A Core Calculus of Dependency

Martín Abadi
Systems Research Center
Compaq
ma@pa.dec.com

Anindya Banerjee
Stevens Institute of Technology
ab@cs.stevens-tech.edu

Nevin Heintze
Bell Laboratories
nch@bell-labs.com

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

Abstract

Notions of program dependency arise in many settings: security, partial evaluation, program slicing, and call-tracking. We argue that there is a central notion of dependency common to these settings that can be captured within a single calculus, the Dependency Core Calculus (DCC), a small extension of Moggi's computational lambda calculus. To establish this thesis, we translate typed calculi for secure information flow, binding-time analysis, slicing, and call-tracking into DCC. The translations help clarify aspects of the source calculi. We also define a semantic model for DCC and use it to give simple proofs of noninterference results for each case.

1 Introduction

Systems that incorporate aspects of program dependency arise in many different contexts. For example, type systems for secure information flow trace dependencies between outputs and inputs of a computation. These type systems are meant to guarantee secrecy and integrity. In the Secure Lambda (SLam) Calculus [13] and the while-program languages of Volpano *et al.* [31, 38], data may be labelled as "high security" or "low security", and the type system ensures that all computations that depend on high-security inputs yield high-security outputs, and conversely, that low-security outputs do not depend on high-security inputs. This independence property is often called the *noninterference property* [8, 9, 17] in the security literature: high-security data does not "interfere" with the calculation of low-security outputs. Fragments of the trust calculus [27] and JFlow [22, 23] also appear to satisfy the noninterference property (although this is not proved).

Program analyses such as slicing, call-tracking, and binding-time analysis are also based on dependency: the goal of these analyses is to compute a conservative approximation of the parts of a program that may contribute to the program's final result (and, more generally, its intermediate results). Correctness of these analyses is often expressed using properties analogous to noninterference. For instance, in slicing [36, 40], the aim is to determine those parts of a program that may contribute to the output; those parts that do not contribute can be replaced by any expression of the same type. In call-tracking [33, 34], we wish to determine the functions that may be called during evaluation; functions that are not called can be replaced by any function of the same type without affecting the final value. In binding-time analysis [5, 25], we wish to separate static from dynamic computations; dynamic values can be replaced by any expression of the same type without affecting the static results.

The similarity between secure information flow and other program analyses is striking, and raises a question: do these analyses share some common substrate? This paper provides one answer by constructing a general framework for type-based dependency analyses in higher-order programs. The framework is a calculus called the Dependency Core Calculus (DCC). We give a denotational semantics for DCC that formalizes the notion of noninterference. We then show how to translate a variety of calculi for security, slicing, binding-time analysis, and call-tracking into DCC in such a way that the noninterference results for the respective calculi are immediate corollaries of the generic results for DCC.

There are three advantages to this foundational approach. First, DCC gives us a way to compare dependency analyses. This idea relates back to Strachey's conception of denotational semantics as a tool for comparing languages [32]. Second, the translations themselves yield a check on type systems for dependency analysis. They help confirm some seemingly ad hoc decisions in some calculi and have uncovered some problems and incompletenesses in others. Third, general results about DCC yield simple noninterference proofs for the individual dependency analyses.

DCC is a simple extension of Moggi's computational lambda calculus [20]. Typically the computational lambda calculus has a single type constructor that is semantically associated with a monad. In DCC, this notion is extended to incorporate multiple monads, one for every level of a predetermined information lattice.

The use of the computational lambda calculus in describing dependency is somewhat surprising. Usually, the computational lambda calculus describes languages with side effects [20], or forms the basis of adding side effects like I/O to pure functional languages [15]. Dependency analyses, in contrast, do not fundamentally change the values being computed. Nevertheless, there is one common idea underlying both uses of the computational lambda calculus. In the case of Haskell, there is no way to compute a value using the I/O type constructor and pass that value to an expression of non-I/O type. Similarly, in information-flow systems, the test of a high-security boolean in an "if-then-else" requires that the branches of the conditional return high-security values. In both cases, the type rules of the computational lambda calculus enforce the necessary restriction.

The rest of the paper describes DCC, a semantic model of DCC, and six translations from type-based dependency analyses into DCC. Certain aspects of dependency analysis cannot be modelled in DCC; we discuss this further in the concluding discussion.

2 Commonality among Dependency Analyses

Before presenting the syntax and semantics of the core language DCC, we give two examples of dependency analyses: the SLam calculus and a slicing calculus.

2.1 Why the SLam Calculus is a Dependency Analysis

The SLam calculus [13] is a typed lambda calculus extended with security annotations for access control and information flow. To simplify the setting, we consider only the functional facet of the calculus with information flow, which corresponds to a fragment of the trust calculus of Ørbæk and Palsberg [27].

A type s is a pair consisting of a structural part, t, and a security annotation, κ, denoting information flow and ranging over elements in a security lattice $\mathcal{L}$, with least element L and greatest element H. For example, the type $(\texttt{bool}, L)$ denotes low-security booleans; similarly, $(\texttt{bool}, H)$ denotes high-security booleans. The type $((\texttt{bool}, H) \to (\texttt{bool}, L), L)$ denotes a low-security function that accepts a high-security integer and returns a low-security result. The terms and type rules of the language are given in Section 4.1. A simple example of a well-typed SLam term of type $((\texttt{bool}, H) \to (\texttt{bool}, L), L)$ is the constant function $(\lambda x : (\texttt{bool}, H). \texttt{true}_L)_L$. Note that all constructors in SLam are labelled with security annotations.

Since low-security computations should not depend on high-security data, the evaluation of an expression such as

$$\texttt{if}\ \texttt{true}_H\ \texttt{then}\ \texttt{true}_L\ \texttt{else}\ \texttt{false}_L$$

must not produce the low-security boolean $\texttt{true}_L$, since otherwise information about the high-security boolean is leaked to the low-security world.

The remedy is simple: whenever a constructor is destructed, we make the security annotation of the constructor flow to the annotation of the result. Specifically, the annotation of the result is the least upper bound of its original annotation and that of the constructor. This propagation of annotations is captured by the following dependency-calculus principle:

> At every elimination rule, properties (*e.g.*, security level, binding-time information, dependency annotation) of the destructed constructor are transferred to the result type of the expression.

This principle is fundamental to the design of the dependency calculus and to the rest of the paper. In the example, then, the result of $\texttt{if}\ \texttt{true}_H\ \texttt{then}\ \texttt{true}_L\ \texttt{else}\ \texttt{false}_L$ is the high-security boolean, $\texttt{true}_H$. The noninterference property is vacuously satisfied, since the result is a high-security boolean. More generally, if in the context $(x : (t, H))$ expression e has type $(\texttt{bool}, L)$, then noninterference says that e must not depend on the high-security variable x, and hence must be constant with respect to x.

2.2 Why the Slicing Calculus is a Dependency Analysis

In program slicing [36, 40], we seek the dependencies of a program—*i.e.*, those subterms of the program that may contribute to its output. For example, the slice of the application $((\lambda x. 3)\ 2)$ should contain only the function $\lambda x. 3$ and the constant 3, since the argument 2 does not contribute to the final result. To identify such subterms, we follow Abadi *et al.* [2] and use a labelled lambda calculus. We give a conservative approximation of the labelled operational semantics using a type system, whereas previous work by Biswas [3] employs set-based analysis.

The type system for slicing is similar to that of the SLam calculus. A type s is a pair consisting of a structural part, t, and a set of labels, κ, denoting slicing information. Note that the powerset of labels forms a complete lattice with empty set as least element and the set of all labels as greatest element. We give the complete type system in Section 4.2. A typing judgement $\Gamma \vdash e : (t, \kappa)$ means that, under the assumptions Γ, the expression e has type t and possible dependency κ. For instance, consider the example from above, where the constructors are all labelled:

$$(\lambda x : (\texttt{int}, \{n_2\}). 3_{n_1})_{n_0} (2_{n_2})$$

It is easy to see that the type of the function part of the application is $((\texttt{int}, \{n_2\}) \to (\texttt{int}, \{n_1\}), \{n_0\})$, so that the type of the whole term is $(\texttt{int}, \{n_0, n_1\})$. Thus the result of evaluating the term cannot depend on n_2.

Noninterference also holds in the slicing calculus: if under the assumption $(x : (t, \kappa_1))$, the expression e has type $(\texttt{int}, \kappa_2)$, where $\kappa_1 \not\sqsubseteq \kappa_2$ in the powerset lattice, then e must not depend on x.

3 Dependency Core Calculus

DCC is a minor extension of Moggi's computational lambda calculus [20]. Three features distinguish it from the computational lambda calculus. First, the calculus contains sum types and lifted types, as well as term recursion. Lifting allows us to model call-by-value calculi. Second, instead of having one type constructor T semantically associated with a monad, the calculus incorporates multiple type constructors T_ℓ, one for every element $\ell \in \mathcal{L}$ of a predetermined lattice $\mathcal{L}$. This idea was also considered by Wadler [39]. The lattice represents different grades of information. In the security setting, the least element usually stands for low security. Type constructors T_ℓ change the level of a type. For instance $T_H(\texttt{bool})$ describes high-security booleans. Third, the monad "bind" operation has a special typing rule that is explained later.

3.1 Syntax

The types of DCC are given by the grammar:

$$s ::= \texttt{unit} \mid (s+s) \mid (s \times s) \mid (s \to s) \mid s_\perp \mid T_\ell(s)$$

where ℓ ranges over elements of a predetermined lattice $\mathcal{L}$. The lifting operation on types, denoted $s_\perp$ in the syntax of types, induces a subset of types called the pointed types:

- $s_\perp$ is a pointed type;
- if s and t are pointed types, then $(s \times t)$ and $T_\ell(s)$ are pointed types; and
- if t is a pointed type, then $(s \to t)$ is a pointed type.

For a recent account of pointed types, see the paper by Howard [14] or Mitchell's text [18]. Similarly, the T_ℓ operation on types induces a subset of types called the types protected at level ℓ:

Table 1: Typing Rules for DCC.

[*Var*]	$\Gamma, x:s, \Gamma' \vdash x : s$	[*Unit*]	$\Gamma \vdash () : \texttt{unit}$
[*Lam*]	$\dfrac{\Gamma, x:s_1 \vdash e : s_2}{\Gamma \vdash (\lambda x:s_1.\, e) : (s_1 \to s_2)}$	[*App*]	$\dfrac{\Gamma \vdash e : (s_1 \to s_2) \quad \Gamma \vdash e' : s_1}{\Gamma \vdash (e\, e') : s_2}$
[*Pair*]	$\dfrac{\Gamma \vdash e_1 : s_1 \quad \Gamma \vdash e_2 : s_2}{\Gamma \vdash \langle e_1, e_2 \rangle : (s_1 \times s_2)}$	[*Proj*]	$\dfrac{\Gamma \vdash e : (s_1 \times s_2)}{\Gamma \vdash (\texttt{proj}_i\, e) : s_i}$
[*Inj*]	$\dfrac{\Gamma \vdash e : s_i}{\Gamma \vdash (\texttt{inj}_i e) : (s_1 + s_2)}$	[*Case*]	$\dfrac{\Gamma \vdash e : (s_1 + s_2) \quad \Gamma, x:s_i \vdash e_i : s}{\Gamma \vdash (\texttt{case}\, e\, \texttt{of}\, \texttt{inj}_1(x).\, e_1 \mid \texttt{inj}_2(x).\, e_2) : s}$
[*UnitM*]	$\dfrac{\Gamma \vdash e : s}{\Gamma \vdash (\eta_\ell\, e) : T_\ell(s)}$	[*BindM*]	$\dfrac{\Gamma \vdash e : T_\ell(s) \quad \Gamma, x:s \vdash e' : t}{\Gamma \vdash \texttt{bind}\, x = e\, \texttt{in}\, e' : t}$ t is protected at level ℓ
[*Lift*]	$\dfrac{\Gamma \vdash e : s}{\Gamma \vdash (\texttt{lift}\, e) : s_\perp}$	[*Seq*]	$\dfrac{\Gamma \vdash e : s_\perp \quad \Gamma, x:s \vdash e' : t}{\Gamma \vdash \texttt{seq}\, x = e\, \texttt{in}\, e' : t}$ t is pointed
[*Rec*]	$\dfrac{\Gamma, f:s \vdash e : s}{\Gamma \vdash (\mu f:s.\, e) : s}$ s is pointed		

- If $\ell \sqsubseteq \ell'$, then $T_{\ell'}(s)$ is protected at level ℓ;
- if s and t are protected at level ℓ, then $(s \times t)$ and $T_{\ell'}(t)$ are protected at level ℓ; and
- if t is protected at level ℓ, then $(s \to t)$ are protected at level ℓ.

The typing rules for DCC appear in Table 1. In all of the typing judgements in this paper, a typing environment Γ denotes a list of distinct variables with types. The rules for unit, function, product, and sum types are all standard, as is the rule for the monadic unit operation. The rule for monadic bind is nonstandard, using the concept of "protected at level ℓ" for the body; usually, the body must have type $T_\ell(s')$ for some s'. The model of the next section gives some justification for this rule. Finally, the rules [*Lift*] and [*Seq*] are just special cases of the monadic unit and bind operations for lifted types, and recursion is permitted only over pointed types.

The operational semantics for DCC is a call-by-name semantics. In particular, the term $(\eta_\ell\ e)$ reduces to e, and $(\texttt{bind}\, x = e\, \texttt{in}\, e')$ reduces to $e'[e/x]$, where $e[e'/x]$ denotes the capture-free substitution of e' for x in e. The rest of the operational semantics is standard and hence omitted.

3.2 Semantics

The model of DCC draws on ideas from other noninterference proofs [13, 19] which use Reynolds's concept of parametricity [28]. The method is easiest to explain with an example with high- and low-security booleans. A high-security computation can depend on a high-security input, but a low-security computation cannot. Our model explains the difference using "views" of the high-security booleans, where each view is captured by a binary relation and where computations must respect the relations. In this simple example, the high-security view is the diagonal relation (*i.e.*, x and y are related iff $x = y$), so that high-security computations can distinguish between the booleans. The low-security view, in contrast, is the everywhere true relation—that is, x is related to y for all x and y. Low-security computations can therefore not take advantage of the distinctions between the high-security booleans. Sabelfeld and Sands develop these ideas in a recent manuscript [30]; similar constructions appear in Nielson's work on strictness analysis [24].

We formalize these ideas via a category. Recall that a complete partial order (cpo) is a poset that contains least upper bounds for every directed subset; a cpo may or may not have a least element [18]. Recall also that a directed-complete relation is a relation that preserves least upper bounds of directed sets. Define the category $\mathcal{DC}$ (for *dependency category*) to be the category with

- OBJECTS An object A is a cpo $|A|$ and a family of directed-complete relations $R_{A,\ell}$ on A for every $\ell \in \mathcal{L}$.
- MORPHISMS A morphism $f : A \to B$ is a continuous function such that for any $(x,y) \in R_{A,\ell}$, $(f(x), f(y)) \in R_{B,\ell}$.

We use $Hom(A,B)$ to denote the set of morphisms from A to B.

The condition on morphisms is crucial. Consider, for instance, the lattice with two points $L \leq H$, let $B = \{\texttt{true}, \texttt{false}\}$ with the trivial ordering, and define the objects

$$\begin{aligned} boolH &= (B, R_L, R_H) \\ boolL &= (B, R'_L, R'_H) \end{aligned}$$

where R_L, R_H, R'_L, R'_H are relations on B. The relation R_L corresponds to a low-security viewer of a high-security boolean; such a viewer cannot distinguish between the booleans. Hence we choose R_L to be the everywhere true relation, $B \times B$. The relation R_H, in contrast, corresponds to a high-security viewer of a high-security boolean; such a viewer can distinguish between the booleans. Hence we choose R_H to be the diagonal relation on B. In a similar manner, we can choose both R'_L and R'_H to be the diagonal relation on B. Now, if $f : boolH \to boolL$ is a morphism, it must send arguments related by R_L to results related by R'_L. Since R_L is the everywhere true relation, for any $x, y \in B$, the pair (x,y) is in R_L. Thus, $(f(x), f(y)) \in R'_L$. In other words, $f(x) = f(y)$ for all $x, y \in B$. Therefore, a function mapping high-security booleans to low-security booleans must be a constant function. However, a relation need not be either the diagonal relation or the everywhere true relation.

The key property we need of $\mathcal{DC}$ is that it is a model of DCC (and therefore of the typed lambda calculus with products and coproducts). To establish this, we adapt standard results from categorical semantics [16] to show that $\mathcal{DC}$ is cartesian closed, has coproducts, and has a monad for each $\ell \in L$. (These results are necessary to justify the constructions in this paper; however, the reader unfamiliar with category theory can safely skip them.) More concretely, if A,B,C,D are objects and $f : A \to B, g : C \to D$:

- The unit object *unit* is defined by the poset $\{\top\}$ and the identity relations.

- Coproducts are given by

$$\begin{array}{rcl} |A+B| &=& |A|+|B| \\ R_{A+B,\ell} &=& \{(inl\, a, inl\, b) \mid (a,b) \in R_{A,\ell}\} \cup \\ && \{(inr\, a, inr\, b) \mid (a,b) \in R_{B,\ell}\} \\ (f+g)(x) &=& \left\{ \begin{array}{ll} inl(f(y)) & \text{if } x = inl(y) \\ inr(g(y)) & \text{if } x = inr(y) \end{array} \right. \end{array}$$

- Products are given by

$$\begin{array}{rcl} |A\times B| &=& |A|\times|B| \\ R_{A\times B,\ell} &=& \{(\langle a,b\rangle, \langle a',b'\rangle) \mid \\ && (a,a') \in R_{A,\ell}, (b,b') \in R_{B,\ell}\} \\ (f\times g)\langle x,y\rangle &=& \langle f(x), g(y)\rangle \end{array}$$

- Exponentiation is given by

$$\begin{array}{rcl} |A \Rightarrow B| &=& Hom(|A|,|B|) \\ R_{A\Rightarrow B,\ell} &=& \{(f,g) \mid \forall (a,a') \in R_{A,\ell},\, (f(a),f(a')) \in R_{B,\ell}\} \\ (f \Rightarrow g)(h : Hom(B,C))(x : A) &=& g(h(f(x))) \end{array}$$

- Lifting is given by

$$\begin{array}{rcl} |A_\perp| &=& \{(0,a) \mid a \in |A|\} \cup \{(1,\perp\} \\ R_{A_\perp,\ell} &=& \{((0,x),(0,y)) \mid \text{for all } (x,y) \in R_{A,\ell}\} \cup \\ && \{((1,\perp),(1,\perp))\} \\ (f_\perp)(x : A_\perp) &=& \left\{ \begin{array}{ll} (1,\perp) & \text{if } x = (1,\perp) \\ (0, f(y)) & \text{if } x = (0,y) \end{array} \right. \end{array}$$

  where $\{(0,a) \mid a \in |A|\}$ is ordered as in $|A|$, and $(1,\perp)$ is ordered below all other elements.

- The monads are given by

$$\begin{array}{rcl} |T_\ell(A)| &=& |A| \\ R_{T_\ell(A),\ell'} &=& \left\{ \begin{array}{ll} R_{A,\ell'} & \text{if } \ell \sqsubseteq \ell' \\ |A|\times|A| & \text{otherwise} \end{array} \right. \\ T_\ell(f : A \to B) &=& f \end{array}$$

  We also define the maps $\eta_\ell[A] : A \to T_\ell(A)$ and $\mu_\ell[A] : T_\ell(T_\ell(A)) \to T_\ell(A)$

$$\begin{array}{rcl} \eta_\ell[A](x) &=& x \\ \mu_\ell[A](x) &=& x \end{array}$$

  That is, both maps are based on the identity function. However, these morphisms are not the identities in the category, since they do not have the same domain and codomain.

The first four of these definitions are not surprising; Mitchell's text [18] gives a history of these definitions.

This structure gives us all the machinery needed to interpret the types and terms of DCC. We use $[\![s]\!]$ for the meaning of a type s in the category, and $[\![x_1 : s_1, \ldots, x_n : s_n \vdash e : s]\!] : [\![s_1 \times \ldots \times s_n]\!] \to [\![s]\!]$ for the meaning of a typing judgement. We omit the definitions of the meanings of terms since they are standard. Using induction on the definition of "pointed", we can show that:

Proposition 3.1 *If s is pointed, then $|[\![s]\!]|$ has a least element.*

Hence recursion can be interpreted via least-fixed points.

The monads T_ℓ give a way to change the level of a type, *e.g.*, as in $T_H(boolL) = boolH$. The operator T_ℓ changes the relations not above ℓ to the everywhere true relation. More generally, when a type is protected at level ℓ, views of that type at a level $\ell' \not\sqsupseteq \ell$ are the everywhere true relation.

Proposition 3.2 *If t is a type protected at level ℓ, and $\ell' \not\sqsupseteq \ell$, then $R_{[\![t]\!],\ell'} = |[\![t]\!]| \times |[\![t]\!]|$.*

4 Applications I: A Strong Version of Noninterference

In languages with recursion and some notion of dependency, there are often two ways to state the notion of noninterference. The first says that if a program terminates with an input and produces a result, then changing the input to a "related" input still causes the program to terminate and the result is related to the original result. This is a strong notion of noninterference. Some calculi, however, do not satisfy the strong property but do satisfy a weaker property: if two related inputs cause the program to terminate, the outputs are related. Under this property, related inputs may yield different convergence behavior.

In this section we study calculi with the strong version of noninterference. These include calculi based on call-by-name semantics, and they turn out to be easier to translate into DCC. In the next section, we study calculi that satisfy the weaker version of noninterference.

4.1 Call-by-name Functional SLam Calculus

Our first source calculus is the call-by-name, purely functional version of the SLam calculus; this calculus is essentially the trust calculus of Ørbæk and Palsberg [27] without the coercion from high to low security. Let $\mathcal{L}$ denote a join semilattice of security levels and let κ range over the levels of $\mathcal{L}$. The types are

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

and the typing rules appear in Table 2. In the typing rules, the operation

$$(t,\kappa) \bullet \kappa' = (t, \kappa \sqcup \kappa')$$

is used to increase the security level of a type. The symbol $\leq$ denotes the subtyping relation. The restriction of recursion to function types is not essential in a call-by-name context; the restriction here merely allows us to use the same type system for a call-by-value version below.

The operational semantics of this calculus deviates from the original operational semantics of the SLam calculus [13] in that arguments are passed by name rather than by value. Evaluation contexts are defined in the style of Felleisen [10] by the grammar

$$E ::= [\cdot] \mid (E\, e) \mid (\mathtt{proj}_i\, E) \mid (\mathtt{case}\, E\, \mathtt{of}\, \mathtt{inj}_1(x).\, e_1 \mid \mathtt{inj}_2(x).\, e_2)$$

and the local operational rules are

$$\begin{array}{rcl} ((\lambda x : s.\, e)_\kappa\, e') &\to& e[e'/x] \\ (\mathtt{proj}_i\, \langle e_1, e_2\rangle_\kappa) &\to& e_i, \quad i = 1,2 \\ (\mu f : s.\, e) &\to& e[(\mu f : s.\, e)/f] \\ (\mathtt{protect}_\kappa\, e) &\to& e \end{array}$$

$$(\mathtt{case}\, (\mathtt{inj}_i\, e)_\kappa\, \mathtt{of}\, \mathtt{inj}_1(x).\, e_1 \mid \mathtt{inj}_2(x).\, e_2) \to e_i[e/x]\ i = 1,2$$

Table 2: Typing Rules for the Functional SLam Calculus.

[*Var*]	$\Gamma, x : s, \Gamma' \vdash x : s$	[*Unit*]	$\Gamma \vdash ()_\kappa : (\texttt{unit}, \kappa)$
[*Sub*]	$\dfrac{\Gamma \vdash e : s \quad s \leq s'}{\Gamma \vdash e : s'}$	[*Rec*]	$\dfrac{\Gamma, f : s \vdash e : s}{\Gamma \vdash (\mu f : s.\, e) : s}$ s is a function type
[*Lam*]	$\dfrac{\Gamma, x : s_1 \vdash e : s_2}{\Gamma \vdash (\lambda x : s_1.\, e)_\kappa : (s_1 \to s_2, \kappa)}$	[*App*]	$\dfrac{\Gamma \vdash e : (s_1 \to s_2, \kappa) \quad \Gamma \vdash e' : s_1}{\Gamma \vdash (e\, e') : s_2 \bullet \kappa}$
[*Pair*]	$\dfrac{\Gamma \vdash e_1 : s_1 \quad \Gamma \vdash e_2 : s_2}{\Gamma \vdash \langle e_1, e_2 \rangle_\kappa : (s_1 \times s_2, \kappa)}$	[*Proj*]	$\dfrac{\Gamma \vdash e : (s_1 \times s_2, \kappa)}{\Gamma \vdash (\texttt{proj}_i\, e) : s_i \bullet \kappa}$
[*Inj*]	$\dfrac{\Gamma \vdash e : s_i}{\Gamma \vdash (\texttt{inj}_i e)_\kappa : (s_1 + s_2, \kappa)}$	[*Case*]	$\dfrac{\Gamma \vdash e : (s_1 + s_2, \kappa) \quad \Gamma, x : s_i \vdash e_i : s}{\Gamma \vdash (\texttt{case}\ e\ \texttt{of}\ \texttt{inj}_1(x).\, e_1 \mid \texttt{inj}_2(x).\, e_2) : s \bullet \kappa}$
[*Protect*]	$\dfrac{\Gamma \vdash e : s}{\Gamma \vdash (\texttt{protect}_\kappa\, e) : s \bullet \kappa}$	[*SubTrans*]	$\dfrac{s_1 \leq s_2 \quad s_2 \leq s_3}{s_1 \leq s_3}$
[*SubUnit*]	$\dfrac{\kappa \sqsubseteq \kappa'}{(\texttt{unit}, \kappa) \leq (\texttt{unit}, \kappa')}$	[*SubSum*]	$\dfrac{\kappa \sqsubseteq \kappa' \quad s_1 \leq s_1' \quad s_2 \leq s_2'}{((s_1 + s_2), \kappa) \leq ((s_1' + s_2'), \kappa')}$
[*SubProduct*]	$\dfrac{\kappa \sqsubseteq \kappa' \quad s_1 \leq s_1' \quad s_2 \leq s_2'}{((s_1 \times s_2), \kappa) \leq ((s_1' \times s_2'), \kappa')}$	[*SubFun*]	$\dfrac{\kappa \sqsubseteq \kappa' \quad s_1' \leq s_1 \quad s_2 \leq s_2'}{((s_1 \to s_2), \kappa) \leq ((s_1' \to s_2'), \kappa')}$

We write $e \Downarrow v$ when e rewrites to v and v cannot be rewritten.

The translation of the SLam calculus into DCC is straightforward. Types are translated into DCC by the following recursive definition, where $\dagger$ maps from types t (without a security level) into DCC types, and $*$ maps from types s (with a security level) into DCC types.

$$\begin{array}{rcl@{\qquad}rcl}
\texttt{unit}^\dagger & = & \texttt{unit}_\perp & (s_1 + s_2)^\dagger & = & (s_1^* + s_2^*)_\perp \\
(s_1 \times s_2)^\dagger & = & (s_1^* \times s_2^*) & (s_1 \to s_2)^\dagger & = & (s_1^* \to s_2^*) \\
(t, \kappa)^* & = & T_\kappa(t^\dagger) & & &
\end{array}$$

A SLam typing derivation of $\Gamma \vdash e : s$ is translated to a valid DCC derivation of $\Gamma^* \vdash e^* : s^*$ by the rules in Table 7. It is easy to check that every SLam typing derivation yields a DCC typing derivation by the translation. We can also prove the following correctness properties of the translation:

Theorem 4.1 (Adequacy) *If, according to Table 7,*

$$\emptyset \vdash e : (\texttt{unit}, \kappa) \Rightarrow \emptyset \vdash e^* : (\texttt{unit}, \kappa)^*$$

then $e \Downarrow v$ iff $[\![e^*]\!] \neq \perp$.

Theorem 4.2 (Noninterference) *Let κ_1 and κ_2 be any two elements of $\mathcal{L}$. Suppose $\kappa_1 \not\sqsubseteq \kappa_2$ and*

$$x : (t, \kappa_1) \vdash e : ((\texttt{unit}, \kappa_2) + (\texttt{unit}, \kappa_2), \kappa_2)$$

is derivable in the SLam type system. Then $(e[e'/x]) \Downarrow v$ iff $(e[e''/x]) \Downarrow v$.

Proof: The proof follows directly from the structure of $\mathcal{DC}$. We sketch the argument for the case where $\mathcal{L} = \{L, H\}$. Suppose

$$x : (t, H) \vdash e : ((\texttt{unit}, L) + (\texttt{unit}, L), L)$$

is derivable in the SLam type system. Applying the typing rule [*Lam*],

$$\emptyset \vdash (\lambda x : (t, H).\, e)_L : ((t, H) \to ((\texttt{unit}, L) + (\texttt{unit}, L), L), L)$$

Now, translating to DCC, we have

$$\emptyset \vdash (\lambda x : (t, H).\, e)_L^* : T_L(T_H(t^\dagger) \to T_L((T_L(\texttt{unit}_\perp) + T_L(\texttt{unit}_\perp))_\perp))$$

Let $f = [\![(\lambda x : (t, H).\, e)_L^*]\!]$. Since L is the least element of $\mathcal{L}$,

$$f \in [\![T_H(t^\dagger) \to (\texttt{unit}_\perp + \texttt{unit}_\perp)_\perp]\!]$$

Let $D = [\![T_H(t^\dagger)]\!]$ and $E = [\![(\texttt{unit}_\perp + \texttt{unit}_\perp)_\perp]\!]$. By the $\mathcal{DC}$ condition on morphisms, for all $l \in \mathcal{L}$ and for all $x, y \in D$,

$$x\, R_{D,l}\, y \quad \text{implies} \quad (f\, x)\, R_{E,l}\, (f\, y).$$

In the case l is L, $R_{D,L}$ is the everywhere true relation, and $R_{E,L}$ is the diagonal relation. Hence, for all $x, y \in D$, $(f\, x) = (f\, y)$. Thus,

$$\begin{array}{rcl}
[\![(e[e'/x])^*]\!] & = & [\![((\lambda x : (t, H).\, e)_L\, e')^*]\!] \\
& = & [\![((\lambda x : (t, H).\, e)_L\, e'')^*]\!] \\
& = & [\![(e[e''/x])^*]\!]
\end{array}$$

and so by adequacy, $(e[e'/x]) \Downarrow v$ iff $(e[e''/x]) \Downarrow v$. ■

This noninterference theorem is stated over one specific type only for readability; it extends to types not involving function types.

4.2 Slicing Calculus

The slicing calculus, introduced in Section 2, attempts to calculate which portions of a program may contribute to the final answer, and which definitely do not. To study slicing, we formulate a type-based slicing analysis. The types of the language are exactly the same as in the SLam calculus, except that κ ranges over sets of labels. The typing rules appear in Table 3. These rules resemble the SLam calculus rules, although the rules for value constructors are different.

Types are translated into DCC exactly as in the call-by-name, functional SLam calculus, and a typing judgement $\Gamma \vdash e : s$ is translated to a judgement of the form $\Gamma^* \vdash e^* : s^*$ by the rules in Table 8. The correctness properties are also the same as for the call-by-name, functional SLam calculus.

Table 3: Typing Rules for the Slicing Calculus (where subtyping is analogous to subtyping in the SLam calculus).

[*Var*]	$\Gamma, x : s, \Gamma' \vdash x : s$	[*Unit*]	$\Gamma \vdash ()_n : (\mathtt{unit}, \{n\})$
[*Sub*]	$\dfrac{\Gamma \vdash e : s \quad s \leq s'}{\Gamma \vdash e : s'}$	[*Rec*]	$\dfrac{\Gamma, f : s \vdash e : s}{\Gamma \vdash (\mu f : s. e) : s}$ s is a function type
[*Lam*]	$\dfrac{\Gamma, x : s_1 \vdash e : s_2}{\Gamma \vdash (\lambda x : s_1 . e)_n : (s_1 \to s_2, \{n\})}$	[*App*]	$\dfrac{\Gamma \vdash e : (s_1 \to s_2, \kappa) \quad \Gamma \vdash e' : s_1}{\Gamma \vdash (e\, e') : s_2 \bullet \kappa}$
[*Pair*]	$\dfrac{\Gamma \vdash e_1 : s_1 \quad \Gamma \vdash e_2 : s_2}{\Gamma \vdash \langle e_1, e_2 \rangle_n : (s_1 \times s_2, \{n\})}$	[*Proj*]	$\dfrac{\Gamma \vdash e : (s_1 \times s_2, \kappa)}{\Gamma \vdash (\mathtt{proj}_i\, e) : s_i \bullet \kappa}$
[*Inj*]	$\dfrac{\Gamma \vdash e : s_i}{\Gamma \vdash (\mathtt{inj}_i\, e)_n : (s_1 + s_2, \{n\})}$	[*Case*]	$\dfrac{\Gamma \vdash e : (s_1 + s_2, \kappa) \quad \Gamma, x : s_i \vdash e_i : s}{\Gamma \vdash (\mathtt{case}\, e\, \mathtt{of}\, \mathtt{inj}_1(x).\, e_1 \mid \mathtt{inj}_2(x).\, e_2) : s \bullet \kappa}$

Table 4: Typing Rules for the Binding-time Calculus.

[*Var*]	$\Gamma, x : s, \Gamma' \vdash x : s$	[*Unit*]	$\Gamma \vdash ()_\beta : (\mathtt{unit}, \beta)$
[*Sub*]	$\dfrac{\Gamma \vdash e : (\mathtt{unit}, \mathtt{sta})}{\Gamma \vdash e : (\mathtt{unit}, \mathtt{dyn})}$	[*Rec*]	$\dfrac{\Gamma, f : s \vdash e : s}{\Gamma \vdash (\mu f : s. e) : s}$ s is a function type
[*Lam*]	$\dfrac{\Gamma, x : s_1 \vdash e : s_2}{\Gamma \vdash (\lambda x : s_1 . e)_\beta : (s_1 \to s_2, \beta)}$	[*App*]	$\dfrac{\Gamma \vdash e : (s_1 \to s_2, \beta) \quad \Gamma \vdash e' : s_1}{\Gamma \vdash (e\, e') : s_2}$
[*Pair*]	$\dfrac{\Gamma \vdash e_1 : s_1 \quad \Gamma \vdash e_2 : s_2}{\Gamma \vdash \langle e_1, e_2 \rangle_\beta : (s_1 \times s_2, \beta)}$	[*Proj*]	$\dfrac{\Gamma \vdash e : (s_1 \times s_2, \beta)}{\Gamma \vdash (\mathtt{proj}_i\, e) : s_i}$
[*Inj*]	$\dfrac{\Gamma \vdash e : s_i}{\Gamma \vdash (\mathtt{inj}_i\, e)_\beta : (s_1 + s_2, \beta)}$	[*Case*]	$\dfrac{\Gamma \vdash e : (s_1 + s_2, \beta) \quad \Gamma, x : s_i \vdash e_i : s}{\Gamma \vdash (\mathtt{case}\, e\, \mathtt{of}\, \mathtt{inj}_1(x).\, e_1 \mid \mathtt{inj}_2(x).\, e_2) : s \bullet \beta}$

4.3 Binding-Time Calculus

The goal of binding-time analysis is to annotate a program with *binding times* and *specialization directives* [12]. The binding times specify when data is available. For instance, if there are only two binding times, static and dynamic, then static denotes "known at specialization-time" and dynamic denotes "known at run-time". Binding times are used to specify specialization directives: if an expression has static binding time, then it is *eliminable*, *i.e.*, can be reduced at compile time. If an expression has dynamic binding time, then it is *residual*, *i.e.*, it cannot be reduced at compile time.

Hatcliff and Danvy define one binding-time type system, focused on the computational lambda calculus [12]. Under their system, if in a dynamic context Γ_d an expression e of type `int` is mapped by the analysis to an annotated term w with annotation `sta` (for static), then w and e must be identical and must be equivalent to some integer constant n [12, Lemma 2]. This property is exactly noninterference: static data cannot rely on dynamic data.

Implicit in the Hatcliff-Danvy type system is a restriction on the structure of types. This restriction can be made explicit by defining a notion of well-formedness of types [26, 35]. For example, if `dyn` denotes dynamic binding-time with $\mathtt{sta} \leq \mathtt{dyn}$, the types $((\mathtt{int}, \mathtt{sta}) \to (\mathtt{int}, \mathtt{sta}), \mathtt{dyn})$ and $((\mathtt{int}, \mathtt{sta}) \times (\mathtt{int}, \mathtt{sta}), \mathtt{dyn})$ are ill-formed, whereas $((\mathtt{int}, \mathtt{dyn}) \to (\mathtt{int}, \mathtt{dyn}), \mathtt{sta})$ is well-formed. Using DCC, we can give a more generic account of this system. Specifically, we can show that the noninterference property is independent of the notion of well-formedness employed. The specific notion of well-formedness is motivated by engineering constraints varying from specializer to specializer. In summary, binding-time analysis can be viewed as a dependency calculus (à la DCC) in conjunction with a notion of well-formed types. The dependency captures a generic notion of whether or not a computation depends on dynamic inputs, and the well-formedness condition captures constraints imposed by the specializer.

We formalize these ideas in a source language similar to SLam, where the types are annotated with binding times $\mathtt{sta} \leq \mathtt{dyn}$. Types in the binding-time calculus are therefore

$$\begin{array}{rcl} t & ::= & \mathtt{unit} \mid (s + s) \mid (s \times s) \mid (s \to s) \\ s & ::= & (t, \beta) \\ \beta & ::= & \mathtt{sta} \mid \mathtt{dyn} \end{array}$$

The well-formed types [35] are a subset of the types defined as follows:

- $(\mathtt{unit}, \beta)$ and $(s_1 + s_2, \beta)$ are well-formed.
- $((t_1, \beta_1)\ op\ (t_2, \beta_2), \beta)$ is well-formed iff (t_1, β_1) and (t_2, β_2) are well-formed and $\beta \leq \beta_i$, where $op = \times, \to$.

The typing rules are given in Figure 4. The judgement $\Gamma \vdash e : s$ means that under assumptions Γ, expression e has the well-formed type s. Note that the well-formedness restriction on types obviates the need for $\bullet$ in the elimination rules *App* and Proj, since $(t, \beta) \bullet \beta' = (t, \beta)$ when $\beta' \leq \beta$.

The binding-time calculus can be translated into DCC. Types are translated into DCC in the same way as in the call-by-name, functional SLam calculus. A typing judgement $\Gamma \vdash e : s$ is translated

Table 5: Typing Rules for the Smith-Volpano Calculus over Booleans.

[*Var*]	$\Gamma_H;\Gamma_L \vdash x : \tau \quad \text{if } x \in \Gamma_\tau$	[*Const*]	$\Gamma_H;\Gamma_L \vdash k : \tau \quad k = \texttt{true} \text{ or } \texttt{false}$
[*Skip*]	$\Gamma_H;\Gamma_L \vdash \texttt{skip} : \tau\ \texttt{cmd}$	[*Sub*]	$\dfrac{\Gamma_H;\Gamma_L \vdash e : \tau \quad \tau \leq \tau'}{\Gamma_H;\Gamma_L \vdash e : \tau'}$
[*Assign*]	$\dfrac{\Gamma_H;\Gamma_L \vdash e : \tau \quad x \in \Gamma_\tau}{\Gamma_H;\Gamma_L \vdash (x := e) : \tau\ \texttt{cmd}}$	[*Seq*]	$\dfrac{\Gamma_H;\Gamma_L \vdash e : \tau\ \texttt{cmd} \quad \Gamma_H;\Gamma_L \vdash e' : \tau\ \texttt{cmd}}{\Gamma_H;\Gamma_L \vdash (e;e') : \tau\ \texttt{cmd}}$
[*If*]	$\dfrac{\Gamma_H;\Gamma_L \vdash e : \tau \quad \Gamma_H;\Gamma_L \vdash c_i : \tau\ \texttt{cmd}}{\Gamma_H;\Gamma_L \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \tau\ \texttt{cmd}}$	[*While*]	$\dfrac{\Gamma_H;\Gamma_L \vdash e : L \quad \Gamma_H;\Gamma_L \vdash c : L\ \texttt{cmd}}{\Gamma_H;\Gamma_L \vdash \texttt{while}\ e\ \texttt{do}\ c : L\ \texttt{cmd}}$

to a judgement of the form $\Gamma^* \vdash e^* : s^*$ by the rules in Table 9. The correctness properties are the same as for the call-by-name, functional SLam calculus.

4.4 Smith-Volpano Calculus

The Smith-Volpano calculus [31] is a simple language of while-programs, modified so that the types keep track of the security levels of variables and commands. Just as in the SLam calculus, the type system prevents high-security inputs from influencing low-security outputs. The translation of the Smith-Volpano calculus to DCC, however, looks very different from translations of SLam, the slicing calculus, and the binding-time calculus. Part of this difference arises from the difference between imperative and functional languages. On a deeper level, some of the subtleties of pointed types in DCC are useful in the translation.

Types in the Smith-Volpano calculus are divided into data types τ and phrase types ρ:

$$\begin{array}{rcl} \tau & ::= & L \mid H \\ \rho & ::= & \tau \mid \tau\ \texttt{cmd} \end{array}$$

When L or H is used in a phrase type, it is the type of storage cells that hold values of type L or H. The subtyping relation, used in the typing rules, is based on the primitive relations $L \leq H$ and $H\ \texttt{cmd} \leq L\ \texttt{cmd}$.

The typing rules for the calculus appear in Table 5. In order to keep the translation to DCC simple, we modify the original type rules [31] in two ways. First, variables have types L or H; variables of command type, possible in the original Smith-Volpano calculus, appear to have no use. Typing contexts are consequently split into two parts, Γ_H and Γ_L, containing the sets of high and low variables respectively; the type contexts are just lists of variables because of this split. Second, the implicit data type is boolean instead of integer. In other words, L is the type of low-security booleans and H is the type of high-security booleans. We make this simplification only for expository purposes, because there is no direct way of encoding the integer type in DCC. To extend the encoding to the original calculus, we could either directly add an integer type to DCC, whose semantic domain would be the flat integers, or add recursive types to DCC so that one could represent the integers as a type expression. Both changes would complicate DCC, and essentially no new difficulties arise with integers.

The operational semantics of the language uses a state, *i.e.*, a map σ from variables to $\{\texttt{true}, \texttt{false}\}$. There are two forms of judgement in the operational semantics. A judgement of the form $(c,\sigma) \to \sigma'$, where c is a command, denotes a computation that terminates in state σ'. A judgement of the form $(c,\sigma) \to (c',\sigma')$ denotes a computation that has not halted yet; the command to be run next is c'. The following rules define the operational semantics:

$$(\texttt{skip},\sigma) \to \sigma$$

$$\frac{(e\ \sigma) = v}{(x := e,\sigma) \to \sigma[x \mapsto v]} \qquad \frac{(e\ \sigma) = \texttt{false}}{(\texttt{while}\ e\ \texttt{do}\ c,\sigma) \to \sigma}$$

$$\frac{(c_1,\sigma) \to (c_1',\sigma')}{((c_1;c_2),\sigma) \to ((c_1';c_2),\sigma')} \qquad \frac{(c_1,\sigma) \to \sigma'}{((c_1;c_2),\sigma) \to (c_2,\sigma')}$$

$$\frac{(e\ \sigma) = \texttt{false}}{(\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2,\sigma) \to (c_2,\sigma)}$$

$$\frac{(e\ \sigma) = \texttt{true}}{(\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2,\sigma) \to (c_1,\sigma)}$$

$$\frac{(e\ \sigma) = \texttt{true}}{(\texttt{while}\ e\ \texttt{do}\ c,\sigma) \to ((c;\texttt{while}\ e\ \texttt{do}\ c),\sigma)}$$

We use $\to^*$ for the reflexive, transitive closure of $\to$.

Two observations about the calculus are in order. First, phrases of type $(H\ \texttt{cmd})$ never modify variables of type L. Thus, a phrase of type $(H\ \texttt{cmd})$ is a function from a state to the portion of the state representing high variables. Low commands, in contrast, can modify high and low variables. Second, `while` loops may include only low expressions and low commands. Without this restriction, the type system does not satisfy the strong noninterference property. Indeed, concurrency can be used to leak information [31]. From the restriction on `while` loops, it follows that only low commands may diverge.

The translation of the Smith-Volpano calculus into DCC depends on these two observations. We define the type `bool` to be the DCC type $(\texttt{unit} + \texttt{unit})$, and let

$$\begin{array}{rcl} SV(L) & = & \texttt{bool} \\ SV(H) & = & T_H(\texttt{bool}) \\ SV_\tau(z_1,\ldots,z_n) & = & \underbrace{SV(\tau) \times \ldots \times SV(\tau)}_{n} \\ SV(\Gamma_H,\Gamma_L,L) & = & SV_H(\Gamma_H) \times SV_L(\Gamma_L) \to \texttt{bool} \\ SV(\Gamma_H,\Gamma_L,H) & = & SV_H(\Gamma_H) \times SV_L(\Gamma_L) \to T_H(\texttt{bool}) \\ SV(\Gamma_H,\Gamma_L,H\ \texttt{cmd}) & = & SV_H(\Gamma_H) \times SV_L(\Gamma_L) \to SV_H(\Gamma_H) \\ SV(\Gamma_H,\Gamma_L,L\ \texttt{cmd}) & = & SV_H(\Gamma_H) \times SV_L(\Gamma_L) \to \\ & & (SV_H(\Gamma_H) \times SV_L(\Gamma_L))_\perp \end{array}$$

The translations of judgements are closed expressions in DCC, with the form

$$\Gamma_H;\Gamma_L \vdash e : \rho \Rightarrow e^* : SV(\Gamma_H,\Gamma_L,\rho)$$

where e ranges over expressions and commands, and e^* denotes the result of the translation of e. The complete translation is given in Table 10. For example, suppose the last rule used in the typing derivation is [*If*]. The judgement $\Gamma_H;\Gamma_L \vdash \texttt{if}\, e\, \texttt{then}\, c_1\, \texttt{else}\, c_2 : L\ \texttt{cmd}$ is translated as

$$\lambda\sigma.\,\texttt{if}\ (e^*\ \sigma)\ \texttt{then}\ (c_1^*\ \sigma)\ \texttt{else}\ (c_2^*\ \sigma)$$

where $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ is shorthand for $(\texttt{case}\, e\, \texttt{of}\, \texttt{inj}_1(x).\, e_1 \mid \texttt{inj}_2(x).\, e_2)$ for a fresh variable x. In contrast, the judgement $\Gamma_H;\Gamma_L \vdash \texttt{if}\, e\, \texttt{then}\, c_1\, \texttt{else}\, c_2 : H\ \texttt{cmd}$ is translated to

$$\lambda\sigma.\,\texttt{bind}\, v = (e^*\ \sigma)\ \texttt{in}\ \texttt{if}\ v\ \texttt{then}\ (c_1^*\ \sigma)\ \texttt{else}\ (c_2^*\ \sigma)$$

Notice the use of the bind in the last rule—the value of the expression e is a high-security boolean, and hence must be decomposed. Since both arms of the conditional are protected at level H, this part of the translation is well typed.

Suppose Γ_L is a set of variables; define $\sigma \sim_{\Gamma_L} \sigma'$ if for all $x \in \Gamma_L$, $\sigma(x) = \sigma'(x)$. We can prove the following theorems from the translation:

Theorem 4.3 (Adequacy) *Suppose* $(x_1,\ldots,x_n);(y_1,\ldots,y_k) \vdash c : L\ \texttt{cmd}$*. Then* $(c,\sigma) \to^* \sigma'$ *iff*

$$[\![c^*]\!]\ \langle\langle [\![\sigma(x_1)]\!],\ldots,[\![\sigma(x_n)]\!]\rangle, \langle [\![\sigma(y_1)]\!],\ldots,[\![\sigma(y_k)]\!]\rangle\rangle \neq \bot$$

Theorem 4.4 (Noninterference) *Suppose* $\Gamma_H;\Gamma_L \vdash c : L\ \texttt{cmd}$ *is derivable in the Smith-Volpano calculus, and* $\sigma \sim_{\Gamma_L} \sigma'$*. If* $(c,\sigma) \to^* \sigma_0$*, then* $(c,\sigma') \to^* \sigma_0'$ *and* $\sigma_0 \sim_{\Gamma_L} \sigma_0'$*. Dually, if* $(c,\sigma') \to^* \sigma_0'$*, then* $(c,\sigma) \to^* \sigma_0$ *and* $\sigma_0 \sim_{\Gamma_L} \sigma_0'$*.*

The proof of the noninterference theorem uses the semantic model of DCC, whereas the original operational proof uses a more detailed operational analysis [31].

5 Applications II: A Weaker Version of Noninterference

Not all calculi that track dependency satisfy the strong version of noninterference. For example, the original functional SLam calculus uses a call-by-value semantics rather than a call-by-name semantics. In this calculus, high-security inputs may affect the termination behavior—but not the outputs—of a low-security computation. An earlier version of the Smith-Volpano calculus, due to Volpano, Smith, and Irvine [38], also satisfies this weaker notion of noninterference; the strong version of noninterference seems to require the restriction of while-loops to low commands.

Unfortunately, it seems difficult to use DCC directly to model these languages. We must alter the syntax and semantics of DCC slightly. The main problem lies in the semantics of lifting. Consider, for instance, the meaning of the DCC type

$$T_H(\texttt{bool}) \to \texttt{bool}_\bot$$

where $\texttt{bool} = (\texttt{unit} + \texttt{unit})$ as before and $L \not\sqsupseteq H$. A function of this type must either map all elements to $\bot$ or all elements to a constant element of type bool, in essence obeying the strong version of noninterference. For the weaker version, we want the relation at $\texttt{bool}_\bot$ to relate $\bot$ to any element of bool, not just to $\bot$; the relation on non-$\bot$ elements should continue to be the diagonal relation.

To model the weaker notion, we use the same underlying category, and change the semantics of the lifting operator to have the relations

$$R_{A_\bot,\ell} = R_{A,\ell} \cup \{(\bot,\bot)\} \cup \{(x,\bot),(\bot,x) \mid x \in |A|\}$$

and change the definition of "protected" to include the clause

- If t is protected at level ℓ, then $t_\bot$ is protected at level ℓ.

We call the new language vDCC, since it is tuned to call-by-value (even though the operational semantics is still call-by-name). The meaning of $T_\ell(t_\bot)$ is now isomorphic to $(T_\ell(t))_\bot$, via the terms

$$\begin{aligned} f &= \lambda x : T_\ell(t_\bot).\,\texttt{bind}\, y = x\, \texttt{in}\, \texttt{seq}\, z = y\, \texttt{in}\, (\texttt{lift}\ (\eta_\ell\, z)) \\ g &= \lambda x : (T_\ell(t))_\bot.\,\texttt{seq}\, y = x\, \texttt{in}\, \texttt{bind}\, z = y\, \texttt{in}\, (\eta_\ell\ (\texttt{lift}\, z)) \end{aligned}$$

The terms are well typed because of the change in the definition of "protected."

We now describe two calculi satisfying the weak version of noninterference and translations of them into vDCC.

5.1 Call-by-value Functional SLam Calculus

The first application of vDCC is the call-by-value version of the functional SLam calculus in Section 4.1. The syntax and type-checking rules of the language are exactly the same as in the call-by-name setting, except that we require in recursion $(\mu f : s.\, e)$ that s has the form $(s_1 \to s_2, \kappa)$ where $s_2 = s_2 \bullet \kappa$. The main change is in the operational semantics, where the evaluation contexts become

$$\begin{aligned} v &::= () \mid (\lambda x : s.\, e) \mid (\texttt{inj}_i\ v) \mid \langle v,v\rangle \\ E &::= [\cdot] \mid (E\ e) \mid (v\ E) \mid (\texttt{inj}_i\ E) \mid \langle E,e\rangle \mid \langle v,E\rangle \mid \\ &\quad\ \ (\texttt{proj}_i\ E) \mid (\texttt{case}\, E\, \texttt{of}\, \texttt{inj}_1(x).\, e \mid \texttt{inj}_2(x).\, e') \end{aligned}$$

and rewrite rules become

$$\begin{array}{lll} ((\lambda x : s.\, e)_\kappa\ v) & \to & e[v/x] \\ (\texttt{proj}_i\ \langle v_1,v_2\rangle_\kappa) & \to & v_i \\ (\texttt{protect}_\kappa\ v) & \to & v \\ \multicolumn{3}{l}{(\texttt{case}\ (\texttt{inj}_i\ v)_\kappa\, \texttt{of}\, \texttt{inj}_1(x).\, e_1 \mid \texttt{inj}_2(x).\, e_2) \to e_i[v/x]} \\ \multicolumn{3}{l}{(\mu f : s.\, e) \to e[(\lambda x : s_1.\, (\mu f : s.\, e)\ x)_\kappa/f] \quad s = (s_1 \to s_2,\kappa)} \end{array}$$

Types are translated into vDCC as follows:

$$\begin{array}{rclrcl} \texttt{unit}^\dagger &=& \texttt{unit} & (s_1+s_2)^\dagger &=& (s_1^*+s_2^*) \\ (s_1 \times s_2)^\dagger &=& (s_1^* \times s_2^*) & (s_1 \to s_2)^\dagger &=& (s_1^* \to (s_2^*)_\bot) \\ (t,\kappa)^* &=& T_\kappa(t^\dagger) &&& \end{array}$$

Unlike in the call-by-name case, not every type is translated to a pointed type; function types, though, are guaranteed to be pointed. A typing judgement $\Gamma \vdash e : s$ is translated to a judgement of the form $\Gamma^* \vdash e^* : (s^*)_\bot$ by the rules in Table 11.

Theorem 5.1 (Adequacy) *If, according to Table 11,*

$$\emptyset \vdash e : (\texttt{unit},\kappa) \Rightarrow \emptyset \vdash e^* : (\texttt{unit},\kappa)^*$$

then $e \Downarrow v$ *iff* $[\![e^*]\!] \neq \bot$*.*

Theorem 5.2 (Noninterference) *Let* κ_1 *and* κ_2 *be any two elements of* $\mathcal{L}$*. Suppose* $\kappa_1 \not\sqsubseteq \kappa_2$ *and*

$$x : (t,\kappa_1) \vdash e : ((\texttt{unit},\kappa_2) + (\texttt{unit},\kappa_2),\kappa_2)$$

is derivable in the SLam type system. Then $(e[e'/x]) \Downarrow v$ *iff* $(e[e''/x]) \Downarrow v$*.*

Table 6: Typing Rules for the Call-tracking Calculus.

[*Var*]	$\Gamma, x : s, \Gamma' \vdash x : s, L$	[*Unit*]	$\Gamma \vdash () : \texttt{unit}, L$
[*Sub*]	$\dfrac{\Gamma \vdash e : s_1, \kappa \qquad s_1 \le s_2}{\Gamma \vdash e : s_2, \kappa'}\ \kappa \sqsubseteq \kappa'$	[*Rec*]	$\dfrac{\Gamma, f : s \vdash e : s, \kappa}{\Gamma \vdash (\mu f : s.\, e) : s, \kappa}\ s = (s_1 \xrightarrow{\kappa} s_2)$
[*Lam*]	$\dfrac{\Gamma, x : s_1 \vdash e : s_2, \kappa}{\Gamma \vdash (\lambda x : s_1.\, e)_n : (s_1 \xrightarrow{\{n\} \sqcup \kappa} s_2), L}$	[*App*]	$\dfrac{\Gamma \vdash e : (s_1 \xrightarrow{\kappa} s_2), \kappa_1 \quad \Gamma \vdash e' : s_1, \kappa_2}{\Gamma \vdash (e\, e') : s_2, \kappa \sqcup \kappa_1 \sqcup \kappa_2}$
[*Pair*]	$\dfrac{\Gamma \vdash e_1 : s_1, \kappa_1 \quad \Gamma \vdash e_2 : s_2, \kappa_2}{\Gamma \vdash \langle e_1, e_2 \rangle : (s_1 \times s_2), \kappa_1 \sqcup \kappa_2}$	[*Proj*]	$\dfrac{\Gamma \vdash e : (s_1 \times s_2), \kappa}{\Gamma \vdash (\texttt{proj}_i\, e) : s_i, \kappa}$
[*Inj*]	$\dfrac{\Gamma \vdash e : s_i, \kappa}{\Gamma \vdash (\texttt{inj}_i\, e) : (s_1 + s_2), \kappa}$	[*Case*]	$\dfrac{\Gamma \vdash e : (s_1 + s_2), \kappa \quad \Gamma, x : s_i \vdash e_i : s, \kappa'}{\Gamma \vdash (\texttt{case}\, e\, \texttt{of}\, \texttt{inj}_1(x).\, e_1 \mid \texttt{inj}_2(x).\, e_2) : s, \kappa \sqcup \kappa'}$

5.2 Call-tracking Calculus

Types in the call-tracking calculus [33, 34] are given by the grammar

$$s ::= \texttt{unit} \mid (s + s) \mid (s \times s) \mid (s \xrightarrow{\kappa} s).$$

where κ ranges over sets of labels. (These labels occur only on lambdas.) The typing rules appear in Table 6. A term is assigned with a type and an effect (a set of labels of lambdas that may be called). We use L to denote the least element of the lattice of sets of labels. The subtyping rule for function types is

$$\frac{s_1' \le s_1 \qquad s_2 \le s_2' \qquad \kappa \sqsubseteq \kappa'}{(s_1 \xrightarrow{\kappa} s_2) \le (s_1' \xrightarrow{\kappa'} s_2')}$$

The other subtyping rules are obvious and omitted.

Types are translated into vDCC as follows:

$$\begin{array}{rclrcl} \texttt{unit}^* & = & \texttt{unit} & (s_1 + s_2)^* & = & (s_1^* + s_2^*) \\ (s_1 \times s_2)^* & = & (s_1^* \times s_2^*) & (s_1 \xrightarrow{\kappa} s_2)^* & = & (s_1^* \to (T_\kappa(s_2^*))_\perp) \end{array}$$

A typing judgement $\Gamma \vdash e : s, \kappa$ is translated to a judgement of the form $\Gamma^* \vdash e^* : (T_\kappa(s^*))_\perp$ by the rules in Table 12.

Theorem 5.3 (Adequacy) *If, according to Table 12,*

$$\emptyset \vdash e : \texttt{unit}, \kappa \Rightarrow \emptyset \vdash e^* : (T_\kappa(\texttt{unit}))_\perp$$

then $e \Downarrow v$ iff $[\![e^*]\!] \neq \perp$.

Theorem 5.4 (Noninterference) *Let κ be an element of $\mathcal{L}$, and $n \notin \kappa$. Suppose*

$$\begin{array}{l} \emptyset \vdash e[(\lambda x : s.\, e')_n / f] : \texttt{unit} + \texttt{unit}, \kappa \\ \emptyset \vdash e[(\lambda x : s.\, e'')_n / f] : \texttt{unit} + \texttt{unit}, \kappa \end{array}$$

are derivable in the call-tracking type system. Then $(e[(\lambda x : s.\, e')_n / f]) \Downarrow v$ iff $(e[(\lambda x : s.\, e'')_n / f]) \Downarrow v$.

This theorem formalizes the intuition "expression e does not calling function f" as the property "function f can be replace by an arbitrary function (of appropriate type) without changing the result of evaluating of e".

6 Discussion

We have shown how many dependency analyses can be cast in DCC. As Section 4 shows, we can compare and contrast various dependency analyses in a single framework. For example, the call-by-name functional SLam calculus, the slicing calculus, and the binding-time calculus share a common translation of types into DCC and a set of common correctness properties; small differences occur only in the translations of terms. Larger differences between these calculi and the Smith-Volpano calculus can also be described.

Another advantage of the translations is their utility in the design of dependency analyses. For instance, we have found a certain incompleteness in the functional SLam calculus; it would make semantic sense to add a rule

$$\frac{\Gamma, x : (t, \kappa) \vdash e : (t', \kappa')}{\Gamma, x : (t, \kappa') \vdash e : (t', \kappa')}\ \kappa \sqsubseteq \kappa'$$

(since it is easily modelled in DCC), but the original SLam calculus does not have the rule. DCC can also be used to point out apparent design inconsistencies in some of the existing calculi. We are currently redesigning the Imperative SLam Calculus [13] using a translation into DCC as a guide for the type system, and as a vehicle for proving noninterference.

The model underlying DCC simplifies proofs of noninterference. The model was also invaluable in developing DCC itself. For instance, the pattern

$$\texttt{seq}\, x = e\ \texttt{in}\ (\texttt{bind}\, y = e'\ \texttt{in}\ e'')$$

occurs frequently in the translations; the type of e'' must be both pointed and protected in order for the translation to work. Without the concepts of "pointed" and "protected", the obvious path might be to adopt an ever increasingly complex set of type conversions and equations. The model was also helpful in developing the weaker notion of noninterference, and extending the notion of "protected" types to lifted types by changing the semantics of lifting. It would have been difficult to make this change in the syntax of DCC alone (other than, perhaps, by directly imposing the equation $T_\ell(s_\perp) = (T_\ell(s))_\perp$).

Not all aspects of dependency can be translated into DCC. For example, the binding-time analyses of Davies and Pfenning [7, 6] cannot be directly translated into DCC because DCC cannot model the coercion from run-time objects to compile-time objects. A rather different semantics due to Moggi [21] has been developed for such binding-time analyses, using the concept of a fibration to model dependency. A similar comment applies to the trust operator that maps from untrusted to trusted in Ørbæk and Palsberg's work [27].

Other possible extensions of DCC include accounting for the spawning of concurrent threads [13] and modelling cryptographic operations in such a way that encrypting a high-security datum could produce a low-security ciphertext [1]. The relationship of DCC to semantic dependency in the context of optimizing compilers [4, 11] and to region systems for memory management [37] should also be explored.

Acknowledgements

Thanks to Eugenio Moggi for discussions and to the anonymous referees for their comments.

Anindya Banerjee is a member of the Church Project and is supported in part by NSF grant EIA-9806835.

6 November 1998

References

[1] M. Abadi. Secrecy by typing in security protocols. In *Theoretical Aspects of Computer Software: Third International Symposium*, volume 1281 of *Lect. Notes in Computer Sci.* Springer-Verlag, 1997.

[2] M. Abadi, B. Lampson, and J.-J. Lévy. Analysis and caching of dependencies. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 83–91. ACM, 1996.

[3] S. K. Biswas. *Dynamic Slicing in Higher-Order Programming Languages*. PhD thesis, University of Pennsylvania, 1997.

[4] R. Cartwright and M. Felleisen. The semantics of program dependence. In *Proceedings of the 1989 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–27. ACM, 1989.

[5] C. Consel. Binding time analysis for higher order untyped functional languages. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 264–272. ACM, 1990.

[6] R. Davies. A temporal-logic approach to binding-time analysis. In *Proceedings, Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, 1996.

[7] R. Davies and F. Pfenning. A modal analysis of staged computation. In *Conference Record of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 258–270. ACM, 1996.

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

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

[10] 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.

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Programming Languages and Systems*, 9(3):319–349, 1987.

[12] J. Hatcliff and O. Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7:507–541, 1997. Special issue containing selected papers presented at the 1995 Workshop on Logic, Domains, and Programming Languages, Darmstadt, Germany.

[13] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Conference Record of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 365–377. ACM, 1998.

[14] B. T. Howard. Inductive, coinductive, and pointed types. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 102–109. ACM, 1996.

[15] P. Hudak, S. L. Peyton Jones, P. L. Wadler, Arvind, B. Boutel, J. Fairbairn, J. Fasel, M. Guzman, K. Hammond, J. Hughes, T. Johnsson, R. Kieburtz, R. S. Nikhil, W. Partain, and J. Peterson. Report on the functional programming language Haskell, Version 1.2. *ACM SIGPLAN Notices*, May 1992.

[16] J. Lambek and P. Scott. *Introduction to higher order categorical logic*. Cambridge studies in advanced mathematics. Cambridge University Press, 1986.

[17] J. McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley Press, 1994.

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

[19] M. Mizuno and D. A. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing*, 4:727–754, 1992.

[20] E. Moggi. Notions of computation and monads. *Information and Control*, 93:55–92, 1991.

[21] E. Moggi. A categorical account of two-level languages. In *Proceedings, Mathematical Foundations of Programming Semantics, Thirteenth Annual Conference*, Electronic Notes in Theoretical Computer Science. Elsevier, 1997. Available from `http://www.elsevier.nl/locate/entcs/`.

[22] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. ACM Press, 1997.

[23] A. C. Myers. Practical mostly-static information flow control. In *Conference Record of the Twenty-sixth Annual ACM Symposium on Principles of Programming Languages*. ACM, 1999.

[24] F. Nielson. Strictness analysis and denotational abstract interpretation. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 120–131. ACM, 1987.

[25] H. R. Nielson and F. Nielson. Automatic binding time analysis for a typed λ calculus. *Science of Computer Programming*, 10:139–176, 1988.

[26] H. R. Nielson and F. Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[27] P. Ørbæk and J. Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6):557–591, November 1997.

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

[29] J. G. Riecke and R. Viswanathan. Isolating side effects in sequential languages. In *Conference Record of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 1–12. ACM, 1995.

[30] A. Sabelfeld and David Sands. A Per model of secure information flow in sequential programs. Unpublished manuscript, 1998.

[31] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*. ACM, 1998.

[32] C. Strachey. The varieties of programming language. In *Proceedings of the International Computing Symposium*, pages 222–233. Cini Foundation, Venice, 1972. Reprinted in Peter O'Hearn and Robert Tennent, eds., *Algol-like Languages*. Birkhäuser, 1997.

[33] Y.-M. Tang. *Systèmes d'effet et interprétation abstraite pour l'analyse de flot de contrôle*. PhD thesis, Ecole Nationale Supériere des Mines de Paris, 1994.

[34] Y.-M. Tang and P. Jouvelot. Effect systems with subtyping. In *ACM Conference on Partial Evaluation and Program Manipulation*, June 1995.

[35] P. Thiemann. A unified framework for binding-time analysis. In M. Bidoit, editor, *Colloquium on Formal Approaches in Software Engineering (FASE '97)*, volume 1214 of *Lect. Notes in Computer Sci.*, pages 742–756. Springer-Verlag, April 1997.

[36] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[37] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

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

[39] P. Wadler. The marriage of effects and monads. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 63–74. ACM, 1998.

[40] M. Weiser. Program slicing. *IEEE Trans. Software Engineering*, 10(4):352–357, July 1984.

A Translations into DCC

The translations of the various source calculi into DCC are given in Tables 7-12 below. To make the translations more readable, most of the cases of sums and products are left out. We also use the DCC combinator and abbreviation

$$\begin{array}{l}\mathtt{dot} : T_\kappa(T_{\kappa'}(t)) \to T_{\kappa\sqcup\kappa'}(t)\\ \mathtt{dot} = \lambda x : T_\kappa(T_{\kappa'}(t)).\,\mathtt{bind}\, y = x\,\mathtt{in}\,\mathtt{bind}\, z = y\,\mathtt{in}\,(\eta_{\kappa\sqcup\kappa'}\, z)\end{array}$$

$$(\mathtt{seqbind}\, f = e\,\mathtt{in}\, e') = (\mathtt{seq}\, v = e\,\mathtt{in}\,\mathtt{bind}\, f = v\,\mathtt{in}\, e')$$

where v is a fresh variable. Most of the translations also require a coercion combinator for interpreting subsumption, but these combinators—and a few others—need to be defined specially for each system. These definitions are found in each translation.

Table 7: Translation of the Call-by-name Functional SLam Calculus into DCC (excerpts).

$$
\begin{array}{rcl}
\mathit{coerce}_{s_1,s_2} & : & s_1^* \to s_2^* \\
\mathit{coerce}_{(\mathtt{unit},\kappa),(\mathtt{unit},\kappa')} & = & \lambda x : T_\kappa(\mathtt{unit}_\perp).\,\mathtt{bind}\, y = x\, \mathtt{in}\, (\eta_{\kappa'}\, y) \\
\mathit{coerce}_{(s_2 \to u_1,\kappa),(s_1 \to u_2,\kappa')} & = & \lambda x : T_\kappa(s_2^* \to u_1^*).\,\mathtt{bind}\, y = x\, \mathtt{in}\, \eta_{\kappa'}\, (\lambda z : s_1^*.\, \mathit{coerce}_{u_1,u_2}(y\, (\mathit{coerce}_{s_1,s_2}\, z))) \\
\mathit{coerce}_{(s_1 \times u_1,\kappa),(s_2 \times u_2,\kappa')} & = & \lambda x : T_\kappa(s_1^* \times u_1^*).\,\mathtt{bind}\, y = x\, \mathtt{in}\, \eta_{\kappa'}\, \langle \mathit{coerce}_{s_1,s_2}(\mathtt{proj}_1\, y), \mathit{coerce}_{u_1,u_2}(\mathtt{proj}_2\, y)\rangle \\
\mathit{coerce}_{(s_1 + u_1,\kappa),(s_2 + u_2,\kappa')} & = & \lambda x : T_\kappa((s_1^* + u_1^*)_\perp).\,\mathtt{bind}\, y = x\, \mathtt{in}\, \eta_{\kappa'}\, (\mathtt{seq}\, z = y\, \mathtt{in}\ \mathtt{case}\, z \\
 & & \qquad \mathtt{of}\ \mathtt{inj}_1(w).\ \mathtt{lift}\ (\mathtt{inj}_1\, (\mathit{coerce}_{s_1,s_2}\, w)) \\
 & & \qquad \mid \mathtt{inj}_2(w).\ \mathtt{lift}\ (\mathtt{inj}_2\, (\mathit{coerce}_{u_1,u_2}\, w)))
\end{array}
$$

[*Var*]
$$\Gamma, x : s, \Gamma' \vdash x : s \Rightarrow \Gamma^*, x : s^*, (\Gamma')^* \vdash x : s^*$$

[*Unit*]
$$\Gamma \vdash ()_\kappa : (\mathtt{unit}, \kappa) \Rightarrow \Gamma^* \vdash \eta_\kappa(\mathtt{lift}\ ()) : T_\kappa(\mathtt{unit}_\perp)$$

[*Sub*]
$$\frac{\Gamma \vdash e : s_1 \qquad s_1 \leq s_2}{\Gamma \vdash e : s_2} \Rightarrow \frac{\Gamma^* \vdash e^* : s_1^*}{\Gamma^* \vdash (\mathit{coerce}_{s_1,s_2}\, e^*) : s_2^*}$$

[*Rec*]
$$\frac{\Gamma, f : s \vdash e : s}{\Gamma \vdash (\mu f : s.\, e) : s} \Rightarrow \frac{\Gamma^*, f : s^* \vdash e^* : s^*}{\Gamma^* \vdash (\mu f : s^*.\, e^*) : s^*}$$

[*Lam*]
$$\frac{\Gamma, x : s_1 \vdash e : s_2}{\Gamma \vdash (\lambda x : s_1.\, e)_\kappa : (s_1 \to s_2, \kappa)} \Rightarrow \frac{\Gamma^*, x : s_1^* \vdash e^* : s_2^*}{\Gamma^* \vdash (\eta_\kappa\, (\lambda x : s_1^*.\, e^*)) : T_\kappa(s_1^* \to s_2^*)}$$

[*App*]
$$\frac{\Gamma \vdash e : (s_1 \to s_2, \kappa) \qquad \Gamma \vdash e_1 : s_1}{\Gamma \vdash (e\, e_1) : s_2 \bullet \kappa} \Rightarrow \frac{\Gamma^* \vdash e^* : T_\kappa(s_1^* \to s_2^*) \qquad \Gamma^* \vdash e_1^* : s_1^*}{\Gamma^* \vdash \mathtt{dot}\ (\mathtt{bind}\, f = e^*\ \mathtt{in}\, (\eta_\kappa\, (f\, e_1^*))) : (s_2 \bullet \kappa)^*} \quad f \text{ is fresh}$$

[*Pair*]
$$\frac{\Gamma \vdash e_i : s_i}{\Gamma \vdash \langle e_1, e_2\rangle_\kappa : (s_1 \times s_2, \kappa)} \Rightarrow \frac{\Gamma^* \vdash e_i^* : s_i^*}{\Gamma^* \vdash (\eta_\kappa\, \langle e_1^*, e_2^*\rangle) : T_\kappa(s_1^* \times s_2^*)}$$

[*Proj*]
$$\frac{\Gamma \vdash e : (s_1 \times s_2, \kappa)}{\Gamma \vdash (\mathtt{proj}_i\, e) : s_i \bullet \kappa} \Rightarrow \frac{\Gamma^* \vdash e^* : T_\kappa(s_1^* \times s_2^*)}{\Gamma^* \vdash \mathtt{dot}\ (\mathtt{bind}\, x = e^*\ \mathtt{in}\, (\eta_\kappa\, (\mathtt{proj}_i\, x))) : (s_i \bullet \kappa)^*} \quad x \text{ is fresh}$$

[*Inj*]
$$\frac{\Gamma \vdash e : s_i}{\Gamma \vdash (\mathtt{inj}_i\, e)_\kappa : (s_1 + s_2, \kappa)} \Rightarrow \frac{\Gamma^* \vdash e^* : s_i^*}{\Gamma^* \vdash (\eta_\kappa\, (\mathtt{lift}\, (\mathtt{inj}_i\, e^*))) : T_\kappa((s_1^* + s_2^*)_\perp)}$$

[*Case*]
$$\frac{\Gamma \vdash e : (s_1 + s_2, \kappa) \qquad \Gamma, x : s_i \vdash e_i : s}{\Gamma \vdash (\mathtt{case}\, e\, \mathtt{of}\, \mathtt{inj}_1(x).\, e_1 \mid \mathtt{inj}_2(x).\, e_2) : s \bullet \kappa} \Rightarrow \frac{\Gamma^* \vdash e^* : T_\kappa((s_1^* + s_2^*)_\perp) \qquad \Gamma^*, x : s_i^* \vdash e_i^* : s^*}{\begin{array}{l}\Gamma^* \vdash \mathtt{dot}\ (\ \mathtt{bind}\, y = e^* \\ \qquad \mathtt{in}\ \ (\eta_\kappa\, (\ \mathtt{seq}\, v = y\, \mathtt{in} \\ \qquad\qquad \mathtt{case}\, v \\ \qquad\qquad\ \ \mathtt{of}\ \mathtt{inj}_1(x).\, e_1^* \\ \qquad\qquad\quad \mid \mathtt{inj}_2(x).\, e_2^*))) : (s \bullet \kappa)^*\end{array}} \quad y \text{ fresh}$$

[*Protect*]
$$\frac{\Gamma \vdash e : s}{\Gamma \vdash (\mathtt{protect}_\kappa\, e) : s \bullet \kappa} \Rightarrow \frac{\Gamma^* \vdash e^* : s^*}{\Gamma^* \vdash \mathtt{dot}\ (\eta_\kappa\, e^*) : (s \bullet \kappa)^*}$$

Table 8: Translation of the Slicing Calculus into DCC (excerpts).

[*Var*]
$$\Gamma, x : s, \Gamma' \vdash x : s \Rightarrow \Gamma^*, x : s^*, (\Gamma')^* \vdash x : s^*$$

[*Unit*]
$$\Gamma \vdash ()_n : (\mathtt{unit}, \{n\}) \Rightarrow \Gamma^* \vdash \eta_{\{n\}}(\mathtt{lift}\ ()) : T_{\{n\}}(\mathtt{unit}_\perp)$$

[*Lam*]
$$\frac{\Gamma, x : s_1 \vdash e : s_2}{\Gamma \vdash (\lambda x : s_1.\, e)_n : (s_1 \to s_2, \{n\})} \Rightarrow \frac{\Gamma^*, x : s_1^* \vdash e^* : s_2^*}{\Gamma^* \vdash (\eta_{\{n\}}\, (\lambda x : s_1^*.\, e^*)) : T_{\{n\}}(s_1^* \to s_2^*)}$$

[*App*]
$$\frac{\Gamma \vdash e : (s_1 \to s_2, \kappa) \qquad \Gamma \vdash e_1 : s_1}{\Gamma \vdash (e\, e_1) : s_2 \bullet \kappa} \Rightarrow \frac{\Gamma^* \vdash e^* : T_\kappa(s_1^* \to s_2^*) \qquad \Gamma^* \vdash e_1^* : s_1^*}{\Gamma^* \vdash \mathtt{dot}\ (\mathtt{bind}\, f = e^*\ \mathtt{in}\, (\eta_\kappa\, (f\, e_1^*))) : (s_2 \bullet \kappa)^*} \quad f \text{ is fresh}$$

Table 9: Translation of the Binding-time Calculus into DCC (excerpts).

[*Var*]
$$\Gamma, x : s, \Gamma \vdash x : s \Rightarrow \Gamma^*, x : s^*, (\Gamma')^* \vdash x : s^*$$

[*Unit*]
$$\Gamma \vdash ()_\beta : (\mathtt{unit}, \beta) \Rightarrow \Gamma^* \vdash (\eta_\beta\ (\mathtt{lift}\ ())) : T_\beta(\mathtt{unit}_\bot)$$

[*Lam*]
$$\frac{\Gamma, x : s_1 \vdash e : s_2}{\Gamma \vdash (\lambda x : s_1 . e)_\beta : (s_1 \to s_2, \beta)} \Rightarrow \frac{\Gamma^*, x : s_1^* \vdash e^* : s_2^*}{\Gamma^* \vdash (\eta_\beta\ (\lambda x : s_1^* . e^*)) : T_\beta(s_1^* \to s_2^*)}$$

[*App*]
$$\frac{\Gamma \vdash e : (s_1 \to s_2, \beta) \qquad \Gamma \vdash e_1 : s_1}{\Gamma \vdash (e\ e_1) : s_2} \Rightarrow \frac{\Gamma^* \vdash e^* : T_\beta(s_1^* \to s_2^*) \qquad \Gamma^* \vdash e_1^* : s_1^*}{\Gamma^* \vdash (\mathtt{bind}\, f = e^*\ \mathtt{in}\ (f\ e_1^*)) : s_2^*} \quad f \text{ is fresh}$$

Table 10: Translation of the Smith-Volpano Calculus into DCC.

$$\begin{array}{rcl}
\mathit{true} & = & (\mathtt{inj}_1\,()) \\
\mathit{false} & = & (\mathtt{inj}_2\,()) \\
(\mathtt{if}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2) & = & (\mathtt{case}\, e\, \mathtt{of}\, \mathtt{inj}_1(x).\, e_1 \mid \mathtt{inj}_2(x).\, e_2), \qquad x \text{ is fresh} \\
\mathtt{proj}_x & = & \text{the projection of the state to the variable } x \\
\mathit{coerce}_{\ell,\ell} & = & \lambda f.\, f, \qquad \ell = L, H, (L\ \mathtt{cmd}), \text{ or } (H\ \mathtt{cmd}) \\
\mathit{coerce}_{L,H} & = & \lambda f.\, \lambda s : SV_H(\Gamma_H) \times SV_L(\Gamma_L).\, \eta_H\ (f\ s) \\
\mathit{coerce}_{H\ \mathtt{cmd}, L\ \mathtt{cmd}} & = & \lambda f.\, \lambda s : SV_H(\Gamma_H) \times SV_L(\Gamma_L).\, \mathtt{lift}\ \langle f\ s, \mathtt{proj}_2\ s \rangle
\end{array}$$

[*Var*]
$$\Gamma_H; \Gamma_L \vdash x : \tau \Rightarrow (\lambda\sigma.\, \mathtt{proj}_x\ \sigma) : SV(\Gamma_H, \Gamma_L, \tau) \qquad \text{if } x \in \Gamma_\tau$$

[*TrueH*]
$$\Gamma_H; \Gamma_L \vdash \mathtt{true} : H \Rightarrow (\lambda\sigma.\, \eta_H\ \mathit{true}) : SV(\Gamma_H, \Gamma_L, H)$$

[*FalseH*]
$$\Gamma_H; \Gamma_L \vdash \mathtt{false} : H \Rightarrow (\lambda\sigma.\, \eta_H\ \mathit{false}) : SV(\Gamma_H, \Gamma_L, H)$$

[*TrueL*]
$$\Gamma_H; \Gamma_L \vdash \mathtt{true} : L \Rightarrow (\lambda\sigma.\, \mathit{true}) : SV(\Gamma_H, \Gamma_L, L)$$

[*FalseL*]
$$\Gamma_H; \Gamma_L \vdash \mathtt{false} : L \Rightarrow (\lambda\sigma.\, \mathit{false}) : SV(\Gamma_H, \Gamma_L, L)$$

[*SkipH*]
$$\Gamma_H; \Gamma_L \vdash \mathtt{skip} : H\ \mathtt{cmd} \Rightarrow (\lambda\sigma.\, \mathtt{proj}_1\ \sigma) : SV(\Gamma_H, \Gamma_L, H\ \mathtt{cmd})$$

[*SkipL*]
$$\Gamma_H; \Gamma_L \vdash \mathtt{skip} : L\ \mathtt{cmd} \Rightarrow (\lambda\sigma.\, \mathtt{lift}\ \sigma) : SV(\Gamma_H, \Gamma_L, L\ \mathtt{cmd})$$

[*Sub*]
$$\frac{\Gamma_H; \Gamma_L \vdash e : s_0 \qquad s_0 \le s_1}{\Gamma_H; \Gamma_L \vdash e : \tau'} \Rightarrow \frac{e^* : SV(\Gamma_H, \Gamma_L, s_0)}{(\mathit{coerce}_{s_0,s_1}\ e^*) : SV(\Gamma_H, \Gamma_L, s_1)}$$

[*AssignH*]
$$\frac{\Gamma_H; \Gamma_L \vdash e : H \qquad \Gamma_H = (x_1, \ldots, x_n)}{\Gamma_H; \Gamma_L \vdash (x_i := e) : H\ \mathtt{cmd}} \Rightarrow \frac{e^* : SV(\Gamma_H, \Gamma_L, H)}{(\lambda\sigma.\, \langle \mathtt{proj}_1\ (\mathtt{proj}_1\ \sigma), \ldots, (e^*\ \sigma), \ldots, \rangle) : SV(\Gamma_H, \Gamma_L, H\ \mathtt{cmd})}$$

[*AssignL*]
$$\frac{\Gamma_H; \Gamma_L \vdash e : L \qquad \Gamma_L = (x_1, \ldots, x_n)}{\Gamma_H; \Gamma_L \vdash (x_i := e) : L\ \mathtt{cmd}} \Rightarrow \frac{e^* : SV(\Gamma_H, \Gamma_L, L)}{(\lambda\sigma.\, \mathtt{lift}\ \langle (\mathtt{proj}_1\ \sigma), \langle \mathtt{proj}_1\ (\mathtt{proj}_2\ \sigma), \ldots, (e^*\ \sigma), \ldots, \rangle \rangle) : SV(\Gamma_H, \Gamma_L, L\ \mathtt{cmd})}$$

[*SeqH*]
$$\frac{\Gamma_H; \Gamma_L \vdash c_1 : H\ \mathtt{cmd} \qquad \Gamma_H; \Gamma_L \vdash c_2 : H\ \mathtt{cmd}}{\Gamma_H; \Gamma_L \vdash (c_1; c_2) : H\ \mathtt{cmd}} \Rightarrow \frac{c_1^* : SV(\Gamma_H, \Gamma_L, H\ \mathtt{cmd}) \qquad c_2^* : SV(\Gamma_H, \Gamma_L, H\ \mathtt{cmd})}{(\lambda\sigma.\, c_2^*\ \langle c_1^*\ \sigma, \mathtt{proj}_2\ \sigma \rangle) : SV(\Gamma_H, \Gamma_L, H\ \mathtt{cmd})}$$

[*SeqL*]
$$\frac{\Gamma_H; \Gamma_L \vdash c_1 : L\ \mathtt{cmd} \qquad \Gamma_H; \Gamma_L \vdash c_2 : L\ \mathtt{cmd}}{\Gamma_H; \Gamma_L \vdash (c_1; c_2) : L\ \mathtt{cmd}} \Rightarrow \frac{c_1^* : SV(\Gamma_H, \Gamma_L, L\ \mathtt{cmd}) \qquad c_2^* : SV(\Gamma_H, \Gamma_L, L\ \mathtt{cmd})}{(\lambda\sigma.\, \mathtt{seq}\, \sigma_1 = (c_1^*\ \sigma)\ \mathtt{in}\ (c_2^*\ \sigma_1)) : SV(\Gamma_H, \Gamma_L, L\ \mathtt{cmd})}$$

[*IfH*]
$$\frac{\Gamma_H; \Gamma_L \vdash e : H \qquad \Gamma_H; \Gamma_L \vdash c_i : H\ \mathtt{cmd}}{\Gamma_H; \Gamma_L \vdash \mathtt{if}\, e\, \mathtt{then}\, c_1\, \mathtt{else}\, c_2 : H\ \mathtt{cmd}} \Rightarrow \frac{e^* : SV(\Gamma_H, \Gamma_L, H) \qquad c_i^* : SV(\Gamma_H, \Gamma_L, H\ \mathtt{cmd})}{(\lambda\sigma.\, \mathtt{bind}\, v = (e^*\ \sigma)\ \mathtt{in}\ \mathtt{if}\, v\, \mathtt{then}\, (c_1^*\ \sigma)\ \mathtt{else}\ (c_2^*\ \sigma)) : SV(\Gamma_H, \Gamma_L, H\ \mathtt{cmd})} \quad v \text{ is fresh}$$

[*IfL*]
$$\frac{\Gamma_H; \Gamma_L \vdash e : L \qquad \Gamma_H; \Gamma_L \vdash c_i : H\ \mathtt{cmd}}{\Gamma_H; \Gamma_L \vdash \mathtt{if}\, e\, \mathtt{then}\, c_1\, \mathtt{else}\, c_2 : L\ \mathtt{cmd}} \Rightarrow \frac{e^* : SV(\Gamma_H, \Gamma_L, L) \qquad c_i^* : SV(\Gamma_H, \Gamma_L, L\ \mathtt{cmd})}{(\lambda\sigma.\, \mathtt{if}\ (e^*\ \sigma)\ \mathtt{then}\ (c_1^*\ \sigma)\ \mathtt{else}\ (c_2^*\ \sigma)) : SV(\Gamma_H, \Gamma_L, L\ \mathtt{cmd})}$$

[*While*]
$$\frac{\Gamma_H; \Gamma_L \vdash e : L \qquad \Gamma_H; \Gamma_L \vdash c : L\ \mathtt{cmd}}{\Gamma_H; \Gamma_L \vdash \mathtt{while}\, e\, \mathtt{do}\, c : L\ \mathtt{cmd}} \Rightarrow \frac{e^* : SV(\Gamma_H, \Gamma_L, L) \qquad c^* : SV(\Gamma_H, \Gamma_L, L\ \mathtt{cmd})}{(\mu f.\, \lambda\sigma.\, \mathtt{if}\ (e^*\ \sigma)\ \mathtt{then}\ \mathtt{seq}\, \sigma' = (c^*\ \sigma)\ \mathtt{in}\ (f\ \sigma')\ \mathtt{else}\ (\mathtt{lift}\ \sigma)) : SV(\Gamma_H, \Gamma_L, L\ \mathtt{cmd})}$$

Table 11: Translation of the Call-by-value Functional SLam Calculus into vDCC (excerpts).

$$
\begin{aligned}
\mathit{fix} &= \mu f.\, \lambda g : s^* \to (s^*)_\perp .\, g\, (\eta_\kappa\, (\lambda x : s_1^* .\, \mathtt{seqbind}\, h = (f\, g)\, \mathtt{in}\, (h\, x))) \qquad \text{if } s = (s_1 \to s_2, \kappa) \text{ and } (s_2 \bullet \kappa) = s_2 \\
\mathit{coerce}_{(\mathtt{unit},\kappa),(\mathtt{unit},\kappa')} &= \lambda x : T_\kappa(\mathtt{unit}).\, \mathtt{bind}\, y = x\, \mathtt{in}\, (\eta_{\kappa'}\, y) \\
\mathit{coerce}_{(s_2 \to u_1,\kappa),(s_1 \to u_2,\kappa')} &= \lambda x : T_\kappa(s_2^* \to (u_1^*)_\perp).\, \mathtt{bind}\, y = x\, \mathtt{in}\, \eta_{\kappa'}\, (\lambda z : s_1^* .\, \mathtt{seq}\, v = (y\, (\mathit{coerce}_{s_1,s_2}\, z))\, \mathtt{in}\, \mathtt{lift}\, (\mathit{coerce}_{u_1,u_2}\, v))
\end{aligned}
$$

[*Var*] $\Gamma, x : s, \Gamma' \vdash x : s \Rightarrow \Gamma^*, x : s^*, (\Gamma')^* \vdash (\mathtt{lift}\, x) : (s^*)_\perp$

[*Unit*] $\Gamma \vdash ()_\kappa : (\mathtt{unit}, \kappa) \Rightarrow \Gamma^* \vdash (\mathtt{lift}\, (\eta_\kappa\, ())) : (T_\kappa(\mathtt{unit}))_\perp$

[*Sub*] $$\frac{\Gamma \vdash e : s_1 \qquad s_1 \leq s_2}{\Gamma \vdash e : s_2} \Rightarrow \frac{\Gamma^* \vdash e^* : (s_1^*)_\perp}{\Gamma^* \vdash \mathtt{seq}\, w = e^*\, \mathtt{in}\, \mathtt{lift}\, (\mathit{coerce}_{s_1,s_2}\, w) : (s_2^*)_\perp} \ w \text{ fresh}$$

[*Rec*] $$\frac{\Gamma, f : s \vdash e : s \qquad s = (s_1 \to s_2, \kappa) \qquad (s_2 \bullet \kappa) = s_2}{\Gamma \vdash (\mu f : s.\, e) : s} \Rightarrow \frac{\Gamma^*, f : s^* \vdash e^* : (s^*)_\perp}{\Gamma^* \vdash \mathit{fix}\, (\lambda f : s^* .\, e^*) : (s^*)_\perp}$$

[*Lam*] $$\frac{\Gamma, x : s_1 \vdash e : s_2}{\Gamma \vdash (\lambda x : s_1 .\, e)_\kappa : (s_1 \to s_2, \kappa)} \Rightarrow \frac{\Gamma^*, x : s_1^* \vdash e^* : (s_2^*)_\perp}{\Gamma^* \vdash \mathtt{lift}\, (\eta_\kappa\, (\lambda x : s_1^* .\, e^*)) : (T_\kappa(s_1^* \to (s_2^*)_\perp))_\perp}$$

[*App*] $$\frac{\Gamma \vdash e : (s_1 \to s_2, \kappa) \qquad \Gamma \vdash e_1 : s_1}{\Gamma \vdash (e\, e_1) : s_2 \bullet \kappa} \Rightarrow \frac{\Gamma^* \vdash e^* : (T_\kappa(s_1^* \to (s_2^*)_\perp))_\perp \qquad \Gamma^* \vdash e_1^* : (s_1^*)_\perp}{\begin{array}{ll}\Gamma^* \vdash & \mathtt{seqbind}\, f = e^*\, \mathtt{in} \\ & \mathtt{seq}\, v = e_1^*\, \mathtt{in}\, \mathtt{seq}\, r = (f\, v)\, \mathtt{in}\, \mathtt{lift}\, (\mathtt{dot}\, (\eta_\kappa\, r)) : (s_2 \bullet \kappa)_\perp^*\end{array}} \ f, v, r \text{ fresh}$$

[*Protect*] $$\frac{\Gamma \vdash e : s}{\Gamma \vdash (\mathtt{protect}_\kappa\, e) : s \bullet \kappa} \Rightarrow \frac{\Gamma^* \vdash e^* : (s^*)_\perp}{\Gamma^* \vdash \mathtt{seq}\, m = e^*\, \mathtt{in}\, \mathtt{lift}\, (\mathtt{dot}\, (\eta_\kappa\, m)) : (s \bullet \kappa)_\perp^*} \ m \text{ fresh}$$

Table 12: Translation of the Call-tracking Calculus into vDCC (excerpts).

$$\mathit{fix} = \mu f.\, \lambda g : s^* \to (T_\kappa(s^*))_\perp .\, g\, (\lambda x : s_1^* .\, \mathtt{seqbind}\, h = (f\, g)\, \mathtt{in}\, (h\, x)) \qquad \text{if } s = (s_1 \xrightarrow{\kappa} s_2)$$

[*Var*] $\Gamma, x : s, \Gamma' \vdash x : s, L \Rightarrow \Gamma^*, x : s^*, (\Gamma')^* \vdash \mathtt{lift}\, (\eta_L\, x) : (T_L(s^*))_\perp$

[*Unit*] $\Gamma \vdash () : \mathtt{unit}, L \Rightarrow \Gamma^* \vdash \mathtt{lift}\, (\eta_L\, ()) : (T_L(\mathtt{unit}))_\perp$

[*Rec*] $$\frac{\Gamma, f : s \vdash e : s, \kappa \qquad s = (s_1 \xrightarrow{\kappa} s_2)}{\Gamma \vdash (\mu f : s.\, e) : s} \Rightarrow \frac{\Gamma^*, f : s^* \vdash e^* : (T_\kappa(s^*))_\perp}{\Gamma^* \vdash \mathit{fix}\, (\lambda f : s^* .\, e^*) : (T_\kappa(s^*))_\perp}$$

[*Lam*] $$\frac{\Gamma, x : s_1 \vdash e : s_2, \kappa}{\Gamma \vdash (\lambda x : s_1 .\, e)_n : (s_1 \xrightarrow{\kappa \sqcup \{n\}} s_2), L} \Rightarrow \frac{\Gamma^*, x : s_1^* \vdash e^* : (T_\kappa(s_2^*))_\perp}{\Gamma^* \vdash (\mathtt{lift}\, (\eta_L\, (\lambda x : s_1^* .\, \mathtt{seq}\, r = e^*\, \mathtt{in}\, \mathtt{lift}\, (\mathtt{dot}\, (\eta_{\{n\}}\, r))))) : (T_L(s_1^* \to (T_{\{n\} \sqcup \kappa}(s_2^*))_\perp))_\perp} \ r \text{ fresh}$$

[*App*] $$\frac{\Gamma \vdash e : (s_1 \xrightarrow{\kappa} s_2), \kappa_1 \qquad \Gamma \vdash e_1 : s_1, \kappa_2}{\Gamma \vdash (e\, e_1) : s_2, \kappa \sqcup \kappa_1 \sqcup \kappa_2} \Rightarrow \frac{\Gamma^* \vdash e^* : (T_{\kappa_1}(s_1^* \to (T_\kappa(s_2^*))_\perp))_\perp \qquad \Gamma^* \vdash e_1^* : (T_{\kappa_2}(s_1^*))_\perp}{\begin{array}{ll}\Gamma^* \vdash & \mathtt{seqbind}\, f = e^*\, \mathtt{in} \\ & \mathtt{seqbind}\, y = e_1^*\, \mathtt{in} \\ & \mathtt{seqbind}\, v = (f\, y)\, \mathtt{in}\, \mathtt{lift}\, (\eta_{\kappa \sqcup \kappa_1 \sqcup \kappa_2}\, v) : (T_{\kappa \sqcup \kappa_1 \sqcup \kappa_2}(s_2^*))_\perp\end{array}} \ f, y, v \text{ fresh}$$