sigma_1) \approx_\zeta^U (H_2 : \Sigma_2)$ and $\vdash (H_2 : \Sigma_2) \approx_\zeta^U (H_3 : \Sigma_3)$ then $\vdash (H_1 : \Sigma_1) \approx_\zeta^U (H_3 : \Sigma_3)$
4. If $\vdash S_1 \approx_\zeta S_2 \div_o C$ and $\vdash S_2 \approx_\zeta S_3 \div_o C$ then $\vdash S_1 \approx_\zeta S_3 \div_o C$

Proof
Parts (1) and (2) follow by simultaneous induction on derivations.
Part (3):

1. By Inversion on each given derivation, $\vdash H_i : \Sigma_i$ for $i = 1,2,3$, $\Sigma_1 \upharpoonright U = \Sigma_2 \upharpoonright U = \Sigma_3 \upharpoonright U$, and for each $\ell \in U$, $\Sigma_1;\Sigma_2;\cdot \vdash H_1(\ell) \approx_\zeta H_2(\ell) : \Sigma_1(\ell)$ and $\Sigma_2;\Sigma_3;\cdot \vdash H_2(\ell) \approx_\zeta H_3(\ell) : \Sigma_2(\ell)$
2. By Part (1), for each $\ell \in U$, $\Sigma_1;\Sigma_3;\cdot \vdash H_1(\ell) \approx_\zeta H_3(\ell) : \Sigma_1(\ell)$
3. By rule (58), $\vdash (H_1 : \Sigma_1) \approx_\zeta^U (H_3 : \Sigma_3)$

Part (4):

1. By pattern matching, $S_i = (H_i, \Sigma_i, E_i)$ for $i = 1,2,3$
2. By Inversion, $\vdash (H_1 : \Sigma_1)) \approx_\zeta^{U_{12}} (H_2 : \Sigma_2)$, $\Sigma_1;\Sigma_2;\cdot \vdash E_1 \approx_\zeta E_2 \div_o C$ where $U_{12} = \mathrm{dom}(\Sigma_1) \cap \mathrm{dom}(\Sigma_2) \cap \downarrow(\zeta)$
3. By Inversion, $\vdash (H_2 : \Sigma_2)) \approx_\zeta^{U_{23}} (H_3 : \Sigma_3)$, $\Sigma_2;\Sigma_3;\cdot \vdash E_2 \approx_\zeta E_3 \div_o C$ where $U_{23} = \mathrm{dom}(\Sigma_2) \cap \mathrm{dom}(\Sigma_3) \cap \downarrow(\zeta)$
4. Let $U_{13} = \mathrm{dom}(\Sigma_1) \cap \mathrm{dom}(\Sigma_3) \cap \downarrow(\zeta)$
5. Suppose $\ell \in U_{13} \setminus (\mathrm{dom}(\Sigma_2) \cap \downarrow(\zeta))$
   (a) Evidently, $\ell \notin U_{12}$ and $\ell \notin U_{23}$
   (b) Choose $\ell' \notin U_{13} \cup \mathrm{dom}(\Sigma_2)$ such that $\mathsf{Level}(\ell') = \mathsf{Level}(\ell)$
   (c) α-vary (H_3, Σ_3, E_3) with ℓ' for ℓ
6. So for all $\ell \in U_{13}$, $\ell \in \mathrm{dom}(\Sigma_2) \cap \downarrow(\zeta)$
7. Evidently, $U_{13} \subseteq U_{12}$ and $U_{13} \subseteq U_{23}$
8. By Store Equivalence Coarsening, $\vdash (H_1 : \Sigma_1) \approx_\zeta^{U_{13}} (H_2 : \Sigma_2)$, and $\vdash (H_2 : \Sigma_2) \approx_\zeta^{U_{13}} (H_3 : \Sigma_3)$
9. By Part (3), $\vdash (H_1 : \Sigma_1) \approx_\zeta^{U_{13}} (H_3 : \Sigma_3)$

10. By Part (2), $\Sigma_1;\Sigma_3;\cdot \vdash E_1 \approx_\zeta E_3 \div_o C$
11. By rule (59), $\vdash S_1 \approx_\zeta S_3 \div_o C$

□

Lemma C.14 (*Regularity of Equivalence*)
Four parts:

1. If $\Sigma_1;\Sigma_2;\Gamma \vdash M_1 \approx_\zeta M_2 : A$ then $\Sigma_i;\Gamma \vdash M_i : A$
2. If $\Sigma_1;\Sigma_2;\Gamma \vdash E_1 \approx_\zeta E_2 \div_o C$ then $\Sigma_i;\Gamma \vdash E_i \div_o C$
3. If $\vdash (H_1 : \Sigma_1) \approx^U_\zeta (H_2 : \Sigma_2)$ then $\vdash H_i : \Sigma_i$
4. If $\vdash S_1 \approx_\zeta S_2 \div_o C$ then $\vdash S_i \div_o C$

Proof
by induction on the derivations. □

Next, we establish inversion and functionality. Inversion will let us reason by cases in subsequent proof. Functionality is the analog of a substitution for the equivalence judgment.

Lemma C.15 (*Equivalent Term Inversion*)
If $\Sigma_1;\Sigma_2;\Gamma \vdash M_1 \approx_\zeta M_2 : A$ then either

there exists a B, such that $\vdash B \leqslant A$ and $\vdash B \nearrow a$ and $a \not\sqsubseteq \zeta$ and M_1 and M_2 are values and $\Sigma_i;\Gamma \vdash M_i : B$ for $i = 1, 2$,

or

1. if $M_1 = x$ then $\vdash \Gamma(x) \leqslant A$ and $M_2 = x$.
2. if $M_1 = *$ then $\vdash 1 \leqslant A$ and $M_2 = *$.
3. if $M_1 = \mathsf{true}$ then $\vdash \mathsf{bool} \leqslant A$ and $M_2 = \mathsf{true}$.
4. if $M_1 = \mathsf{false}$ then $\vdash \mathsf{bool} \leqslant A$ and $M_2 = \mathsf{false}$.
5. if $M_1 = \mathsf{if}\ N_1\ \mathsf{then}\ P_{11}\ \mathsf{else}\ P_{12}$ then $M_2 = \mathsf{if}\ N_2\ \mathsf{then}\ P_{21}\ \mathsf{else}\ P_{22}$ and $\Sigma_1;\Sigma_2;\Gamma \vdash N_1 \approx_\zeta N_2 : \mathsf{bool}$ and $\Sigma_1;\Sigma_2;\Gamma \vdash P_{11} \approx_\zeta P_{21} : B$ and $\Sigma_1;\Sigma_2;\Gamma \vdash P_{12} \approx_\zeta P_{22} : B'$ and $\vdash B \leqslant A, \vdash B' \leqslant A$
6. if $M_1 = \ell$ then $\vdash \mathsf{ref}_b\, B \leqslant A$ and $M_2 = \ell$ and $b \sqsubseteq \zeta$ and $\Sigma_i(\ell) = B$ for $i = 1, 2$ and $\mathsf{Level}(\ell) = b$
7. if $M_1 = \lambda x : B.N_1$ then $\vdash B \to C \leqslant A$ and $M_2 = \lambda x : B.N_2$ and $\Sigma_1;\Sigma_2;\Gamma, x : B \vdash N_1 \approx_\zeta N_2 : C$
8. if $M_1 = \mathsf{val}\ E_1$ then $\vdash \bigcirc_o B \leqslant A$ and $M_2 = \mathsf{val}\ E_2$ and $\Sigma_1;\Sigma_2;\Gamma \vdash E_1 \approx_\zeta E_2 \div_o B$
9. if $M_1 = N_1 P_1$ then $M_2 = N_2 P_2$ and $\Sigma_1;\Sigma_2;\Gamma \vdash N_1 \approx_\zeta N_2 : B \to C$ and $\Sigma_1;\Sigma_2;\Gamma \vdash P_1 \approx_\zeta P_2 : B$ and $\vdash C \leqslant A$

Proof
by induction on the derivation. □

Lemma C.16 (*Equivalent Expression Inversion*)
If $\Sigma_1;\Sigma_2;\Gamma \vdash E_1 \approx_\zeta E_2 \div_o A$ then

1. if $E_1 = [M_1]$ then $E_2 = [M_2]$ and $\Sigma_1;\Sigma_2;\Gamma \vdash M_1 \approx_\zeta M_2 : A$

Proof
by induction on the given derivation. By cases on the last rule used. The proof is straightforward. □

Lemma C.17 (*Functionality*)
If $\Sigma_1;\Sigma_2;\Gamma,\Gamma' \vdash M_1 \approx_\zeta M_2 : A$ then

1. if $\Sigma_1;\Sigma_2;\Gamma,x : A,\Gamma' \vdash N_1 \approx_\zeta N_2 : C$ then $\Sigma_1;\Sigma_2;\Gamma,\Gamma' \vdash N_1[M_1/x] \approx_\zeta N_2[M_2/x] : C$
2. if $\Sigma_1;\Sigma_2;\Gamma,x : A,\Gamma' \vdash E_1 \approx_\zeta E_2 \div_o C$ then $\Sigma_1;\Sigma_2;\Gamma,\Gamma' \vdash E_1[M_1/x] \approx_\zeta E_2[M_2/x] \div_o C$.

Proof
by induction on the TD. □

Although we established Functionality for arbitrary terms to be substituted for x, as befits a call by value language, we only substitute values in the proof of non-interference.

Lemma C.18 (*Store Equivalence Coarsening*)
If $\vdash (H_1 : \Sigma_1) \approx_\zeta^{U'} (H_2 : \Sigma_2)$ and $U \subseteq U'$ then $\vdash (H_1 : \Sigma_1) \approx_\zeta^{U} (H_2 : \Sigma_2)$

Proof

1. By Inversion, $\vdash H_i : \Sigma_i$ for $i = 1,2$, $\Sigma_1 \restriction U' = \Sigma_2 \restriction U'$, for each $\ell \in U'$, $\Sigma_1;\Sigma_2;\cdot \vdash H_1(\ell) \approx_\zeta H_2(\ell) : \Sigma_1(\ell)$
2. Evidently, $\Sigma_1 \restriction U = \Sigma_2 \restriction U$
3. Evidently, for each $\ell \in U$, $\Sigma_1;\Sigma_2;\cdot \vdash H_1(\ell) \approx_\zeta H_2(\ell) : \Sigma_1(\ell)$
4. By rule (58), $\vdash (H_1 : \Sigma_1) \approx_\zeta^{U} (H_2 : \Sigma_2)$

□

Lemma C.19 (*Equivalent Values*)
If $\Sigma_1;\Sigma_2;\cdot \vdash M_1 \approx_\zeta M_2 : A$ then M_1 is a value if and only if M_2 is a value.

Proof
by induction on the equivalence derivation. By cases on the last rule used. The proof is straightforward. The restriction to values in rule (39) greatly simplifies matters. □

With the Equivalent Values lemma in hand, we can establish the Hexagon Lemma for terms.

C.3 Term Hexagon lemma proof

Term Hexagon Lemma For all ζ, if $\Sigma_1;\Sigma_2;\cdot \vdash M_1 \approx_\zeta M_2 : A$ and $M_1 \to M_1'$ and $M_2 \to M_2'$ and $M_1' \downarrow$ and $M_2' \downarrow$, then there exist M_1'', M_2'' such that $M_1' \to^* M_1''$, $M_2' \to^* M_2''$, $\Sigma_1;\Sigma_2;\cdot \vdash M_1'' \approx_\zeta M_2'' : A$

Proof
by induction on the given derivation. By cases on the last rule used.

- Cases rules (39), (40), (42), (43), (45), (47), (48). Vacuous, M_1, M_2 are values, no applicable evaluation rules.
- Case rule (41). Vacuous, $\Gamma = \cdot$
- Case rule (49). By IH.
- Case rule (44):

$$\frac{\Sigma_1;\Sigma_2;\cdot \vdash N_1 \approx_\zeta N_2 : \mathsf{bool} \quad \Sigma_1;\Sigma_2;\cdot \vdash P_{11} \approx_\zeta P_{21} : A \quad \Sigma_1;\Sigma_2;\cdot \vdash P_{12} \approx_\zeta P_{22} : A}{\Sigma_1;\Sigma_2;\cdot \vdash \mathsf{if}\ N_1\ \mathsf{then}\ P_{11}\ \mathsf{else}\ P_{12} \approx_\zeta \mathsf{if}\ N_2\ \mathsf{then}\ P_{21}\ \mathsf{else}\ P_{22} : A} \quad (44)$$

  By pattern matching, $M_i = \mathsf{if}\ N_i\ \mathsf{then}\ P_{i1}\ \mathsf{else}\ P_{i2}$ for $i = 1, 2$
  There are three possible evaluation rules for $M_1 \to M_1'$
  — Case IF1: $M_1' = \mathsf{if}\ N_1'\ \mathsf{then}\ P_{11}\ \mathsf{else}\ P_{12}$, $N_1 \to N_1'$
    1. By Equivalent Values, N_2 is not a value
    2. The only applicable evaluation rule for $M_2 \to M_2'$ is IF1: $M_2' = \mathsf{if}\ N_2'\ \mathsf{then}\ P_{21}\ \mathsf{else}\ P_{22}$, $N_2 \to N_2'$
    3. By Subterm Termination, $N_1' \downarrow$, $N_2' \downarrow$
    4. By IH, there exist N_1'', N_2'' such that $N_i' \to^* N_i''$ for $i = 1, 2$, and $\Sigma_1;\Sigma_2;\cdot \vdash N_1'' \approx_\zeta N_2'' : \mathsf{bool}$
    5. By repeated application of IF1, $M_i' \to^* M_i''$ for $i = 1, 2$
    6. By rule (44), $\Sigma_1;\Sigma_2;\cdot \vdash M_1'' \approx_\zeta M_2'' : A$
  — Case IFTRUE: $N_1 = \mathsf{true}$, $M_1' = P_{11}$
    1. By Equivalent Values, N_2 is a value
    2. By Equivalent Term Inversion, there are two subcases:
       – Either there exists a B such that $\vdash B \leqslant \mathsf{bool}$, $\vdash B \nearrow a$, $a \not\sqsubseteq \zeta$ and $\Sigma_i;\cdot \vdash N_i : B$
         By subtyping inversion, $B = \mathsf{bool}$. By Informativeness Inversion, $a = \bot$, for a contradiction (since $\bot \sqsubseteq \zeta$)
       – Or $N_2 = \mathsf{true}$
         (a) There is a single applicable evaluation rule for $M_2 \to M_2'$, IFTRUE: $M_2' = P_{21}$.
         (b) Let $M_i'' = M_i'$ for $i = 1, 2$.
         (c) Evidently, $\Sigma_1;\Sigma_2;\cdot \vdash M_1'' \approx_\zeta M_2'' : A$
  — Case IFFALSE: $N_1 = \mathsf{false}$, $M_2' = P_{12}$
    Similar to previous case.
- Case rule (46):

$$\frac{\Sigma_1;\Sigma_2;\cdot \vdash N_1 \approx_\zeta N_2 : B \to A \quad \Sigma_1;\Sigma_2;\cdot \vdash P_1 \approx_\zeta P_2 : B}{\Sigma_1;\Sigma_2;\cdot \vdash N_1 P_1 \approx_\zeta N_2 P_2 : A} \quad (46)$$

  Similar to the previous case.

□

C.4 High security step

To show the HSS Lemma 5.2, we need to show that after executing a high security expression, the resulting store is equivalent to the original store. We first show this for one evaluation step, and then extend to multiple steps.

Lemma C.20 (*Single High Security Step*)
If $\vdash (H, \Sigma, E) \div_o A$, $o = (r, w)$ and $w \not\sqsubseteq \zeta$, and $(H, \Sigma, E) \to (H', \Sigma', E')$ then $\vdash (H : \Sigma) \approx_\zeta^{\mathrm{dom}(\Sigma) \cap \downarrow(\zeta)} (H' : \Sigma')$.

Proof
By induction on $(H, \Sigma, E) \to (H', \Sigma', E')$
By inversion on $\vdash (H, \Sigma, E) \div_o A$, we have

- $\vdash H : \Sigma$
- $\Sigma; \cdot \vdash E \div_o A$

Now consider cases on the evaluation rule used:

- Case RET1: $E = [M]$, $H' = H$, $\Sigma' = \Sigma$, $E' = [N]$, $M \to N$
  By Reflexivity, $\vdash (H : \Sigma) \approx_\zeta^U (H' : \Sigma')$ where $U = \mathrm{dom}(\Sigma) \cap \downarrow(\zeta)$
  Cases for LETVAL1, REF1, BANG1, ASSN1, ASSN2, BANG, and LETVAL are similar.
- Case LETVALVAL: $E = \mathsf{let\ val}\ x = \mathsf{val}\ E_1\ \mathsf{in}\ F$, $E' = \mathsf{let\ val}\ x = \mathsf{val}\ E_2\ \mathsf{in}\ F$, $(H, \Sigma, E_1) \to (H', \Sigma', E_2)$
  1. By Inversion, for some $o' = (r', w')$, $\Sigma; \cdot \vdash \mathsf{val}\ E_1 : \bigcirc_{o'} B$, $\Sigma; x : B \vdash F \div_{o'} C$, $\vdash C \leqslant A$, where either $o' \preceq o$ or both $\vdash C \nearrow r'$ and $(\bot, w') \preceq o$
  2. Either way, $w \sqsubseteq w'$, so $w' \not\sqsubseteq \zeta$
  3. By Inversion, $\Sigma; \cdot \vdash E_1 \div_{o''} B'$ and $\vdash \bigcirc_{o''} B' \leqslant \bigcirc_{o'} B$
  4. By rule (38), $\vdash (H, \Sigma, E_1) \div_{o''} B'$
  5. By IH, $\vdash (H : \Sigma) \approx_\zeta^U (H' : \Sigma')$ where $U = \mathrm{dom}(\Sigma) \cap \downarrow(\zeta)$
- Case REF: $E = \mathsf{ref}_a\,(V : B)$, $H' = H\{\ell \mapsto V\}$, $\Sigma' = \Sigma\{\ell : B\}$, $E' = \ell$, $\ell \notin \mathrm{dom}(H)$, $\mathsf{Level}(\ell) = a$
  1. By Inversion, $\Sigma; \cdot \vdash V : B$, $\vdash \mathsf{ref}_a\, B \leqslant A$
  2. By rule (37), $\vdash H' : \Sigma'$
  3. Consider $\ell' \in U$, by construction, $H'(\ell') = H(\ell')$ and $\Sigma'(\ell') = \Sigma(\ell')$
  4. By rule (58) $\vdash (H : \Sigma) \approx_\zeta^U (H' : \Sigma')$
- Case ASSN: $E = \ell := V$, $H' = H\{\ell \mapsto V\}$, $\Sigma' = \Sigma$, $E' = [*]$
  1. By Inversion, $\Sigma; \cdot \vdash \ell : \mathsf{refw}_a\, B$, $\Sigma; \cdot \vdash V : B$, $(\bot, a) \preceq o$, $\vdash 1 \leqslant A$
  2. By Inversion, $\vdash \mathsf{ref}_{\mathsf{Level}(\ell)}\, \Sigma(\ell) \leqslant \mathsf{refw}_a\, B$
  3. By Subtyping Inversion, $\vdash B \leqslant \Sigma(\ell)$, $a \sqsubseteq \mathsf{Level}(\ell)$
  4. Since $(\bot, a) \preceq o$, $w \sqsubseteq a \sqsubseteq \mathsf{Level}(\ell)$
  5. Since $w \not\sqsubseteq \zeta$, $\mathsf{Level}(\ell) \not\sqsubseteq \zeta$, so $\ell \notin U$ where $U = \mathrm{dom}(\Sigma) \cap \downarrow(\zeta)$
  6. By rule (37), $\vdash H' : \Sigma'$
  7. By rule (58), $\vdash (H : \Sigma) \approx_\zeta^U (H' : \Sigma')$

□

Corollary C.21
If $\vdash (H, \Sigma, E) \div_o A$, $o = (r, w)$ and $w \not\sqsubseteq \zeta$, and $(H, \Sigma, E) \rightarrow^n (H', \Sigma', E')$ then $\vdash (H : \Sigma) \approx_\zeta^{\mathrm{dom}(\Sigma) \cap \downarrow(\zeta)} (H' : \Sigma')$.

Proof
By induction on n, the number of steps.

By inversion,

- $\vdash H : \Sigma$
- $\Sigma; \cdot \vdash E \div_o A$

If $n = 0$, the result follows by Reflexivity.

If $n > 0$, then $(H, \Sigma, E) \rightarrow (H'', \Sigma'', E'') \rightarrow^{n-1} (H', \Sigma', E')$. The result follows by IH, Single High Security Step, Preservation, Store Equivalence Coarsening and transitivity of the store equivalence judgment. □

D $\lambda_{\mathrm{SEC}}^{\mathrm{REF}}$ well-typed translation proof

Well-typed Translation

1. If $\Sigma; \Gamma \vdash bv : t \Rightarrow M$ then $\overline{\Sigma}; \overline{\Gamma} \vdash M : \overline{t}$
2. If $\Sigma; \Gamma[\mathsf{pc}] \vdash e : s \Rightarrow E$ then $\overline{\Sigma}; \overline{\Gamma} \vdash E \div_{(\bot, \mathsf{pc})} \overline{s}$

Proof
Both parts simultaneously, by induction on the given derivations. By cases on the last rule used.

Part (1)

- The cases for unit and boolean values, and store locations are immediate.
- Case

$$\frac{\Sigma; \Gamma, x : s_1[\mathsf{pc}] \vdash e : s_2 \Rightarrow E}{\Sigma; \Gamma \vdash \lambda[\mathsf{pc}] x : s_1.e : s_1 \xrightarrow{\mathsf{pc}} s_2 \Rightarrow \lambda x : \overline{s_1}.\mathsf{val}\ E}$$

  1. By IH,

$$\overline{\Sigma}; \overline{\Gamma}, x : \overline{s_1} \vdash E \div_{(\bot, \mathsf{pc})} \overline{s_2}$$

  2.

$$\frac{\dfrac{\overline{\Sigma}; \overline{\Gamma}, x : \overline{s_1} \vdash E \div_{(\bot, \mathsf{pc})} \overline{s_2}}{\overline{\Sigma}; \overline{\Gamma}, x : \overline{s_1} \vdash \mathsf{val}\ E : \bigcirc_{(\bot, \mathsf{pc})} \overline{s_2}}}{\overline{\Sigma}; \overline{\Gamma} \vdash \lambda x : \overline{s_1}.\mathsf{val}\ E : \overline{s_1 \xrightarrow{\mathsf{pc}} s_2}}$$

- The case for subsumption follows by well-typed type translation

Part (2)

- Case

$$\frac{\Sigma; \Gamma[\mathsf{pc}] \vdash e_1 : (\mathsf{bool}, a) \Rightarrow E_1 \quad \Sigma; \Gamma[\mathsf{pc} \sqcup a] \vdash e_2 : s \Rightarrow E_2 \quad \Sigma; \Gamma[\mathsf{pc} \sqcup a] \vdash e_3 : s \Rightarrow E_3}{\Sigma; \Gamma[\mathsf{pc}] \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : s \Rightarrow \begin{array}{l} \mathsf{let}\ y = E_1\ \mathsf{in} \\ \mathsf{let}\ y' = !y\ \mathsf{in} \\ \quad \mathsf{run}\quad \mathsf{if}\ y' \\ \qquad\quad \mathsf{then}\ \mathsf{val}\ E_2 \\ \qquad\quad \mathsf{else}\ \mathsf{val}\ E_3 \end{array}}$$

1. By IH, $\overline{\Sigma};\overline{\Gamma} \vdash E_1 \div_{(\bot,\mathsf{pc})} \mathsf{refr}_a\,\mathsf{bool}$, and $\overline{\Sigma};\overline{\Gamma} \vdash E_i \div_{(\bot,\mathsf{pc}\sqcup a)} \overline{s}$ for $i = 2,3$
2. Let $\Gamma_1 = \overline{\Gamma}, y : \mathsf{refr}_a\,\mathsf{bool}$
3. By rule (32), $\overline{\Sigma};\Gamma_1 \vdash !y \div_{(a,\top)} \mathsf{bool}$
4. Let $\Gamma_2 = \Gamma_1, y' : \mathsf{bool}$
5. $$\dfrac{\dfrac{\overline{\Sigma};\Gamma_2 \vdash y' : \mathsf{bool}}{}\,(19) \quad \dfrac{\overline{\Sigma};\Gamma_2 \vdash \mathsf{val}\ E_i : \bigcirc_{(\bot,\mathsf{pc}\sqcup a)}\overline{s}}{}\,(27)\ \text{for } i = 2,3}{\dfrac{\overline{\Sigma};\Gamma_2 \vdash \mathsf{if}\ y'\ \mathsf{then\ val}\ E_2\ \mathsf{else\ val}\ E_3 : \bigcirc_{(\bot,\mathsf{pc}\sqcup a)}\overline{s}}{\overline{\Sigma};\Gamma_2 \vdash \mathsf{run\ if}\ y'\ \mathsf{then\ val}\ E_2\ \mathsf{else\ val}\ E_3 \div_{(\bot,\mathsf{pc}\sqcup a)} \overline{s}}\,(24)}$$
6. We can promote the operation levels of $!y$ and $\mathsf{run}\ \ldots$ to $(a, \mathsf{pc} \sqcup a)$, such that
$$\overline{\Sigma};\Gamma_1 \vdash \mathsf{let}\ y' = !y\ \mathsf{in\ run}\ \ldots \div_{(a,\mathsf{pc}\sqcup a)} \overline{s}$$
7. Let $(t,b) = s$, and note that $\overline{s} = \mathsf{refr}_b\,\overline{t}$.
8. By lemma 6.1, $a \sqsubseteq b$. Hence $\vdash \overline{s} \nearrow a$.
9. Therefore,
$$\dfrac{\dfrac{\overline{\Sigma};\Gamma_1 \vdash \mathsf{let}\ y' = !y\ \mathsf{in\ run}\ \ldots \div_{(a,\mathsf{pc}\sqcup a)} \overline{s} \quad \vdash \overline{s} \nearrow a}{\overline{\Sigma};\Gamma_1 \vdash \mathsf{let}\ y' = !y\ \mathsf{in\ run}\ \ldots \div_{(\bot,\mathsf{pc}\sqcup a)} \overline{s}}\,(35)}{\overline{\Sigma};\Gamma_1 \vdash \mathsf{let}\ y' = !y\ \mathsf{in\ run}\ \ldots \div_{(\bot,\mathsf{pc})} \overline{s}}\,(34)$$
10. Therefore,
$$\overline{\Sigma};\overline{\Gamma} \vdash \mathsf{let}\ y = E_1\ \mathsf{in\ let}\ y' = !y\ \mathsf{in\ run}\ \ldots \div_{(\bot,\mathsf{pc})} \overline{s}$$

- Case
$$\dfrac{\Sigma;\Gamma[\mathsf{pc}] \vdash e_1 : (\mathsf{ref}\ (t,b), a) \Rightarrow E_1 \quad \Sigma;\Gamma[\mathsf{pc}] \vdash e_2 : (t,b) \Rightarrow E_2 \quad a \sqsubseteq b}{\Sigma;\Gamma[\mathsf{pc}] \vdash e_1 := e_2 : (1,\mathsf{pc}) \Rightarrow \begin{array}{l}\mathsf{let}\ y_1 = E_1\ \mathsf{in}\\ \mathsf{let}\ y_2 = E_2\ \mathsf{in}\\ \mathsf{let}\ y_1' = !y_1\ \mathsf{in}\\ \mathsf{let}\ \_ = y_1' := y_2\ \mathsf{in}\\ \mathsf{ref}_{\mathsf{pc}}\,(* : 1)\end{array}}$$
1. By IH, $\overline{\Sigma};\overline{\Gamma} \vdash E_1 \div_{(\bot,\mathsf{pc})} \mathsf{refr}_a\,\mathsf{ref}_b\,\mathsf{refr}_b\,\overline{t}$ and $\overline{\Sigma};\overline{\Gamma} \vdash E_2 \div_{(\bot,\mathsf{pc})} \mathsf{refr}_b\,\overline{t}$
2. By Lemma 6.1, $\mathsf{pc} \sqsubseteq b$
3. Let $\Gamma_1 = \overline{\Gamma}, y_1 : \mathsf{refr}_a\,\mathsf{ref}_b\,\mathsf{refr}_b\,\overline{t}, y_2 : \mathsf{refr}_b\,\overline{t}$
4. $$\dfrac{\overline{\Sigma};\Gamma_1 \vdash !y_1 \div_{(a,\top)} \mathsf{ref}_b\,\mathsf{refr}_b\,\overline{t} \quad (a,\top) \preceq (a,b)}{\overline{\Sigma};\Gamma_1 \vdash !y_1 \div_{(a,b)} \mathsf{ref}_b\,\mathsf{refr}_b\,\overline{t}}$$
Note that (a,b) is a well-formed operation level since $a \sqsubseteq b$
5. Let $\Gamma_2 = \Gamma_1, y_1' : \mathsf{ref}_b\,\mathsf{refr}_b\,\overline{t}$
$$\dfrac{\dfrac{\overline{\Sigma};\Gamma_2 \vdash y_1' := y_2 \div_{(\bot,b)} 1 \quad \overline{\Sigma};\Gamma_2 \vdash \mathsf{ref}_{\mathsf{pc}}\,(* : 1) \div_{(\bot,\top)} \mathsf{refr}_{\mathsf{pc}}\,1}{\overline{\Sigma};\Gamma_2 \vdash \mathsf{let}\ \_ = y_1' := y_2\ \mathsf{in}\ \mathsf{ref}_{\mathsf{pc}}\,(* : 1) \div_{(\bot,b)} \mathsf{refr}_{\mathsf{pc}}\,1} \quad (\bot,b) \preceq (a,b)}{\overline{\Sigma};\Gamma_2 \vdash \mathsf{let}\ \_ = y_1' := y_2\ \mathsf{in}\ \mathsf{ref}_{\mathsf{pc}}\,(* : 1) \div_{(a,b)} \mathsf{refr}_{\mathsf{pc}}\,1}$$

6.

$$\dfrac{\dfrac{\overline{\Sigma};\Gamma_1 \vdash !y_1 \div_{(a,b)} \mathsf{ref}_b\, \mathsf{refr}_b\, \overline{t} \quad \overline{\Sigma};\Gamma_2 \vdash \mathsf{let}\ \_ = y_1' := y_2\ \mathsf{in}\ \mathsf{ref}_{\mathsf{pc}}\,(* : 1) \div_{(a,b)} \mathsf{refr}_{\mathsf{pc}}\, 1}{\overline{\Sigma};\Gamma_1 \vdash \begin{array}{l}\mathsf{let}\ y_1' = !y_1\ \mathsf{in}\\ \mathsf{let}\ \_ = y_1' := y_2\ \mathsf{in}\ \mathsf{ref}_{\mathsf{pc}}\,(* : 1) \div_{(a,b)} \mathsf{refr}_{\mathsf{pc}}\, 1\end{array}} \quad \dfrac{\vdash 1 \nearrow a}{\vdash \mathsf{refr}_{\mathsf{pc}}\, 1 \nearrow a}}{\overline{\Sigma};\Gamma_1 \vdash \begin{array}{l}\mathsf{let}\ y_1' = !y_1\ \mathsf{in}\\ \mathsf{let}\ \_ = y_1' := y_2\ \mathsf{in}\ \mathsf{ref}_{\mathsf{pc}}\,(* : 1) \div_{(\bot,b)} \mathsf{refr}_{\mathsf{pc}}\, 1\end{array}}$$

7. Since $\mathsf{pc} \sqsubseteq b$,

$$\overline{\Sigma};\overline{\Gamma} \vdash \mathsf{let}\ y_1 = E_1\ \mathsf{in}\ \mathsf{let}\ y_2 = E_2\ \mathsf{in}\ \mathsf{let}\ y_1' = !y_1\ \mathsf{in}\ \ldots \div_{(\bot,\mathsf{pc})} \mathsf{refr}_{\mathsf{pc}}\, 1$$

- Other cases are similar.

□

Acknowledgements

This material is based on work supported in part by NSF grants CCR-9984812 and CCR-0121633. Any opinions, findings, and conclusions or recommendations in this publication are those of the authors and do not reflect the views of this agency.

References

Abadi, M., Banerjee, A., Heintze, N. and Riecke, J. G. (1999) A core calculus of dependency. *Twenty-sixth ACM Symposium on Principles of Programming Languages*, pp. 147–160.

Harrison, W., Tullsen, M. and Hook, J. (2003) Domain separation by construction. *Foundations of Computer Security Workshop (FCS'03)*.

Heintze, N. and Riecke, J. G. (199) The SLam calculus: Programming with secerecy and integrity. *Twenty-fifth ACM Symposium on Principles of Programming Languages*, pp. 365–377.

Honda, K. and Yoshida, N. (2002) A uniform type structure for secure information flow. *Twenty-ninth ACM Symposium on Principles of Programming Languages*, pp. 81–92.

Huet, G. (1980) Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, **27**(4), 797–821.

Moggi, E. (1989) Computational lambda-calculus and monads. *Fourth IEEE Symposium on Logic in Computer Science*, pp. 14–23.

Moggi, E. (1991) Notions of computation and monads. *Infor. & Computation*, **93**, 55–92.

Morrisett, G., Walker, D., Crary, K. and Glew, N. (1999) From System F to typed assembly language. *ACM Trans. Program. Lang. & Syst.* **21**(3), 527–568.

Myers, A. C. (1999) JFlow: Practical mostly-static information flow control. *Twenty-sixth ACM Symposium on Principles of Programming Languages*, pp. 228–241.

Pfenning, F. and Davies, R. (2001) A judgmental reconstruction of modal logic. *Mathematical Struct. in Comput. Sci.* **11**(4), 511–540.

Pottier, F. and Simonet, V. (2003) Information flow inference for ML. *ACM Trans. Program. Lang. & Syst.* **25**(1), 117–158.

Sabelfeld, A. and Myers, A. C. (2003) Language-based information-flow security. *IEEE J. Selected Areas in Comm.* **21**(1), 5–19.

Smith, G. and Volpano, D. (1998) Secure information flow in a multi-threaded imperative language. *Twenty-fifth ACM Symposium on Principles of Programming Languages*, pp. 355–364.

Volpano, D., Smith, G. and Irvine, C. (1996) A sound type system for secure flow analysis. *J. Comput. Security*, **4**(3), 167–187.

Zdancewic, S. (2002) *Programming languages for information security*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York.

Zdancewic, S. (2003) A type system for robust declassification. *Nineteenth Mathematical Foundations of Programming Semantics*. Electronic Notes in Theoretical Computer Science.

Zdancewic, S. and Myers, A. C. (2001a) Robust declassification. *Fourteenth IEEE Computer Security Foundations Workshop*, pp. 15–23.

Zdancewic, S. and Myers, A. C. (2001b) Secure information flow and CPS. *Tenth European Symposium on Programming: Lecture Notes in Computer Science 2028*, pp. 46–61. Springer-Verlag.

Zdancewic, S. and Myers, A. C. (2002) Secure information flow via linear continuations. *Higher Order & symbolic Computation*, **15**(2–3), 209–234.