The SLam Calculus: Programming with Secrecy and Integrity

Nevin Heintze
Bell Laboratories
nch@bell-labs.com

Jon G. Riecke
Bell Laboratories
riecke@bell-labs.com

Abstract

The **SLam calculus** is a typed λ-calculus that maintains *security information* as well as type information. The type system propagates security information for each object in four forms: the object's creators and readers, and the object's *indirect* creators and readers (*i.e.*, those agents who, through flow-of-control or the actions of other agents, can influence or be influenced by the content of the object). We prove that the type system prevents security violations and give some examples of its power.

1 Introduction

How do we build a system that manipulates and stores information whose secrecy and integrity must be preserved? The information might, for example, contain employee salaries, tax information or social security numbers, or it might involve data whose integrity is essential to global system security, such as the UNIX (TM) `/etc/passwd` file or a database of public keys.

One solution is to provide a secure persistent store that controls access to each location in the store, *e.g.*, it might maintain access control lists that specify who may read and write each location. However, this only addresses part of the problem: it does not trace the security of information through *computation*, and hence is easy to defeat. For example, a privileged user might write a program that reads a secret location and copies it to an insecure location that everyone can read. Trust is central to the usability of this system. If you share some secrets with someone, then you must trust their intentions *and* their competence, since they may release secrets accidentally as a result of a programming error.

An alternative is to associate security with data objects instead of locations, and track security through computation at runtime. To do this, each object must come equipped with security information that specifies access rights, *i.e.*, a *capability*. However, this is not enough. We must also track the flow of information as we build new objects from old. For example, if we have a secret string and concatenate it with another string, then we must also treat the new string as secret. Such a scheme has two flaws. First, explicitly tracing security information through computation is very expensive. Second, the system must guarantee that security information is not forged. We can address these issues by *statically* approximating information flow, and by using a trusted run-time that only executes programs that have passed the static check. For example, we can view a program as a black box so that its output is at least as secret as each of its inputs. Similarly, we can view program output as having no higher integrity than each of its inputs. This approach of tracing information flow has been thoroughly explored in the security literature [3, 4, 5, 6, 7].

Unfortunately, in classic information flow systems, data quickly floats to the highest level of security. For example, consider a program that takes as input a string `x` representing a user id and another string `y` representing the user's password, and whose output is some object `z` built from `x` and `y`. Then the security level of the entire object `z` must be the security level appropriate for user passwords, even though the password information may be only a small component of `z`, or may not appear at all.

In this paper, we investigate how assumptions at the programming language level can address this limitation and provide fine grained control of security. We focus on the role of *strong typing*. We assume that all programs are compiled with a trusted compiler that enforces our type discipline, *i.e.*, there are no "back-doors" for inserting and executing unchecked code and there are no operations for direct access and

modification of raw memory locations. We also assume that the persistent store understands and preserves the types of objects. For example, in UNIX the `/etc/passwd` file is stored as a string; in our system, we shall implement and store this data structure as a list of records where each record contains, *e.g.*, a user name, user id, and password hash. By exposing this structure, we can attach different levels of security to the different components of an object, and thereby express security information more accurately and flexibly.

We present a core functional programming language called the *Secure Lambda Calculus* (or SLam calculus). The types of this language contain not only standard typing information, but also security information. This security information takes four forms: readers, creators, indirect readers and indirect creators. Intuitively, an agent must be a reader of an object in order to inspect the object's contents. An agent is a creator of any object it constructs. An agent is an indirect reader of an object if it may be influenced by the object's contents. An agent is an indirect creator of an object if it may influence the object's construction. For example, consider the statement

```
if (x > 25) then y := 1 else y := 2.
```

Here, partial information about `x` is available via variable `y`. If agent A can read `y` but not `x`, then A can still find out partial information about `x` and so A is an indirect reader of `x`. If the statement itself were executed by agent B, then B would require read access to `x`. B would also be a creator of `y`'s content. Moreover, if `x` was created by a third agent C, then C would be an indirect creator of `y`'s content.

Readers and indirect readers together specify an object's secrecy (who finds out about the object), whereas creators and indirect creators specify integrity (who is responsible for the object). Readers and creators together capture access control, while indirect readers and indirect creators capture information flow. In conjunction with higher-order functions and a rich underlying type structure (*e.g.*, records, sums), these four security forms provide a very flexible basis for controlled sharing and distribution of information. For example, higher-order functions can be used to build up complex capabilities. By specifying the set of readers and/or indirect readers of these functions—where the ability to read the function is the right to apply it—we can additionally restrict the capability so that it can only be shared by a specified group of agents. (Note that our assumptions ensure that all an agent can do with the function is apply it; in particular, there is no way to open up a function and gain access to its closure or other internals.)

Why is it convenient to have both access control and information flow in the type system? To illustrate the utility of having both, suppose we have just two security levels, `H` (high security) and `L` (low security), and a type called `users` that is a list containing strings whose direct readers are `H`, and whose indirect readers are `L`. If we ignore creators and indirect creators, and write direct readers before indirect readers, the type definition might look something like this in a Standard ML-like syntax [10]:

```
type users = (list (string,H,L),L,L)
```

Now suppose we want to look up names in a value of type `users`, and return `true` if the name is in the list. We might write the code as follows:

```
fun lookup ([]:users) name = false : (bool,L,L)
  | lookup ((x::rst):users) name =
    if x = name
       then true
       else lookup rst name
```

Our type system guarantees that only high-security agents can write the `lookup` function, or indeed any code that branches on the values of the strings in the list. Low-security agents can call the `lookup` function, but cannot get direct access to the strings held in the list. Information flows from the strings into the boolean output, but since we have labeled the *indirect* readers of the strings to be low security, the program is still type safe. If we had only indirect readers, the output of lookup would have to be a high-security boolean. More generally, if an agent is an indirect reader of an object but not a direct reader, then any information that agent finds out about that object must be via another agent who is a direct reader. Readers determine how much of an object is revealed to indirect readers, but they *cannot* reveal information to agents who are not indirect readers. At one extreme, a (direct) reader can reveal *all* information about an object to the indirect readers. We *trust* the object's readers to reveal only appropriate information about the object to indirect readers.

We present the SLam calculus in stages. In Section 2, we define the purely functional core of the SLam calculus, restricting the security properties to reader and indirect-reader security. The operational semantics explicitly checks for security errors. We prove that well-typed programs never cause security errors, and hence the checks may be omitted. Sections 3 and 4 extend the core calculus with assignments (using an effects-style extensions to the type system [17]), concurrency, and integrity. Section 5 concludes the paper with a discussion of other work and limitations of the system.

The type soundness theorems provide a direct proof that our type system enforces reader and creator security. However, the situation for indirect readers and indirect creators is less satisfactory. The operational semantics tracks this security information through computation, but this part of the definition is quite complex. What we seek is an independent confirmation of the soundness of our definitions. For example, if a global variable `x` has security information that says agent A is neither a reader nor indirect reader, then agent A's behavior should be independent of the value of `x`. No matter what value we give to `x`, A's behavior should not change. In the security literature, this property is called *noninterference* [18]. Borrowing ideas from Reynolds [15], we formalize this by using an equivalence relation to represent what agent A can know about `x` (in this case nothing, so the equivalence relation relates all values), and then check to see whether A's behavior respects this equivalence relation on `x`. We prove such a noninterference theorem for the core calculus in Section 2 using a denotational argument. This style of proof, while common in the languages literature, is novel to the security world. This proof technique seems to carry over to the extensions of the basic calculus. The notion of noninterference is problematic in a concurrent setting, a problem we discuss further in Section 3.

2 The Core Calculus

We illustrate the core ideas of our calculus using a language with functions, recursion, tuples, and sums, restricting the security properties to readers and indirect readers. We extend our treatment to a language with assignment and concurrency in Section 3, and to creators and indirect creators in Section 4.

2.1 Types and terms

The types of the SLam calculus essentially comprise those of a monomorphic type system—with products, sums, and functions—in which each type is annotated with security properties. We could add booleans, integers and strings, but the essential typing properties of these types are already covered by products and sums. Define **security properties** κ, **types** t, and **secure types** s by the grammar

$$\begin{array}{rcl} \kappa & ::= & (r, ir) \\ t & ::= & \mathbf{unit} \mid (t + t) \mid (t \times t) \mid (t \to t) \\ s & ::= & (t, \kappa) \end{array}$$

where r (readers) and ir (indirect readers) range over some collection of basic security descriptions. For example, a simple multi-level security system might have security descriptions L (low), M (medium) and H (high), with ordering $L \sqsubseteq M \sqsubseteq H$. Alternatively, a UNIX-like security system would begin with a collection of groups and users, and an ordering such that anything is less than `root`, and $g \sqsubseteq u$ if user u is in group g. In general, we assume that r and ir range over some collection S of basic security descriptions with ordering $\sqsubseteq$. We assume $(S, \sqsubseteq)$ is a lattice (*i.e.*, a partially ordered set with meets, joins, a top element $\top$ and bottom element $\bot$). Higher in the lattice means "more secure"; $\top$ is the most secure element. Intuitively, each element of S represents a set of agents or users; for this reason we refer to elements of S as **security groups**. For the purposes of presenting our static type system, we assume that S is static (each element represents a fixed set of agents). This is unrealistic because security changes over time (*e.g.*, new users and groups are added, users are added to and removed from groups). We discuss this further in Section 5.

We maintain the invariant that any security property (r, ir) satisfies $ir \sqsubseteq r$. In other words, r should be more restrictive (it represents a smaller set of agents) than ir. If a value v has security property (r, ir), then only the agents described by r may directly read a value; and only agents in ir may find out (partial) information about v. That is, r tracks access to an object, whereas ir tracks information flow.

The SLam calculus is a *call-by-value* language, and hence terms are contain a set of values that represent terminated computations. The sets of **basic values** and **values** are defined by the grammar

$$\begin{array}{rcl} bv & ::= & () \mid (\mathbf{inj}_i\ v) \mid \langle v, v\rangle \mid (\lambda x : s.\ e) \\ v & ::= & bv_\kappa \end{array}$$

The security properties on values describe which agents may read the object, and which agents may indirectly depend on the value. The terms of the SLam calculus are given by the grammar

$$\begin{array}{rcl} e & ::= & x \mid v \mid (\mathbf{inj}_i\ e)_\kappa \mid \langle e, e\rangle_\kappa \mid (e\ e)_r \mid (\mathbf{proj}_i\ e)_r \mid (\mu f : s.\ e) \mid (\mathbf{protect}_{ir}\ e) \mid \\ & & (\mathbf{case}\ e\ \mathbf{of}\ \mathbf{inj}_1(x) \Rightarrow e \mid \mathbf{inj}_2(x) \Rightarrow e)_r \end{array}$$

The term $(\mu f : s.\ e)$ defines a recursive function, and the term $(\mathbf{protect}_{ir}\ e)$ increases the security property of a term. Bound and free variables are defined in the usual way: variables may be bound by λ, μ, and **case**.

The security group r appearing on the destructors—application, projection, or case—represents the security group of the *programmer* of that code. It is the compiler's job to check that the annotations on programs are consistent with the author's security. For example, the compiler must prevent arbitrary users from writing programs with destructors annotated with `root`. As evaluation proceeds, terms with mixed annotations arise. Note that `root` can write programs that can be run by anyone, but once started, can access files and data structures as `root`. Such a "setuid" program would be a function

$$f = (\lambda x : (t, (\bot, \bot)).\ \textit{body involving}\ \texttt{root}\ \textit{annotations})_{(\bot,\bot)}.$$

Anyone can write an application $(f\ v)_r$, because $\bot \sqsubseteq r$, but the body runs at `root` and can access data that r cannot access. When the application $(f\ v)_r$ is reduced, the resultant body will have v substituted for x. If v is an abstraction, with destructors annotated r, the resultant body will mix r and `root` annotations.

2.2 Operational Semantics

The relation $e \to e'$ represents a single atomic action taken by an agent. The definition uses structured operational semantics [13] via **evaluation contexts** [8]. The set of evaluation contexts is given by

$$\begin{array}{rcl} E & ::= & [\cdot] \mid (E\ e)_r \mid (v\ E)_r \mid (\mathbf{proj}_i\ E)_r \mid (\mathbf{inj}_i\ E)_\kappa \mid \langle E, e\rangle_\kappa \mid \langle v, E\rangle_\kappa \mid (\mathbf{protect}_{ir}\ E) \mid \\ & & (\mathbf{case}\ E\ \mathbf{of}\ \mathbf{inj}_1(x) \Rightarrow e_1 \mid \mathbf{inj}_2(x) \Rightarrow e_2)_r \end{array}$$

Note that this defines a left-to-right, call-by-value, deterministic reduction strategy.

The basic rules for the operational semantics appear in Table 1. In the rules, we use an operation for increasing the security properties on terms: given $\kappa = (r, ir)$, $\kappa \bullet ir'$ is the security property $(r \sqcup ir', ir \sqcup ir')$. Abusing notation, we extend this operation to values: $bv_\kappa \bullet ir$ denotes the value $bv_{\kappa \bullet ir}$. These rules reduce simple redexes. The rules lift to arbitrary terms via the rule

$$\frac{e \to e'}{E[e] \to E[e']}$$

which shows how to reduce all terms *except* terms which have type or security errors. Note that the operational semantics is essentially *untyped*: the types on bound variables, upon which the type checking rules of the next section depend, are ignored during reduction. The security properties on values and destructors are, of course, checked during reduction; this corresponds to checking, for instance, that a pair is the value being taken apart by a projection. Note also that, after a value has been destructed, the *indirect* readers of the value are used to increase the secrecy of the result (via **protect**). This tracks information flow from the destructed value to the result.

2.3 Type System

The type system of the SLam calculus appears in Tables 2 and 3. The system includes subtyping and the subsumption rule. The subtyping rules in Table 2 start from lifting the $\sqsubseteq$ relation on security groups to the

Table 1: Operational Semantics.

$((\lambda x : s.\, e)_{r,ir}\ v)_{r'}$	$\rightarrow$	$(\mathbf{protect}_{ir}\ e[v/x])$	if $r \sqsubseteq r'$
$(\mathbf{proj}_i\ \langle v_1, v_2\rangle_{(r,ir)})_{r'}$	$\rightarrow$	$(\mathbf{protect}_{ir}\ v_i)$	if $r \sqsubseteq r'$
$(\mathbf{case}\ (\mathbf{inj}_j\ v)_{(r,ir)}\ \mathbf{of}\ \mathbf{inj}_1(x) \Rightarrow e_1 \mid \mathbf{inj}_2(x) \Rightarrow e_2)_{r'}$	$\rightarrow$	$(\mathbf{protect}_{ir}\ e_j[v/x])$	if $r \sqsubseteq r'$
$(\mu f : s.\, e)$	$\rightarrow$	$e[(\lambda x : s_1.\, ((\mu f : s.\, e)\ x)_r)_{(r,ir)}/f]$	if $s = (s_1 \rightarrow s_2, (r, ir))$
$(\mathbf{protect}_{ir}\ v)$	$\rightarrow$	$v \bullet ir$	

Table 2: Subtyping Rules for Pure Functional Language.

$$\frac{s_1 \leq s_2 \qquad s_2 \leq s_3}{s_1 \leq s_3} \qquad \frac{\kappa \leq \kappa'}{(\mathbf{unit}, \kappa) \leq (\mathbf{unit}, \kappa')}$$

$$\frac{\kappa \leq \kappa' \qquad s_i \leq s_i'}{((s_1 + s_2), \kappa) \leq ((s_1' + s_2'), \kappa')} \qquad \frac{\kappa \leq \kappa' \qquad s_i \leq s_i'}{((s_1 \times s_2), \kappa) \leq ((s_1' \times s_2'), \kappa')}$$

$$\frac{\kappa \leq \kappa' \qquad s_1' \leq s_1 \qquad s_2 \leq s_2'}{((s_1 \rightarrow s_2), \kappa) \leq ((s_1' \rightarrow s_2'), \kappa')}$$

Table 3: Typing Rules for Pure Functional Language.

$$[Var]\ \ \Gamma, x : s \vdash x : s \qquad [Unit]\ \ \Gamma \vdash ()_\kappa : (\mathbf{unit}, \kappa)$$

$$[Sub]\ \ \frac{\Gamma \vdash e : s \qquad s \leq s'}{\Gamma \vdash e : s'} \qquad [Rec]\ \ \frac{\Gamma, f : s \vdash e : s}{\Gamma \vdash (\mu f : s.\, e) : s}\ \ s \text{ is a function type}$$

$$[Lam]\ \ \frac{\Gamma, x : s_1 \vdash e : s_2}{\Gamma \vdash (\lambda x : s_1.\, e)_\kappa : (s_1 \rightarrow s_2, \kappa)} \qquad [App]\ \ \frac{\Gamma \vdash e : (s_1 \rightarrow s_2, (r, ir)) \qquad \Gamma \vdash e' : s_1}{\Gamma \vdash (e\ e')_{r'} : s_2 \bullet ir}\ \ r \sqsubseteq r'$$

$$[Pair]\ \ \frac{\Gamma \vdash e_1 : s_1 \qquad \Gamma \vdash e_2 : s_2}{\Gamma \vdash \langle e_1, e_2\rangle_\kappa : (s_1 \times s_2, \kappa)} \qquad [Proj]\ \ \frac{\Gamma \vdash e : (s_1 \times s_2, (r, ir))}{\Gamma \vdash (\mathbf{proj}_i\ e)_{r'} : s_i \bullet ir}\ \ r \sqsubseteq r'$$

$$[Inj]\ \ \frac{\Gamma \vdash e : s_i}{\Gamma \vdash (\mathbf{inj}_i\ e)_\kappa : (s_1 + s_2, \kappa)} \qquad [Protect]\ \ \frac{\Gamma \vdash e : s}{\Gamma \vdash (\mathbf{protect}_{ir}\ e) : s \bullet ir}$$

$$[Case]\ \ \frac{\Gamma \vdash e : (s_1 + s_2, (r, ir)) \qquad \Gamma, x : s_i \vdash e_i : s}{\Gamma \vdash (\mathbf{case}\ e\ \mathbf{of}\ \mathbf{inj}_1(x) \Rightarrow e_1 \mid \mathbf{inj}_2(x) \Rightarrow e_2)_{r'} : s \bullet ir}\ \ r \sqsubseteq r'$$

$\leq$ relation on security properties. Define

$$(r, ir) \leq (r', ir') \quad \text{iff} \quad r \sqsubseteq r', ir \sqsubseteq ir'.$$

The subtyping rules formalize the idea that one may always increase the security property of a value.

The typing rules appear in Table 3. Abusing notation, we write $(t, \kappa) \bullet ir$ to denote the secure type $(t, \kappa \bullet ir)$. The rules for type-checking constructors are straightforward. For type-checking destructors, the rules guarantee that the destructor has the permission to destruct the value (apply a function, project from a pair, or branch on a sum). Note that like the operational semantics, the security of the destructed value is promoted to reflect the *indirect* readers of the destructed value. This type system satisfies Subject Reduction and Progress (see Appendix for proofs).

Theorem 2.1 (Subject Reduction) *Suppose $\emptyset \vdash e : s$ and $e \to e'$. Then $\emptyset \vdash e' : s$.*

Theorem 2.2 (Progress) *Suppose $\emptyset \vdash e : s$ and e is not a value. Then there is a reduction $e \to e'$.*

These theorems show that, for well-typed, closed expressions, one may omit the security checks in the operational semantics without compromising the security of expressions.

The subject reduction result provides a fairly direct proof that our type system enforces reader security. Objects are created with their reader annotation set to some specific security group and this annotation is preserved throughout the program. Specifically, reader annotations may only be changed by **protect**, whose effect is to *increase* security, *i.e.*, set more restrictive access to the object.

However, the situation for indirect readers is more subtle. For example, we would like to prove that if x is a high security variable (with respect to indirect readers) and e is a low security expression that contains x, then no matter what value we give to x, the resulting evaluation of e does not change (assuming it terminates). More generally, we want to show that if an expression e of low security has a high security sub-expression, then we can arbitrarily change the high security sub-expression without changing the value of e. However, this property does not hold in general. First, it does not hold if e evaluates to a value that contains abstractions. Hence, we restrict the type of e so that it contains only **unit**, sums and products (call these *ground types*). Second, it does not hold if e evaluates to a value whose subcomponents have higher security than e. Hence, we further restrict the type of e so that security properties decrease as we descend into its type structure *e.g.*, $((\mathbf{unit}, (L, L)) + (\mathbf{unit}, (L, L)), (H, H))$ (call these *transparent types*). To formally state property, we shall use contexts: $C[\cdot]$ denotes a context (expression with a hole in it); $C[e]$ denotes the expression obtained by filling context $C[\cdot]$ with expression e. We also define a special equivalence relation to factor out termination issues: $e \simeq e'$ if whenever both expressions halt at values, the values (when stripped of security information) are identical. We can now state:

Theorem 2.3 (Noninterference) *Suppose $\emptyset \vdash e : (t, (r, ir))$, $\emptyset \vdash C[e] : (t', (r', ir'))$, t' is a transparent ground type and $ir \not\sqsubseteq ir'$. If e' is an expression where $\emptyset \vdash e' : (t, (r, ir))$, then $C[e] \simeq C[e']$.*

For simplicity, we have restricted this theorem to closed terms e; it can be generalized to open terms. The proof uses a denotational semantics of the language and a logical-relations-style argument, and is given in the Appendix. The proof is particularly simple, especially when compared with other proofs based on direct reasoning with the operational semantics (*cf.* [18]).

3 Assignment and Concurrency

The calculus in the previous section is single threaded and side-effect free. This is inadequate to model the behavior of a collection of agents that execute concurrently and interact (*e.g.*, via a shared store or file system). To model such a system, we extend the basic calculus with assignment (via ML-style reference cells), generalize evaluation to a multi-process setting, and add a "spawn" operation to create new processes.

We first extend the definition of basic values bv and expressions e by

$$\begin{array}{rcl} bv & ::= & \cdots \mid l^s \\ e & ::= & \cdots \mid (\mathbf{ref}_s\ e)_\kappa \mid (\mathbf{write}\ e\ e)_\kappa \mid (\mathbf{read}\ e)_r \mid (\mathbf{spawn}\ e)_\kappa \end{array}$$

where l^s is a location (we assume an infinite sequence of locations at each type s; whenever a new location is needed, we use the next available location in the sequence). We modify the definition of types t to include reference types and also to change arrow types so that they carry a latent "effect" ir, representing a lower bound on the security of cells that may be written when the function is executed.

$$\begin{array}{rcl} \kappa & ::= & (r, ir) \\ t & ::= & \mathbf{unit} \mid (s + s) \mid (s \times s) \mid (s \xrightarrow{ir} s) \mid (\mathbf{ref}\, s) \end{array}$$

We extend evaluation contexts appropriately:

$$E \quad ::= \quad \cdots \mid (\mathbf{ref}_s\ E)_\kappa \mid (\mathbf{write}\ E\ e)_r \mid (\mathbf{write}\ v\ E)_r \mid (\mathbf{read}\ E)_r$$

Recall that $\bot$ denotes the bottom (most insecure) security group. Abusing notation, define $ir(E)$ by

$$ir(E) = \begin{cases} \bot & \text{if } E = [\cdot] \\ ir \sqcup ir' & \text{if } E = (\mathbf{protect}_{ir}\ E') \text{ and } ir(E') = ir' \\ ir(E') & \text{if, for instance, } E = (E'\ e)_r \end{cases}$$

We use the notation E_{ir} to denote an evaluation context with $ir(E) = ir$. A **state** is a finite partial function from typed locations l^s into values.

The starting point of the operational semantics for the extended calculus is the collection of simple redex rules given previously in Table 1. Again we lift these rules to arbitrary terms via

$$\frac{e \to e'}{E[e] \to E[e']}$$

where E is understood to be the extended definition of contexts given above. Using this basic notion of $e \to e'$, we now define reduction for side-effect operations and thread spawning. Specifically, Table 4 defines a reduction relation $(e_1, \ldots, e_n; \sigma) \Rightarrow (e'_1, \ldots, e'_{n+k}; \sigma')$, where σ is a state.

Subtyping in the system with effects is exactly the same as before, except that the rule for function types now becomes

$$\frac{\kappa \le \kappa' \quad ir' \sqsubseteq ir \quad s'_1 \le s_1 \quad s_2 \le s'_2}{((s_1 \xrightarrow{ir} s_2), \kappa) \le ((s'_1 \xrightarrow{ir'} s'_2), \kappa')}$$

and the rule for reference types is

$$\frac{\kappa \le \kappa'}{(\mathbf{ref}\, s, \kappa) \le (\mathbf{ref}\, s, \kappa')}$$

Note that subtyping on reference types only affects top-level security properties. Table 5 presents the typing rules for the extended calculus. This type system is essentially the previous system with an effect system layered over the top of it in the style of [17]. This effect system tracks potential information leakage/dependency that may be introduced by reference cells. Each context carries with it a security group ir that is a lower bound on the security of the reference cells that may be written in that context; as expected, this security group is carried over onto arrow types.

Analogs of the Subject Reduction Theorem 2.1 and Progress Theorem 2.2 can be established for this system; the proofs are quite similar to the proofs in the Appendix. Unfortunately, noninterference is problematic in concurrent setting (see [9] for a discussion). Consider a system with two agents A, B and C, and suppose that there is some variable `x` that contains information that should be kept secret from agent C. The first agent, A, generates a random number, and puts it into a variable `tmp` that everyone can read. Agent B waits for a while and then just copies the contents of `x` into `tmp`. Agent C reads `tmp` and immediately terminates. Now, although C cannot tell with certainty that it has captured the contents of `x` or some random garbage, it clearly finds out more about `x` than it should. However, since the possible set of behaviors of C (*i.e.*, the possible values C generates) is independent of the initial value of `x`, the use of equivalence classes would lead us to the conclusion that this system is secure.

We have not found a satisfactory, general, abstract theorem expressing a noninterference property in the presence of concurrency. Curiously, though, we expect our *lemmas* using logical relations to hold in the setting with side effects and concurrency, and are exploring notions of noninterference using logical relations.

Table 4: Operational Semantics for Effects.

$$\frac{e \to e'}{(\ldots, E_{ir'}[e], \ldots; \sigma) \Rightarrow (\ldots, E_{ir'}[e'], \ldots; \sigma)}$$

$$\begin{array}{lcll}
(\ldots, E_{ir'}[(\mathbf{spawn}\ e)_\kappa], \ldots; \sigma) & \Rightarrow & (\ldots, (\mathbf{protect}_{ir'}\ e), E_{ir'}[()_\kappa], \ldots; \sigma) & \\
(\ldots, E_{ir'}[(\mathbf{ref}_s\ v)_\kappa], \ldots; \sigma) & \Rightarrow & (\ldots, E_{ir'}[l^s_\kappa], \ldots; \sigma[l^s \mapsto v \bullet ir']) & \text{if } l^s \notin dom(\sigma) \\
(\ldots, E_{ir'}[(\mathbf{write}\ l^s_{(r,ir)}\ v)_{r'}], \ldots; \sigma) & \Rightarrow & (\ldots, E_{ir'}[v], \ldots; \sigma[l^s \mapsto v \bullet ir']) & \text{if } r \sqsubseteq r' \\
(\ldots, E_{ir'}[(\mathbf{read}\ l^s_{(r,ir)})_{r'}], \ldots; \sigma) & \Rightarrow & (\ldots, E_{ir'}[\sigma(l^s) \bullet ir], \ldots; \sigma) & \text{if } r \sqsubseteq r'
\end{array}$$

Table 5: Typing Rules for Effects.

$$[Sub] \quad \frac{\Gamma \vdash_{ir} e : s \qquad s \leq s'}{\Gamma \vdash_{ir} e : s'}$$

$$[Var] \quad \Gamma, x : s \vdash_{ir} x : s$$

$$[Unit] \quad \Gamma \vdash_{ir} ()_\kappa : (\mathbf{unit}, \kappa)$$

$$[Lam] \quad \frac{\Gamma, x : s_1 \vdash_{ir'} e : s_2}{\Gamma \vdash_{ir} (\lambda x : s_1.\ e)_\kappa : (s_1 \xrightarrow{ir'} s_2, \kappa)}$$

$$[Pair] \quad \frac{\Gamma \vdash_{ir} e_1 : s_1 \qquad \Gamma \vdash_{ir} e_2 : s_2}{\Gamma \vdash_{ir} \langle e_1, e_2 \rangle_\kappa : (s_1 \times s_2, \kappa)}$$

$$[Inj] \quad \frac{\Gamma \vdash_{ir} e : s_i}{\Gamma \vdash_{ir} (\mathbf{inj}_i\ e)_\kappa : (s_1 + s_2, \kappa)}$$

$$[App] \quad \frac{\Gamma \vdash_{ir'} e : (s_1 \xrightarrow{ir''} s_2, (r, ir)) \qquad \Gamma \vdash_{ir'} e' : s_1}{\Gamma \vdash_{ir'} (e\ e')_{r'} : s_2 \bullet ir} \quad r \sqsubseteq r',\ (ir' \sqcup ir) \sqsubseteq ir''$$

$$[Proj] \quad \frac{\Gamma \vdash_{ir'} e : (s_1 \times s_2, (r, ir))}{\Gamma \vdash_{ir'} (\mathbf{proj}_i\ e)_{r'} : s_i \bullet ir} \quad r \sqsubseteq r'$$

$$[Case] \quad \frac{\Gamma \vdash_{ir'} e : (s_1 + s_2, (r, ir)) \qquad \Gamma, x : s_i \vdash_{ir''} e_i : s}{\Gamma \vdash_{ir'} (\mathbf{case}\ e\ \mathbf{of}\ \mathbf{inj}_1(x) \Rightarrow e_1 \mid \mathbf{inj}_2(x) \Rightarrow e_2)_{r'} : s \bullet ir} \quad r \sqsubseteq r',\ (ir' \sqcup ir) \sqsubseteq ir''$$

$$[Protect] \quad \frac{\Gamma \vdash_{ir''} e : s}{\Gamma \vdash_{ir'} (\mathbf{protect}_{ir}\ e) : s \bullet ir} \quad r \sqsubseteq r',\ (ir' \sqcup ir) \sqsubseteq ir''$$

$$[Spawn] \quad \frac{\Gamma \vdash_{ir'} e : s}{\Gamma \vdash_{ir'} (\mathbf{spawn}\ e)_\kappa : (\mathbf{unit}, \kappa)}$$

$$[Loc] \quad \Gamma \vdash_{ir'} l^s_\kappa : (\mathbf{ref}\ s, \kappa)$$

$$[Ref] \quad \frac{\Gamma \vdash_{ir'} e : s}{\Gamma \vdash_{ir'} (\mathbf{ref}_s\ e)_\kappa : (\mathbf{ref}\ s, \kappa)} \quad (s \bullet ir') = s$$

$$[Assign] \quad \frac{\Gamma \vdash_{ir'} e_1 : (\mathbf{ref}\ s, (r, ir)) \qquad \Gamma \vdash_{ir'} e_2 : s}{\Gamma \vdash_{ir'} (\mathbf{write}\ e_1\ e_2)_{r'} : s} \quad r \sqsubseteq r',\ (s \bullet ir') = s$$

$$[Deref] \quad \frac{\Gamma \vdash_{ir'} e : (\mathbf{ref}\ s, (r, ir))}{\Gamma \vdash_{ir'} (\mathbf{read}\ e)_{r'} : s \bullet ir} \quad r \sqsubseteq r'$$

4 Integrity

We now sketch how to add integrity to the basic calculus of Section 2 and the extended calculus of Section 3, using the concepts of creators and indirect creators. Recall that *creators* track the agents that directly built the value, whereas *indirect creators* track the agents that may have influence over the eventual choice of a value.

Creators and indirect creators are drawn from the same underlying hierarchy of security groups as readers and indirect readers. High integrity is modeled by points near the top of the hierarchy, low integrity by points near the bottom. But there is a twist with respect to subtyping. Recall that for readers, one may always *restrict* access to a value, *e.g.*, change the reader annotation to a higher security group. For creators, it works just the opposite way: one may always *weaken* the integrity of a value, *e.g.*, change the creator annotation to a *lower* security group. More formally, security properties now incorporate creator and indirect creator information:

$$\kappa \quad ::= \quad (r, ir, c, ic).$$

The variables r, ir, c and ic range over security groups; we assume that $ic \sqsubseteq c$. Subsumption for κ becomes

$$(r, ir, c, ic) \leq (r', ir', c', ic') \quad \text{iff } r \sqsubseteq r',\ ir \sqsubseteq ir',\ c' \sqsubseteq c, \text{ and } ic' \sqsubseteq ic$$

which formalizes the intuition that one may always weaken the integrity of a value.

The operational semantics must now track indirect creators. For example, the rule for **case** becomes

$$(\mathbf{case}\ (\mathbf{inj}_j\ v)_{r,ir,c,ic}\ \mathbf{of}\ \mathbf{inj}_1(x) \Rightarrow e_1 \mid \mathbf{inj}_2(x) \Rightarrow e_2)_{r'} \rightarrow (\mathbf{protect}_{ir,ic \sqcap r'}\ \ e_j[v/x]) \quad \text{if } r \sqsubseteq r'$$

Note that the **protect** operation must take into account indirect creators. The rule registers the reader r' of the injected value as an indirect creator of the result of the computation. Typing rules that involve the $\bullet$ operation must be modified. For example, the **case** rule becomes

$$[Case] \quad \frac{\Gamma \vdash e : (s_1 + s_2, (r, ir, c, ic)) \qquad \Gamma, x : s_i \vdash e_i : s}{\Gamma \vdash (\mathbf{case}\ e\ \mathbf{of}\ \mathbf{inj}_1(x) \Rightarrow e_1 \mid \mathbf{inj}_2(x) \Rightarrow e_2)_{r'} : s \bullet (ir, ic \sqcap r')} \ \ r \sqsubseteq r'$$

We have proven Subject Reduction and Progress Theorems analogous to Theorems 2.1 and 2.2 for this system. We can also prove a security result for indirect creators that is analogous to Theorem 2.3:

Theorem 4.1 (Noninterference) *Suppose $\emptyset \vdash e : (t, (r, ir))$, $\emptyset \vdash C[e] : (t', (r', ir'))$, t' is a transparent ground type and $ir \not\sqsubseteq ir'$. If e' is an expression where $\emptyset \vdash e' : (t, (r, ir))$, then $C[e] \simeq C[e']$.*

Intuitively, if the indirect creators of the subexpression e do not include that of the entire computation, then e cannot influence the result of the computation. The proofs of these results use the techniques established in the Appendix.

Creators and indirect creators can also be added to the calculus of Section 3. Recall that in the case of readers, the type system must guarantee that information does not leak out via side effects. A similar property must be guaranteed in the case of creators: we must make sure that indirect creators of the computation are carried over onto the values written in reference cells. Therefore, judgements $\Gamma \vdash_{ir} e : s$ must be changed to $\Gamma \vdash_{ir,ic} e : s$, where ic is a lower bound on the integrity of values that may be written to reference cells in the evaluation of e. As before, indirect information is placed over $\rightarrow$ to represent the latent effects of a computation. The type-checking rules for abstraction and application thus become

$$[Lam] \qquad \frac{\Gamma, x : s_1 \vdash_{ir', ic'} e : s_2}{\Gamma \vdash_{ir} (\lambda x : s_1.\ e)_\kappa : (s_1 \xrightarrow{ir', ic'} s_2, \kappa)}$$

$$[App] \quad \frac{\Gamma \vdash_{ir'} e : (s_1 \xrightarrow{ir'', ic''} s_2, (r, ir)) \qquad \Gamma \vdash_{ir'} e' : s_1}{\Gamma \vdash_{ir'} (e\ e')_{r'} : s_2 \bullet (ir, ic \sqcap r')} \ \ r \sqsubseteq r',\ (ir' \sqcup ir) \sqsubseteq ir'', ic'' \sqsubseteq (ic' \sqcap ic)$$

We have proven Subject Reduction and Progress Theorems for this system; the proofs follow the structure of the proofs in the Appendix for the pure case.

5 Discussion

Our work is not the first to use a programming language framework for security. The interpreter for Perl 5.0, for instance, can be put into a special mode which tracks information flow and rejects programs that may reveal secret information. Type systems have been used to statically check programs before they are run. Recent work by Volpano, Smith and Irvine reformulates Denning's framework as a type system in order to reason about its soundness [18], and Abadi's type system for the Spi calculus may be used to reason about protocols [1, 2]. Type systems have been also used for the related problem of reasoning about trustworthiness of data. For instance, [12] introduces a calculus in which one can explicitly annotate expressions as trusted or distrusted and check their trust/distrust status; this system enforces consistent use of these annotations, although one can freely coerce from trusted to distrusted and vice-versa. Concurrency issues were first addressed by [4], although there appear to be some difficulties with that approach—see [18].

The main novelties of our work are the use of both access protection and information flow, and the incorporation of higher-order functions and data structures; these are both essential for a development of practical languages that provide mechanisms for security. This introduces a number of new technical issues that have not been previously addressed.

The system we have presented is vulnerable to timing attacks. For example we could write

```
let val t1 : (int, L, L) = getTime()
    val tmp = if secureBool : (bool, H, H) then longComputation else shortComputation
    val t2 : (int, L, L) = getTime()
    val insecureBool : (bool, L, L) = ((t2 - t1) > timeForShortComputation)
in insecureBool end
```

where `getTime` gets the current time, and `longComputation` is some computation that takes longer than `shortComputation`. This program allows us to leak information about the secure value `secureBool` to the low security value `insecureBool`. In short, our type system does not protect the execution time of computations. Depending on the latency and accuracy of `getTime` and scheduling issues, this could reliably leak information at a rate of perhaps 1 bit/ms – 1 bit/sec. This would be disastrous for certain applications: *e.g.*, we could leak a 100-bit cryptographic key in the order of seconds.

This bandwidth of timing attacks could be reduced by restricting access and accuracy of `getTime`. Alternatively, we could change the type system. The key type rule is one for **case** statements, where the security context (the subscript on $\vdash$ in Table 5) is increased as we move into the body of the **case** statement to reflect the security of the tested expression. We could constrain these contexts to be equal (*i.e.*, so that we cannot look at a high security value unless we are already in a high security context), and only allow contexts to be increased in security at **spawn** expressions. The idea is that if we are in a low security context and want to do some computation with a high security value, then we must first spawn off a high security process for this purpose. We can still leak information *between processes* by taking more or less of the processor's resources (time or space) according to some high security value, but we have significantly reduced the bandwidth of these attacks. A more speculative approach involves forcing the arms of a **case** statement to take the same time and space resources by adding timeout mechanisms and various padding operations. An important area of future work involves studying the tradeoffs of implementation costs and programming inconvenience versus reduction in timing attack vulnerability.

As with many other security systems, our approach relies on a TCB (trusted computer base): in our case trusted type-checking/compilation/runtime infrastructure. A failure in any of these components potentially breaks the entire security system. It would be possible to factor out some of the critical components by moving to a bytecode/bytecode-verifier organization (*à la* Java), although the benefits of doing so are unclear.

We view the SLam calculus as a first step towards providing a language basis for secure systems programming. It deals with the essence of computing with secure information, but a number of important issues remain. First, the type system we have presented is monomorphic. Clearly this is too restrictive: we need to be able to write code that behaves uniformly over a variety of security groups (*e.g.*, in writing a generic string editing/searching package). We are currently investigating two approaches to this problem: parametric security types and a notion of "type dynamic" for security types. The former involves bounded quantification, and it is not clear we can compute concise, intuitive representations of types; the latter involves runtime overheads.

Second, our type system is static, but the security of objects changes dynamically. For instance, in a file system, the files that one can read today will probably be different from those one can read tomorrow. How can we accommodate new files, new objects, new cells, new agents, changing security groups, etc? We plan to address these issues using a dynamically typed object manager. The basic idea is that access to shared objects is via the object manager; although each program is statically typed, a program's interface to the object manager is via dynamic types (at runtime, a dynamically typed object returned from the object manager must be unpacked and its security properties checked before the raw object it contains is passed to the internals of the program).

Third, any practical language based on the SLam calculus must provide ways to reduce the amount of type information that must be specified by a programmer; the core SLam calculus is an explicitly typed calculus. Can we perform effective type reconstruction? What kinds of language support should we provide? For example, it would be useful to introduce a statically scoped construct that defines a default security group for all objects created in its scope, *i.e.*, like UNIX's `umask`.

We are investigating these issues in the context of an implementation of our type system for Java. While many of the appropriate typing rules for Java can be adapted easily from the SLam calculus, some new issues arise from exceptions, `break`, `continue`, `return`, and `instanceOf`. The implementation is joint work with Philip Wickline.

Acknowledgements: We thank Kathleen Fisher, Geoffrey Smith, Ramesh Subrahmanyam, Dennis Volpano, and Philip Wickline for helpful comments.

References

[1] M. Abadi. Secrecy by typing in security protocols. In *Proceedings of TACS*, 1997. To appear.

[2] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 36–47, 1997.

[3] G. Andrews and R. Reitman. An axiomatic approach to information flow in programs. *ACM Trans. Programming Languages and Systems*, 2(1):56–76, 1980.

[4] J. Banâtre, C. Bryce, and D. L. Metáyer. Compile-time detection of information flow in sequential programs. In European Symposium on Research in Computer Security, number 875 in Lect. Notes in Computer Sci., pages 55–73. Springer-Verlag, 1994.

[5] D. Denning. *Secure Information Flow in Computer Systems.* PhD thesis, Purdue University, 1975.

[6] D. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–242, 1976.

[7] D. Denning and P. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[8] M. Felleisen. The theory and practice of first-class prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190. ACM, 1988.

[9] D. McCullough. Noninterference and the composability of security properties. In *1988 IEEE Symposium on Security and Privacy*, pages 177–186, 1988.

[10] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[11] J. C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[12] J. Palsberg and P. Ørbæk. Trust in the λ-calculus. In *Proceedings of the 1995 Static Analysis Symposium*, number 983 in Lect. Notes in Computer Sci. Springer-Verlag, 1995.

[13] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus Univ., Computer Science Dept., Denmark, 1981.

[14] G. D. Plotkin. (Towards a) logic for computable functions. Unpublished manuscript, CSLI Summer School Notes, 1985.

[15] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. North Holland, Amsterdam, 1983.

[16] J. G. Riecke and A. Sandholm. A relational account of call-by-value sequentiality. In *Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 258–267, 1997.

[17] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2:245–271, 1992.

[18] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):1–21, 1996.

A Proofs for Pure Functional Language with Secrecy

A.1 Basic Facts

Proposition A.1 *Suppose* $s_1 \leq s_2$ *and* $ir_1 \sqsubseteq ir_2$. *Then* $s_1 \bullet ir_1 \leq s_2 \bullet ir_2$.

Proposition A.2 *For any secrecy property* κ *and type* s,

1. $\kappa \bullet ir \bullet ir' = \kappa \bullet ir' \bullet ir$.
2. $s \bullet ir \bullet ir' = s \bullet ir' \bullet ir$.

A.2 Substitution Lemma

Lemma A.3 *If* $\Gamma \vdash v : s'$ *and* $\Gamma, x : s' \vdash e : s$, *then* $\Gamma \vdash e[v/x] : s$.

Proof: By induction on the proof of $\Gamma, x : s' \vdash e : s$. We consider a few of the most representative cases and leave the others to the reader.

1. $\Gamma, x : s' \vdash y : s$ where $y \neq x$. Obvious.
2. $\Gamma, x : s' \vdash x : s$ where $s = s'$. Since $e[v/x] = v$, we are done.
3. $\Gamma, x : s' \vdash (\lambda y : s_1.\, e')_\kappa : (s_1 \to s_2, \kappa)$, where $\Gamma, x : s', y : s_1 \vdash e' : s_2$ and $y \neq x$. By induction, $\Gamma, y : s_1 \vdash e'[v/x] : s_2$. Thus, by rule $[Lam]$,

   $$\Gamma \vdash (\lambda y : s_0.\, e')_\kappa[v/x] : s$$

   as desired.

This completes the induction and hence the proof. ■

A.3 Subject Reduction Theorem

Lemma A.4 *Suppose* $\emptyset \vdash e_a : s_a$, *and* $e_a \to e_b$. *Then* $\emptyset \vdash e_b : s_a$.

Proof: By cases depending on the reduction rule used.

1. $((\lambda x : s_1.\, e)_{r,ir}\ v)_{r'} \to (\mathbf{protect}_{ir'}\ e[v/x])$. By assumption, $\emptyset \vdash e_a : s_a$. Since e_a is $((\lambda x : s_1.\, e)_{r,ir}\ v)_{r'}$, this derivation must end in a (possibly empty) series of $[Sub]$ applications that are immediately preceded by an application of $[App]$. Hence there exists some $s'_a \leq s_a$ and a derivation $\emptyset \vdash e_a : s'_a$ whose last rule application is $[App]$. By inspection of $[App]$, there exist derivations for:

   $$\begin{array}{rl} & \emptyset \vdash (\lambda x : s_1.\, e)_{r,ir} : (s'_1 \to s'_2, r'', ir'') \\ & \emptyset \vdash v : s'_1 \\ \text{where} & r'' \sqsubseteq r' \text{ and } s'_2 \bullet ir'' = s'_a \leq s_a \end{array}$$

The derivation $\emptyset \vdash (\lambda x : s_1.\, e)_{r,ir} : (s_1' \to s_2', r'', ir'')$ must end in a (possibly empty) series of $[Sub]$ applications that are immediately preceded by an application of $[Abs]$. Hence there exists an s_3 where $s_3 \leq (s_1' \to s_2', r'', ir'')$, and a derivation $\emptyset \vdash (\lambda x : s_1.\, e)_{r,ir} : s_3$ whose last rule is $[Abs]$. By the $[Abs]$ rule, there is a derivation:

$$\begin{array}{rl} & x : s_1 \vdash e : s_2 \\ \text{where} & s_3 = (s_1 \to s_2, r, ir) \end{array}$$

Now, $s_3 = (s_1 \to s_2, r, ir) \leq (s_1' \to s_2', r'', ir'')$ implies that:

$$\begin{array}{l} s_1' \leq s_1 \\ s_2 \leq s_2' \\ ir \sqsubseteq ir'' \end{array}$$

Combining $s_1' \leq s_1$ with $\emptyset \vdash v : s_1'$ implies $\emptyset \vdash v : s_1$ by the $[Sub]$ rule. Hence we have $x : s_1 \vdash e : s_2$ and $\emptyset \vdash v : s_1$, and so by the Substitution Lemma,

$$\emptyset \vdash e[v/x] : s_2.$$

By the $[Protect]$ rule, $\emptyset \vdash (\mathbf{protect}_{ir}\ e[v/x]) : s_2 \bullet ir$. Now, $s_2 \leq s_2'$ and $ir \sqsubseteq ir''$ and so by Proposition A.1,

$$s_2 \bullet ir \leq s_2' \bullet ir'' = s_a' \leq s_a$$

Hence, by (Sub), $\emptyset \vdash (\mathbf{protect}_{ir}\ e[v/x]) : s_a$.

2. $(\mathbf{proj}_i\ \langle v_1, v_2 \rangle_{(r,ir)})_{r'} \to (\mathbf{protect}_{ir}\ v_i)$, where $r \sqsubseteq r'$. Since $\emptyset \vdash e_a : s_a$, there must exist $s_a' \leq s_a$ and a derivation $\emptyset \vdash (\mathbf{proj}_i\ \langle v_1, v_2 \rangle_{(r,ir)})_{r'} : s_a'$ whose last rule application is $[Proj]$. By the $[Proj]$ rule:

$$\begin{array}{rl} & \emptyset \vdash \langle v_1, v_2 \rangle_{(r,ir)} : (s_1' \times s_2', (r'', ir'')) \\ \text{where} & s_i' \bullet ir'' = s_a' \leq s_a \text{ and } r'' \sqsubseteq r' \end{array}$$

Since $\emptyset \vdash \langle v_1, v_2 \rangle_{(r,ir)} : (s_1' \times s_2', (r'', ir''))$, there is a derivation of $\emptyset \vdash \langle v_1, v_2 \rangle_{(r,ir)} : s_3$ ending in a use of the $(Pair)$ rule, such that $s_3 \leq (s_1' \times s_2', (r'', ir''))$. From the $(Pair)$ rule we have derivations:

$$\begin{array}{rl} & \emptyset \vdash v_1 : s_1 \\ & \emptyset \vdash v_2 : s_2 \\ \text{where} & s_3 = (s_1 \times s_2, (r, ir)) \end{array}$$

and so by the $[Protect]$ rule, $\emptyset \vdash (\mathbf{protect}_{ir}\ v_i) : s_i \bullet ir$. Since $s_3 = (s_1 \times s_2, (r, ir)) \leq (s_1' \times s_2', (r'', ir''))$,

$$\begin{array}{l} s_1 \leq s_1' \\ s_2 \leq s_2' \\ ir \sqsubseteq ir'' \end{array}$$

Hence $s_i \bullet ir \leq s_i' \bullet ir'' = s_a' \leq s_a$ by Proposition A.1, and so $\emptyset \vdash (\mathbf{protect}_{ir}\ v_i) : s_a$ by (Sub).

3. $(\mathbf{case}\ (\mathbf{inj}_j\ v)_{r,ir}\ \mathbf{of}\ \mathbf{inj}_1(x) \Rightarrow e_1 \mid \mathbf{inj}_2(x) \Rightarrow e_2)_{r'} \to (\mathbf{protect}_{ir}\ e_j[v/x])$, where $r \sqsubseteq r'$. Since $\emptyset \vdash e_a : s_a$, there must exist $s_a' \leq s_a$ and a derivation $\emptyset \vdash (\mathbf{case}\ (\mathbf{inj}_j\ v)_{(r,ir)}\ \mathbf{of}\ \mathbf{inj}_1(x) \Rightarrow e_1 \mid \mathbf{inj}_2(x) \Rightarrow e_2)_{r'} : s_a'$, whose last rule application is $[Case]$. By the $[Case]$ rule, there is a derivation:

$$\begin{array}{rl} & \emptyset \vdash (\mathbf{inj}_j\ v)_{r,ir} : (s_1' + s_2', (r'', ir'')) \\ & x : s_i' \vdash e_i : s \\ \text{where} & s \bullet ir'' = s_a' \leq s_a \text{ and } r'' \sqsubseteq r' \end{array}$$

Hence, there is a derivation $\emptyset \vdash (\mathbf{inj}_j\ v)_{(r,ir)} : s_3$ whose last rule is $[Inj]$ and where $s_3 \leq (s_1' + s_2', (r'', ir''))$. By the $[Inj]$ rule, there are derivations:

$$\begin{array}{rl} & \emptyset \vdash v : s_j \\ \text{where} & s_3 = (s_1 + s_2, (r, ir)) \end{array}$$

Since $s_3 = (s_1 + s_2, r, ir) \leq (s_1' + s_2', (r'', ir''))$,

$$\begin{array}{l} s_1 \leq s_1' \\ s_2 \leq s_2' \\ ir \sqsubseteq ir'' \end{array}$$

Now, $\emptyset \vdash v : s_j$ and $s_j \leq s_j'$ implies $\emptyset \vdash v : s_j'$. Hence, we have $x : s_j' \vdash e_j : s$ and $\emptyset \vdash v : s_j'$, and so by the Substitution Lemma

$$\emptyset \vdash e_j[v/x].$$

By the $[Protect]$ rule, $\emptyset \vdash (\mathbf{protect}_{ir}\ e_j[v/x]) : s \bullet ir$. Now, $ir \sqsubseteq ir''$ and so by Proposition A.1,

$$s \bullet ir \leq s \bullet ir'' = s_a' \leq s_a$$

Hence, by (Sub), $\emptyset \vdash (\mathbf{protect}_{ir}\ e_j[v/x]) : s_a$.

4. $(\mathbf{protect}_{ir}\ ()_\kappa) \rightarrow ()_{\kappa \bullet ir}$. Since $\emptyset \vdash e_a : s_a$, there exists $s_a' \leq s_a$ and a derivation $\emptyset \vdash (\mathbf{protect}_{ir}\ ()_\kappa) : s_a'$ whose last rule is $[Protect]$. Hence, there is derivation

$$\begin{array}{ll} & \emptyset \vdash ()_\kappa : s \\ \text{where} & s \bullet ir = s_a' \leq s_a \end{array}$$

The derivation of $\emptyset \vdash ()_\kappa : s$ must consist of an application of the $[Unit]$ rule followed by some number of applications of $[Sub]$. Hence $(\mathbf{unit}, \kappa) \leq s$. Now, applying the $[Unit]$ rule to $()_{\kappa \bullet ir}$ gives:

$$\emptyset \vdash ()_{\kappa \bullet ir} : (\mathbf{unit}, \kappa \bullet ir)$$

Since $(\mathbf{unit}, \kappa \bullet ir) = (\mathbf{unit}, \kappa) \bullet ir$ and $(\mathbf{unit}, \kappa) \leq s$, it follows from Proposition A.1 that $(\mathbf{unit}, \kappa \bullet ir) \leq s \bullet ir = s_a' \leq s_a$. Hence, $\emptyset \vdash ()_{\kappa \bullet ir} : s_a$ by $[Sub]$.

5. $(\mathbf{protect}_{ir}\ \langle v_1, v_2 \rangle_\kappa) \rightarrow \langle v_1, v_2 \rangle_{\kappa \bullet ir}$. Since $\emptyset \vdash e_a$, there exists $s_a' \leq s_a$ and a derivation $\emptyset \vdash (\mathbf{protect}_{ir}\ \langle v_1, v_2 \rangle_\kappa) : s_a'$ whose last rule is $[Protect]$, and so:

$$\begin{array}{ll} & \emptyset \vdash \langle v_1, v_2 \rangle_\kappa : s \\ \text{where} & s \bullet ir = s_a' \leq s_a \end{array}$$

Since $\emptyset \vdash \langle v_1, v_2 \rangle_\kappa : s$, there exists $s' \leq s$ and a derivation $\emptyset \vdash \langle v_1, v_2 \rangle_\kappa : s'$ whose last rule is $[Pair]$. Hence

$$\begin{array}{ll} & \emptyset \vdash v_1 : s_1 \\ & \emptyset \vdash v_2 : s_2 \\ \text{where} & s' = (s_1 \times s_2, \kappa) \end{array}$$

Now, applying rule $[Pair]$ to $\langle v_1, v_2 \rangle_{\kappa \bullet ir}$ gives:

$$\emptyset \vdash \langle v_1, v_2 \rangle_{\kappa \bullet ir} : (s_1 \times s_2, \kappa \bullet ir)$$

Since $(s_1 \times s_2, \kappa \bullet ir) = (s_1 \times s_2, \kappa) \bullet ir = s' \bullet ir$, and $s' \leq s$, Proposition A.1 implies $(s_1 \times s_2, \kappa \bullet ir) \leq s \bullet ir = s_a' \leq s_a$. Hence $\emptyset \vdash \langle v_1, v_2 \rangle_{\kappa \bullet ir} : s_a$ by $[Sub]$.

6. $(\mathbf{protect}_{ir}\ (\mathbf{inj}_i\ v)_\kappa) \rightarrow (\mathbf{inj}_i\ v)_{\kappa \bullet ir}$. Similar to the previous case.

7. $(\mathbf{protect}_{ir}\ (\lambda x : s_1.\ e)_\kappa) \rightarrow (\lambda x : s_1.\ e)_{\kappa \bullet ir}$. Similar to the previous case.

8. $(\mu f : s.\ e) \rightarrow e[(\lambda x : s_1.\ ((\mu f : s.\ e)\ x)_r)_{(r,ir)}/f]$, where $s = (s_1 \rightarrow s_2, (r, ir))$. Simple and hence omitted.

This concludes the case analysis and hence the proof. ■

Theorem A.5 (Subject Reduction) *Suppose* $\emptyset \vdash e : s$ *and* $e \rightarrow e'$. *Then* $\emptyset \vdash e' : s$.

Proof: Note that $e = E[e_1]$, where $e_1 \rightarrow e_2$ via one of the rules in Table 1, and $e' = E[e_2]$. A simple induction on evaluation contexts, using Lemma A.4, completes the proof. ■

A.4 Progress Theorem

Theorem A.6 (Progress) *Suppose $\emptyset \vdash e : s$ and e is not a value. Then there is a reduction $e \to e'$.*

Proof: Suppose, by way of contradiction, that there is no reduction of e. Then it must be the case that $e = E[e_0]$, $\emptyset \vdash e_0 : s_0$ for some s_0, and e_0 has one of the following forms:

1. $e_0 = (v\ v')_r$.
2. $e_0 = (\mathbf{proj}_i\ v)_r$.
3. $e_0 = (\mathbf{case}\ v\ \mathbf{of}\ \mathbf{inj}_1(x) \Rightarrow e_1 \mid \mathbf{inj}_2(x) \Rightarrow e_2)_r$.

We consider the first case and leave the others to the reader. Since e_0 is well-typed,

$$\begin{array}{l} s_0' \leq s_0 \\ \emptyset \vdash (v\ v')_r : s_0' \\ \emptyset \vdash v : (s_1 \to s_0', (r_2, ir_2)) \\ \emptyset \vdash v : s_1 \end{array}$$

and $r_2 \sqsubseteq r$, Note that v must have the form $(\lambda x : s_1.\ e_0')_{(r_1, ir_1)}$, since it has a functional type (this can be seen by an easy induction on typing derivations).

This gives us enough room to complete the proof. By rule $[Abs]$, we know

$$\begin{array}{l} (s_1 \to s_0', (r_1, ir_1)) \leq (s_1 \to s_0', (r_2, ir_2)) \\ \emptyset \vdash_{ir} (\lambda x : s_1.\ e_0')_{(r_1, ir_1)} : (s_1 \to s_0', (r_1, ir_1)) \\ x : s_1 \vdash_{ir'} e_0' : s_0' \end{array}$$

It follows that $r_1 \sqsubseteq r_2 \sqsubseteq r$, and so the application reduction rule applies. This contradicts the initial assumption that there is no reduction of e, so there must be a reduction of the term. ■

A.5 Noninterference

We can assign a standard denotational semantics to the language by adopting the partial function model of [14]. Define the meaning of a type expression s, denoted $[\![s]\!]$, by

$$\begin{array}{rcl} [\![(\mathbf{unit}, (r, ir))]\!] & = & \mathit{unit} \\ [\![(s + t, (r, ir))]\!] & = & ([\![s]\!] + [\![t]\!]) \\ [\![(s \times t, (r, ir))]\!] & = & ([\![s]\!] \times [\![t]\!]) \\ [\![(s \to t, (r, ir))]\!] & = & ([\![s]\!] \to_p [\![t]\!]) \end{array}$$

where $(D \to_p E)$ is the set of partial continuous functions from D to E. Note that this semantics ignores the security properties.

The meaning of terms is a *partial* function. If $\Gamma = x_1 : t_1, ..., x_n : t_n$ is a typing context then $[\![\Gamma]\!] = [\![t_1]\!] \times \ldots \times [\![t_n]\!]$. (The order is not important here, as we could rely on some fixed ordering of $x_i : t_i$ pairs.) In the case that Γ is empty, $[\![\Gamma]\!]$ is the unit object *unit*. For an environment $\eta \in |[\![\Gamma]\!]|$, write $\eta(x)$ for the projection to the component corresponding to variable x, and $\eta[x \mapsto d]$ for the environment in which the x component is extended (or overwritten) to d. The definition of the meaning function on terms, like that of types, ignores the security properties; similar definitions may be found in, say, [14, 16]. The model is adequate for observing the final answers of programs:

Theorem A.7 (Plotkin) *For any typing judgement $\emptyset \vdash M : s$ and any environment η, $[\![\emptyset \vdash M : s]\!]\eta$ is defined iff $M \to^* v$ for some value v.*

Our proof of noninterference uses logical relations (see [11] for other uses of logical relations). Define $\mathcal{R}$ to be a family of relations indexed by secure types and indirect readers ir where

1. If $s = (t, (r, ir))$ and $ir \not\sqsubseteq ir'$, then $R^s_{ir'} = \{(d, e) \mid d, e \in [\![s]\!]\}$.

2. If $t = (\mathbf{unit}, (r, ir))$ and $ir \sqsubseteq ir'$, then $R^s_{ir'} = \{(\top, \top)\}$.

3. If $t = (s_1 + s_2, (r, ir))$ and $ir \sqsubseteq ir'$, then $R^s_{ir'} = \{(inj_i(d), inj_i(e)) \mid (d, e) \in R^{s_i}_{ir'}, i = 1, 2\}$.

4. If $s = (s_1 \times s_2, (r, ir))$ and $ir \sqsubseteq ir'$, then $R^s_{ir'} = \{(\langle d_1, e_1\rangle, \langle d_2, e_2\rangle) \mid (d_i, e_i) \in R^{s_i \bullet ir}_{ir'}\}$.

5. If $s = (s_1 \to s_2, (r, ir))$ and $ir \sqsubseteq ir'$, then $R^s_{ir'} = \{(f, g) \mid$ if $(d, e) \in R^{s_1}_{ir'}$, then $(f(d), g(e)) \in \overline{R^{s_2 \bullet ir}_{ir'}}\}$.

Here, $(f(d), g(e)) \in \overline{R^s_{ir'}}$ means that if $f(d)$ and $g(e)$ are defined, then $(f(d), g(e)) \in R^s_{ir'}$. Intuitively, the ir' index specifies the secrecy group of an indirect reader of group ir'. When the secrecy group ir' is not above the group of the type itself, the indirect reader does not have permission to find out any information about the value.

Proposition A.8 *1. Each R^s_{ir} is directed complete,* i.e.*, if $\{(d_i, e_i) \mid i \in I\} \subseteq R^s_{ir}$ is a directed set, then $(\bigsqcup d_i, \bigsqcup e_i) \in R^s_{ir}$.*

2. If $s \leq s'$, then $R^s_{ir} \subseteq R^{s'}_{ir}$.

Proof: By induction on types. ■

Theorem A.9 *Suppose $\Gamma \vdash e : s$ and $\eta, \eta' \in [\![\Gamma]\!]$. Suppose that for all $x : s' \in \Gamma$, $(\eta(x), \eta'(x)) \in R^{s'}_{ir'}$. Then $([\![\Gamma \vdash e : s]\!]\eta, [\![\Gamma \vdash e : s]\!]\eta') \in \overline{R^s_{ir'}}$.*

Proof: By induction on the proof of $\Gamma \vdash e : s$.

1. $\Gamma, x : s \vdash x : s$. Follows easily from the hypothesis.

2. $\Gamma \vdash ()_\kappa : (\mathbf{unit}, \kappa)$. Trivial.

3. $\Gamma \vdash (\lambda x : s_1.\ M)_\kappa : (s_1 \to s_2, \kappa)$, where $\Gamma, x : s_1 \vdash M : t$. Suppose $\kappa = (r, ir)$ and $(d, e) \in R^{s_1}_{ir'}$. By induction,
$$([\![\Gamma, x : s_1 \vdash M : s_2]\!]\eta[x \mapsto d], [\![\Gamma, x : s_1 \vdash M : s_2]\!]\eta'[x \mapsto e]) \in \overline{R^{s_2}_{ir'}}.$$
By Proposition A.8,
$$([\![\Gamma, x : s_1 \vdash M : s_2]\!]\eta[x \mapsto d], [\![\Gamma, x : s_1 \vdash M : s_2]\!]\eta'[x \mapsto e]) \in \overline{R^{s_2 \bullet ir}_{ir'}}.$$
Note that
$$[\![\Gamma, x : s_1 \vdash M : s_2]\!]\eta[x \mapsto d] = [\![\Gamma \vdash (\lambda x : s_1.\ M)_\kappa : (s_1 \to s_2, \kappa)]\!]\eta\ (d)$$
and similarly for the other expression. Thus,
$$([\![\Gamma \vdash (\lambda x : s_1.\ M)_\kappa : (s_1 \to s_2, \kappa)]\!]\eta, [\![\Gamma \vdash (\lambda x : s_1.\ M)_\kappa : (s_1 \to s_2, \kappa)]\!]\eta') \in \overline{R^{(s_1 \to s_2, \kappa)}_{ir'}}.$$

4. $\Gamma \vdash (M\ N)_{r'} : s_2 \bullet ir$, where $\Gamma \vdash M : (s_1 \to s_2, (r, ir))$, $\Gamma \vdash N : s_1$, and $r \sqsubseteq r'$. Let $\kappa = (r, ir)$. By induction,
$$(f, g) = ([\![\Gamma \vdash M : (s_1 \to s_2, \kappa)]\!]\eta, [\![\Gamma \vdash M : (s_1 \to s_2, \kappa)]\!]\eta') \in \overline{R^{(s_1 \to s_2, \kappa)}_{ir'}}$$
$$(d, e) = ([\![\Gamma \vdash N : s_1]\!]\eta, [\![\Gamma \vdash N : s_1]\!]\eta') \in \overline{R^{s_1}_{ir'}}$$
There are two cases:

   (a) $ir \sqsubseteq ir'$. Then $(f(d), g(e)) \in \overline{R^{s_2 \bullet ir}_{ir'}}$ directly from the definition of $R^{(s_1 \to s_2, \kappa)}_{ir'}$.

   (b) $ir \not\sqsubseteq ir'$. Suppose $s_2 = (t, (r'', ir''))$. Note that $s_2 \bullet ir = (t, (r'' \sqcup ir, ir'' \sqcup ir))$. It follows that $(ir'' \sqcup ir) \not\sqsubseteq ir'$, and so $R^{s \bullet ir}_{ir'}$ is the complete relation. Therefore, $(f(d), g(e)) \in \overline{R^{s_2 \bullet ir}_{ir'}}$.

5. $\Gamma \vdash \langle M, N\rangle_\kappa : (s_1 \times s_2, \kappa)$, where $\Gamma \vdash M : s_1$ and $\Gamma \vdash N : s_2$. Let $\kappa = (r, ir)$. By induction and the definition of R,

$$(d_1, d_2) = (\llbracket \Gamma \vdash M : s_1 \rrbracket \eta, \llbracket \Gamma \vdash M : s_1 \rrbracket \eta') \in \overline{R^{s_1}_{ir'}}$$
$$(e_1, e_2) = (\llbracket \Gamma \vdash N : s_2 \rrbracket \eta, \llbracket \Gamma \vdash N : s_2 \rrbracket \eta') \in \overline{R^{s_2}_{ir'}}$$

By Proposition A.8, $(d_1, d_2) \in \overline{R^{s_1 \bullet ir}_{ir'}}$ and $(e_1, e_2) \in \overline{R^{s_2 \bullet ir}_{ir'}}$. It follows that

$$(\langle d_1, e_1\rangle, \langle d_2, e_2\rangle) \in \overline{R^{(s_1 \times s_2, \kappa)}_{ir'}}$$

as desired.

6. $\Gamma \vdash (\mathbf{proj}_i\ M)_{r'} : s_i \bullet ir$, where $\Gamma \vdash M : s$, $s = (s_1 \times s_2, r, ir)$ and $r \sqsubseteq r'$. By induction,

$$(d_1, d_2) = (\llbracket \Gamma \vdash M : s \rrbracket \eta, \llbracket \Gamma \vdash M : s \rrbracket \eta') \in \overline{R^{s}_{ir'}}$$

There are two cases.

   (a) $ir \sqsubseteq ir'$. If both d_1 and d_2 are defined, by the definition of the relation $R^s_{ir'}$, $d_j = \langle e_j, f_j\rangle$ for $(e_1, e_2) \in R^{s_1 \bullet ir}_{ir'}$ and similarly for (f_1, f_2). It thus follows that

$$(\llbracket \Gamma \vdash (\mathbf{proj}_i\ M)_{r'} : s_i \bullet ir \rrbracket \eta, \llbracket \Gamma \vdash (\mathbf{proj}_i\ M)_{r'} : s_i \bullet ir \rrbracket \eta') \in \overline{R^{s_i \bullet ir}_{ir'}}$$

   as desired.

   (b) $ir \not\sqsubseteq ir'$. Suppose $s_i = (t, (r'', ir''))$. Note that $s_i \bullet ir = (t, (r'' \sqcup ir, ir'' \sqcup ir))$. It follows that $(ir'' \sqcup ir) \not\sqsubseteq ir'$, and so $R^{s_i \bullet ir}_{ir'}$ is the complete relation. Therefore, $(proj_i(d_1), proj_i(d_2)) \in \overline{R^{s_i \bullet ir}_{ir'}}$.

7. $\Gamma \vdash (\mathbf{inj}_i\ M)_\kappa : (s_1 + s_2, \kappa)$, where $\Gamma \vdash M : s_i$. Let $\kappa = (r, ir)$. By induction and the definition of R,

$$(d_1, d_2) = (\llbracket \Gamma \vdash M : s_i \rrbracket \eta, \llbracket \Gamma \vdash M : s_i \rrbracket \eta') \in \overline{R^{s_i}_{ir'}}.$$

It follows that

$$(inj_i(d_1), inj_i(d_2)) \in \overline{R^{(s_1 + s_2, \kappa)}_{ir'}}$$

as desired.

8. $\Gamma \vdash P : s \bullet ir$, where $P = (\mathbf{case}\ M\ \mathbf{of}\ \mathbf{inj}_1(x) \Rightarrow N_1 \mid \mathbf{inj}_2(x) \Rightarrow N_2)_{r'}$, $\Gamma \vdash M : (s_1 + s_2, \kappa)$, $\kappa = (r, ir)$, $\Gamma, x : s_i \vdash N_i : s$, and $r \sqsubseteq r'$. By induction,

$$(d_1, d_2) = (\llbracket \Gamma \vdash M : (s_1 + s_2, \kappa) \rrbracket \eta, \llbracket \Gamma \vdash M : (s_1 + s_2, \kappa) \rrbracket \eta') \in \overline{R^{(s_1 + s_2, \kappa)}_{ir'}}$$

There are two cases to consider.

   (a) $ir \sqsubseteq ir'$. If both d_1, d_2 are defined, then by the definition of $R^{(s_1 + s_2, \kappa)}_{ir'}$, each $d_j = (inj_i\ e_j)$ for some i and $(e_1, e_2) \in R^{s_i}_{ir'}$. By induction,

$$(\llbracket \Gamma, x : s_i \vdash N_i : s \rrbracket \eta[x \mapsto e_1], \llbracket \Gamma, x : s_i \vdash N_i : s \rrbracket \eta'[x \mapsto e_2]) \in \overline{R^{s}_{ir'}}$$

   By Proposition A.8, $(\llbracket \Gamma, x : s_i \vdash N_i : s \rrbracket \eta[x \mapsto e_1], \llbracket \Gamma, x : s_i \vdash N_i : s \rrbracket \eta'[x \mapsto e_2]) \in \overline{R^{s \bullet ir}_{ir'}}$. Thus,

$$(\llbracket \Gamma \vdash P : s \bullet ir \rrbracket \eta, \llbracket \Gamma \vdash P : s \bullet ir \rrbracket \eta') \in \overline{R^{s \bullet ir}_{ir'}}$$

   as desired.

   (b) $ir \not\sqsubseteq ir'$. Suppose $s = (t, (r_1, ir_1))$. Then $(ir \sqcup ir_1) \not\sqsubseteq ir'$, and hence $R^{s \bullet ir}_{ir'}$ is the complete relation. Thus,

$$(\llbracket \Gamma \vdash P : s \bullet ir \rrbracket \eta, \llbracket \Gamma \vdash P : s \bullet ir \rrbracket \eta') \in \overline{R^{s \bullet ir}_{ir'}}$$

9. $\Gamma \vdash M : s$, where $\Gamma \vdash M : s'$ and $s' \leq s$. This case follows from the induction hypothesis and the fact that $R^{s'}_{ir'} \subseteq R^{s}_{ir'}$, which is part of Proposition A.8.

10. $\Gamma \vdash (\mu f : s.\, e) : s$, where $s = (s_1 \to s_2, \kappa)$. Let $g_1(x) = [\![\Gamma, f : s \vdash e : s]\!]\eta[f \mapsto x]$ and $g_2(x) = [\![\Gamma, f : s \vdash e : s]\!]\eta[f \mapsto x]$. We claim that for all $n \geq 0$, $(g_1^n(\bot), g_2^n(\bot)) \in \overline{R^s_{ir'}}$, from whence it follows by Proposition A.8 that
$$([\![\Gamma \vdash (\mu f : s.\, e) : s]\!]\eta, [\![\Gamma \vdash (\mu f : s.\, e) : s]\!]\eta') \in \overline{R^s_{ir'}}.$$
Proceed by induction on n. For the base case, it is easy to see that if $\bot$ is the everywhere undefined function, $(g_1^0(\bot), g_2^0(\bot)) = (\bot, \bot) \in \overline{R^s_{ir'}}$. For the inductive case, suppose $(g_1^n(\bot), g_2^n(\bot)) \in \overline{R^s_{ir'}}$. If one is not defined, then $(g_1^{n+1}(\bot), g_2^{n+1}(\bot)) \in \overline{R^s_{ir'}}$. If both are defined, then
$$(g_1^{n+1}(\bot), g_2^{n+1}(\bot)) = ([\![\Gamma, f : s \vdash e : s]\!]\eta[x \mapsto g_1^n(\bot)], [\![\Gamma, f : s \vdash e : s]\!]\eta'[x \mapsto g_2^n(\bot)]) \in \overline{R^s_{ir'}}$$
by induction, which completes the proof of the claim.

This completes the induction and hence the proof. ■

Suppose s is a type. Then s is **transparent at security property** κ if

1. $s = (\mathbf{unit}, \kappa')$ and $\kappa' \leq \kappa$;
2. $s = (s_1 + s_2, \kappa')$, $\kappa' \leq \kappa$, and s_1, s_2 are transparent at security property κ;
3. $s = (s_1 \times s_2, \kappa')$, $\kappa' \leq \kappa$, and s_1, s_2 are transparent at security property κ; or
4. $s = (s_1 \to s_2, \kappa')$, $\kappa' \leq \kappa$, and s_1, s_2 are transparent at security property κ.

$s = (t, \kappa)$ is **transparent** if s is transparent at κ.

Lemma A.10 *Suppose $s = (t, (r, ir))$ is a ground type transparent at (r', ir'). If $(f_1, f_2) \in R^s_{ir'}$, then $f = f'$.*

Proof: By induction on t. The base case, when $t = \mathbf{unit}$, is obvious. When $t = (s_1 + s_2, (r, ir))$, since $ir \sqsubseteq ir'$, it follows from the definition of $R^s_{ir'}$ that $f_j = (inj_i\ e_j)$ for some i and $(e_1, e_2) \in R^{s_i}_{ir'}$. By induction, $e_1 = e_2$. Thus, $f_1 = f_2$.

When $t = (s_1 \times s_2, (r, ir))$, it follows from the definition of $R^s_{ir'}$ that $f_j = \langle d_j, e_j \rangle$ and $(d_1, d_2) \in R^{s_1 \bullet ir}_{ir'}$ and $(e_1, e_2) \in R^{s_2 \bullet ir}_{ir'}$. Note that $s_1 \bullet ir = (t_1, (r_1, ir_1)) \bullet ir = (t_1, (r_1 \sqcup ir, ir_1 \sqcup ir))$ and similarly for $s_2 \bullet ir = (t_2, (r_2, ir_2)) \bullet ir$. Since $ir_1 \sqsubseteq ir$, $(ir_1 \sqcup ir) = ir \sqsubseteq ir'$, and similarly $(ir_2 \sqcup ir) \sqsubseteq ir'$. Thus, by induction, $d_1 = d_2$ and $e_1 = e_2$, which proves that $f_1 = f_2$ as desired. ■

Theorem A.11 (Noninterference) *Suppose $\emptyset \vdash e : (t, (r, ir))$, $\emptyset \vdash C[e] : (t', (r', ir'))$, t' is a ground, transparent type, and $ir \not\sqsubseteq ir'$. Then for all $\emptyset \vdash e' : (t, (r, ir))$, $C[e] \simeq C[e']$.*

Proof: To simplify notation, let **unit** stand for the least secure unit type $(\mathbf{unit}, (\bot, \bot))$ (with lowest security) and $() : \mathbf{unit}$ denote the least secure value of type **unit**. Consider the open term
$$y : (\mathbf{unit} \to s, (\bot, \bot)) \vdash C[(y_\bot\ ())_\bot] : s'$$
It is easy to see that this is a well-formed typing judgement. Consider any $\emptyset \vdash e_i : s$ for $i = 1, 2$. Let
$$d_i = [\![\emptyset \vdash (\lambda x : \mathbf{unit}.\, e_i)_{(\bot,\bot)} : (\mathbf{unit} \to s, (\bot, \bot))]\!].$$
It is easy to show that $(d_1, d_2) \in R^{(\mathbf{unit} \to s, (\bot,\bot))}_{ir'}$, since $ir \not\sqsubseteq ir'$. Let
$$f_i = [\![y : (\mathbf{unit} \to s, (\bot, \bot)) \vdash C[(y_\bot\ ())_\bot] : s']\!][x \mapsto d_i].$$
By Theorem A.9,
$$(f_1, f_2) \in \overline{R^{s'}_{ir'}}.$$
If f_1, f_2 are defined, then by Lemma A.10, $f_1 = f_2$. When f_i is defined, it is simple to show that there is a value v_i such that $f_i = [\![\emptyset \vdash v_i : s']\!]$. Since $v_1 \simeq v_2$, we are done. ■