

# Liberalizing Dependency

Avik Chaudhuri

University of Maryland at College Park  
avik@cs.umd.edu

**Abstract.** The dependency core calculus (DCC) is a simple extension of the computational lambda calculus, that captures a common notion of dependency that arises in many programming language settings. This notion of dependency is closely related to the notion of information flow in security; it is sensitive not only to data dependencies that cause explicit flows, but also to control dependencies that cause implicit flows. In this paper, we study variants of DCC in which the data and control dependencies are decoupled. This allows us to consider settings where a weaker notion of dependency—one that restricts only explicit flows—may usefully coexist with DCC’s stronger notion of dependency. In particular, we show how strong, noninterference-based security may be reconciled with weak, trace-based security within the same system, improving soundness in one direction and completeness in the other.

## 1 Introduction

The dependency core calculus (DCC) [2] is a simple extension of the computational lambda calculus [12], where each level  $\ell$  in a lattice is associated with a type constructor  $T_\ell$  that behaves as a monad. DCC was designed to capture a central notion of dependency common to many programming language settings, including security. This notion of dependency is closely related to the concepts of parametricity [14, 17] and noninterference [10, 2]. Roughly, DCC’s type system guarantees that the computational effects of a program protected by some level  $\ell$  can only be observed by programs protected by levels  $\ell$  or higher in the lattice. Unsurprisingly, this notion of dependency tracks not only explicit effects due to data flow, but also implicit effects due to control flow. For example, consider the following functions:

$$\begin{aligned} f &= \lambda x : T_\ell(s_1 + s_2). \text{bind } y = x \text{ in } y \\ g &= \lambda x : T_\ell(s_1 + s_2). \text{bind } y = x \text{ in case } y \text{ of } \text{inj}_1(z). (\text{inj}_1 ()) \parallel \text{inj}_2(z). (\text{inj}_2 ()) \end{aligned}$$

The type of  $x$  is an  $\ell$ -protected union type  $(s_1 + s_2)$ . A value of this type has the form  $(\eta_\ell (\text{inj}_i e_i))$ ,  $i \in \{1, 2\}$ , where  $\eta_\ell$  is the unit of  $T_\ell$  and represents some  $\ell$ -protection mechanism,  $\text{inj}_i$  is a case constructor, and  $e_i$  is some expression of type  $s_i$ . The function  $f$  removes the protection on  $x$  and returns it. The function  $g$  removes the protection on  $x$ , applies its case constructor to unit, and returns it. Of course, neither function is typable in DCC, since  $f$  and  $g$  return unprotected

results that depend on  $x$ . In other words,  $f$  and  $g$  leak information on  $x$ . Still, intuitively  $g$  may seem safer than  $f$ —while  $f$  explicitly reveals all information on  $x$  through data flow,  $g$  implicitly reveals “only” one bit of information on  $x$  through control flow.<sup>1</sup>

Traditionally, security experts have dismissed this intuition as unsound, since the attacker might be able to amplify the one-bit leak of information in  $g$  to leak all information on  $x$ , effectively making it as dangerous as  $f$ . However, such attacks are complex and seem rare in practice [16, 11]. Thus, several recent static analyses for security have focused on restricting effects due to data flow, while ignoring other effects [7, 16, 3, 6]. From a theoretical perspective, one may simply consider these analyses unsound, and assume that they provide no guarantee. Alternatively, one may try to understand the precise guarantee that these analyses provide, and evaluate whether such a guarantee is at all important for security. We take the latter stance in this paper.

Previous work on downgrading and robustness [18, 13] deals with similar concerns. Roughly, downgrading allows some specific information in the system to be released, and robustness guarantees that this does not cause further, unintentional leak of information in the system. For example, a function that checks whether a given password is correct releases information on the correct password whenever it returns the result of the check. A system using this function may still be robust, in the sense that the attacker cannot exploit the information released by the function to leak further information in the system.

However, downgrading as a mechanism of information release may be too coarse. For example, it blurs the qualitative distinction between a usual password-checking function that releases partial information on the correct password, and a function that releases the correct password itself. (This distinction is similar to the one between functions  $g$  and  $f$  above.) In this paper, we explore a finer mechanism of information release, called *weakening*. In particular, weakening the protection on the correct password allows information on it to be released implicitly through control flow, but not explicitly through data flow. Robustness requires that this weakening does not trigger further weakening in the system.

We study weakening and its properties by considering variants of DCC in which explicit and implicit effects are decoupled. The implicit effects arise entirely out of case analysis, so the main differences with DCC lie in the handling of union types. For instance, consider the following typing rule in DCC:

$$\frac{\Gamma \vdash e : T_\ell(s) \quad \Gamma, x : s \vdash e' : t}{\Gamma \vdash (\text{bind } x = e \text{ in } e') : t} \quad t \text{ is protected at } \ell$$

The variable  $x$  binds the result of  $e$  after removing its protection. Since  $x$  is in the scope of  $e'$ , the computational effects of  $e'$  should only be observable to programs that are protected by levels  $\ell$  or higher. This is ensured by the side condition in the rule above, which restricts  $t$  to be only of certain forms. (We will review

---

<sup>1</sup> Of course, this advantage vanishes if  $x$  contains only one bit of information, such as if  $x$  is a boolean; on the other hand, if  $x$  is a list of integers, the advantage is significant—while  $f$  reveals the list itself,  $g$  only reveals whether the list is empty.

the formal definition of this condition later.) In particular,  $t$  cannot be a union type, because information on  $x$  may be leaked through the case constructor of a value of such type. Indeed, this is exactly why  $f$  and  $g$  are not typable in DCC; their results have, resp., types  $(s_1 + s_2)$  and  $(\text{unit} + \text{unit})$ .

In contrast, we consider the following rule in a variant of DCC, called  $\text{DCC}^D$ :

$$\frac{\Gamma \vdash e : \overline{T}_\ell(s) \quad \Gamma, x : s^\ell \vdash e' : t}{\Gamma \vdash (\text{bind } x = e \text{ in } e') : t} \quad t \text{ is weakly protected at } \ell$$

The type constructor  $\overline{T}_\ell$  provides weaker protection than  $T_\ell$ ; as discussed earlier, it focuses on restricting effects due to data flow, while ignoring other effects. In particular, the side condition in the rule above allows  $t$  to be a union type. At the same time,  $t$  is adequately restricted to ensure that  $x$  itself is not released without protection. We introduce *open types* for this purpose; roughly, an open type  $s^\ell$  is given to a value of type  $s$  that requires protection by level  $\ell$ . We assume such a type for  $x$ , and prevent  $t$  from being an open type. In the resulting system,  $g$  is typable (after weakening the type of  $x$ ) but  $f$  is not. We show that if a program cannot be typed in  $\text{DCC}^D$ , then it cannot be typed in DCC. Furthermore, we formalize the precise guarantee enforced by  $\text{DCC}^D$ . This guarantee eliminates (at least) Denning and Denning's so-called *explicit flow* attacks [8]. Since such attacks are control-insensitive by definition,  $\text{DCC}^D$ 's guarantee seems to be important for security, at least from a practical perspective.

While the typing rules of  $\text{DCC}^D$  have an interesting flavor of their own, mixing them with DCC's typing rules can yield surprisingly pleasant cocktails. We explore a couple of such recipes in this paper; they highlight the symbiotic nature of these systems.

- We study a *weaken* primitive that allows values of type  $T_\ell$  to be cast as values of type  $\overline{T}_\ell$ . This weakening may invalidate strong protection guarantees at levels  $\ell$  and lower. However, weak protection guarantees will still hold at these levels, and strong protection guarantees should hold at all other levels. We enforce these guarantees by recycling DCC's types to carry *blames* for weakening. Specifically, let  $\theta$  be a fixed isomorphism from the lattice of levels to some lattice of blames; we include the following rule:

$$\frac{\Gamma \vdash e : T_\ell(s)}{\Gamma \vdash \text{weaken } e : T_{\theta(\ell)}(\overline{T}_\ell(s))}$$

The behavior of the resulting system,  $\text{DCC}^{DC}$ , rests on the definition of  $\theta$ .

- If  $\theta$  preserves joins and meets over the relevant lattices, then the type of a program must carry a blame  $\theta(\ell)$  such that  $\ell$  *upper-bounds* the levels of weakening on which its results may depend. In other words, strong protection guarantees hold at all levels not  $\ell$  or lower.
- If  $\theta$  exchanges joins and meets over the relevant lattices, then the type of a program must carry a blame  $\theta(\ell)$  such that  $\ell$  *lower-bounds* the levels of weakening on which its results may depend. In other words,  $\ell$  provides a measure of robustness for these guarantees.

- We improve the precision of DCC’s type system by relying on  $DCC^D$ ’s type system as an oracle. Consider the following functions, rejected by DCC because union types are not considered protected, as discussed earlier:

$$\lambda x. \text{bind } y = x \text{ in } (\text{inj}_i ()) \quad i \in \{1, 2\}$$

Clearly this restriction on union types is too harsh—after all, the functions above are constant! We relax this restriction by observing that any information leak is ultimately due to either an explicit leak through data flow or an implicit leak through control flow. Specifically, evaluating an expression of union type may reveal information about sensitive data only if that expression either does a case analysis on sensitive data, or releases the sensitive data itself. We prevent the former possibility by including a side condition in the rule for `case`, and the latter by delegating to  $DCC^D$ ’s typing rules. We show that the resulting system,  $DCC^{CD}$ , is sound and accepts strictly more programs than DCC.

In the context of security, these results suggest some interesting ways in which strong, noninterference-based security may be reconciled with weak, trace-based security within the same system, improving soundness in one direction and completeness in the other. Specifically, in a system where protection may have been partially weakened, a strong blame analysis can be used to provide strong protection guarantees for those parts of the system that are not affected by such weakening. Conversely, a weak flow analysis can be used to increase the coverage of such guarantees.

To summarize, we make the following contributions in this paper.

- We deconstruct DCC, which captures standard information flow, into a weaker system  $DCC^D$  that is instead inspired by Denning and Denning’s characterization of explicit information flow. We argue that this system provides the missing foundations for several recent static analyses for security that do not restrict implicit information flow (Section 3).
- We study a language primitive `weaken` that translates between DCC-style protection and  $DCC^D$ -style protection of programs. Such weakening may be viewed as a milder form of downgrading that preserves data-flow guarantees for the resulting programs. Furthermore, we show how such weakening can be controlled by reusing DCC mechanisms to associate blames for weakening (Section 4).
- Going in the other direction, we study how  $DCC^D$ ’s typing rules can improve the precision of DCC’s typing rules. This technique (once again) relies on deconstructing information flow into explicit and implicit information flow (Section 5).

Overall, we believe that the main importance of these results lies in their conceptual rather than technical details. Still, the technical details are sometimes challenging. For instance, we need to develop a theory of open types to correctly track explicit flows in  $DCC^D$  (Section 3). Mixing the typing rules of DCC and

DCC<sup>D</sup> to obtain improved hybrid systems (Sections 4 and 5) also requires much care for correctness.

We review DCC next (Section 2), deferring further discussion on limitations, related work, and conclusions until the end (Section 6).

## 2 Background on DCC

Recall that the computational lambda calculus [12] extends the simply typed lambda calculus with a type constructor that is interpreted as a monad. The monad is used to systematically include and control effects in the language; the same idea appears in Haskell to guarantee that “pure” functions cannot depend on “impure” computations. DCC [2] carries this idea further by distinguishing the effects of different “levels” of computation, and allowing computations at some levels to depend on those at others. Specifically, DCC includes a monadic type constructor for each level in a lattice, and has a special typing rule that allows computations at different levels to be composed based on the lattice. We review this system below.

Let  $\ell$  denote levels in a lattice with ordering  $\sqsubseteq$ , join  $\sqcup$ , meet  $\sqcap$ , bottom  $\perp$ , and top  $\top$ . We focus on the following syntax for types and terms in DCC. (For simplicity, we omit any discussion of pointed types and nonterminating programs in this paper; see Section 6 for further comments on this issue.)

### Syntax

$\begin{aligned} \text{types } s, t ::= & \text{unit} \mid (s \times t) \mid (s + t) \mid (s \rightarrow t) \mid T_\ell(s) \\ \text{terms } e, v ::= & () \mid \langle e, e' \rangle \mid (\text{proj}_i e) \mid (\text{inj}_i e) \mid \text{case } e \text{ of } \text{inj}_1(x). e_1 \parallel \text{inj}_2(x). e_2 \\ & \mid \lambda x. e \mid (e \ e') \mid (\eta_\ell e) \mid \text{bind } x = e \text{ in } e' \end{aligned}$
---

Types include unit, intersection, union, and function types, as well as types  $T_\ell(s)$  for each  $\ell$  in the lattice. Programs include the introduction and elimination forms for these types. In particular,  $(\eta_\ell e)$  has type  $T_\ell(s)$  whenever  $e$  has type  $s$ , and the reduction rule for bind is the following.

$$\text{bind } x = (\eta_\ell e) \text{ in } e' \longrightarrow e'[e/x]$$

In practice,  $\eta_\ell$  may represent any mechanism that provides “protection” at level  $\ell$ , broadly construed. In the context of secrecy, for instance,  $(\eta_\ell e)$  may be viewed as an encryption of  $e$  with a key secret to level  $\ell$ .<sup>2</sup> The typing rule for bind should then ensure that the secrecy of  $e$  is preserved in the above reduction. In particular, this may require that the result be similarly encrypted.

This intuition is captured by a predicate  $\ell \preceq t$ , read as “ $t$  is protected at  $\ell$ ”. Roughly, this means that terms of type  $t$  cannot leak any information at level  $\ell$ —in other words, terms of type  $t$  are indistinguishable to any level  $\ell'$  that is not at least  $\ell$  in the lattice. The following rules define this predicate.

<sup>2</sup> Conversely, in the context of integrity,  $(\eta_\ell e)$  may be viewed as a copy of  $e$  that is tainted by  $\ell$ .

## Protection rules

$$\begin{array}{l}
 \text{(P-unit)} \quad \ell \preceq \text{unit} \\
 \text{(P-intersection)} \quad \ell \preceq s \wedge \ell \preceq t \Rightarrow \ell \preceq (s \times t) \\
 \text{(P-function)} \quad \ell \preceq t \Rightarrow \ell \preceq (s \rightarrow t) \\
 \text{(P-monad-1,2)} \quad \ell \sqsubseteq \ell' \Rightarrow \ell \preceq T_{\ell'}(s) \quad , \quad \ell \preceq s \Rightarrow \ell \preceq T_{\ell'}(s)
 \end{array}$$

These rules may be explained as follows. The term  $()$  of type `unit` cannot leak any information since it is the only term of that type. Terms of type  $(s \times t)$ —which evaluate to tuples—cannot leak any more information than their projections, which have types  $s$  and  $t$ . Similarly, terms of type  $(s \rightarrow t)$ —which evaluate to functions—cannot leak any more information than their bodies, which have type  $t$ . Finally, terms of type  $T_{\ell'}(s)$ —which evaluate to terms protected at level  $\ell'$ —cannot leak information at level  $\ell$  if  $\ell'$  is at least  $\ell$ ; and in any case, they cannot leak any more information than their unprotected payloads.

Significantly, this definition does not consider union types to be protected. The broad reason is that case constructors are the (only) carriers of information in terms, which makes any term of union type a potential “suspect”. (Indeed, the other constructors—unit, tupling, function abstraction, and  $\ell$ -protection—cannot convey any information since they are the only introduction forms for the associated types.) For instance, a boolean may be encoded as either  $(\text{inj}_1 ())$  or  $(\text{inj}_2 ())$ , thereby carrying one bit of information; so the union type  $(\text{unit} + \text{unit})$  can serve as an encoding of the datatype `boolean`. Similarly,  $(\text{unit} + (\text{unit} + \text{unit}))$  can serve as an encoding of `boolean option`, with  $(\text{inj}_1 ())$  encoding the value `none` and  $(\text{inj}_2 b)$  encoding the value `some(b)` for a boolean  $b$ . In general, complex datatypes can be encoded using union types, and the only way of distinguishing values of those types is by analyzing the case constructors used in the terms encoding those values. Thus, it makes sense to require explicit protection on any term of a union type. (However, we will show in Section 5 that this restriction is overly conservative.)

Figure 1 shows the typing rules for DCC. Judgments are of the form  $\Gamma; \Pi \vdash e : t$ , where  $\Gamma$  contains type hypotheses for free variables and  $\Pi$  is a *protection context*<sup>3</sup> [17], which indicates the maximum level of protection promised by the context. If  $e$  is closed,  $\Gamma$  is empty and  $\Pi$  is  $\perp$ , and we use the simpler notation  $\vdash e : t$  for the typing judgment.

Other than **(T-ret)** and **(T-bind)**, these are standard rules for a simply typed lambda calculus with union and intersection types. In addition, **(T-ret)** states that  $(\eta_{\ell} e)$  has type  $T_{\ell}(s)$  whenever  $e$  has type  $s$ ; the latter derivation may assume that  $e$  will be protected at  $\ell$  by its context, which is made evident by joining  $\ell$  with the protection context. **(T-bind)** states that `bind  $x = e$  in  $e'$`  has type  $t$  only if  $e$  has a type of the form  $T_{\ell}(s)$  and  $e'$  has type  $t$  assuming that  $x$  has type  $s$ ; furthermore, the type  $T_{\Pi}(t)$  must be protected at  $\ell$ . This means that either  $t$  must be protected at  $\ell$ , or  $\Pi$  must be at least  $\ell$ . In any case, this ensures

<sup>3</sup> Protection contexts did not appear in the original definition of DCC [2], but their inclusion has some pleasant consequences; see [17].

**Figure 1: Typing rules (DCC)**

(T-var)	$\Gamma, x : s, \Gamma'; \Pi \vdash x : s$
(T-unit)	$\Gamma; \Pi \vdash () : \text{unit}$
(T-abs)	$\frac{\Gamma, x : s; \Pi \vdash e : t}{\Gamma; \Pi \vdash \lambda x. e : (s \rightarrow t)}$
(T-app)	$\frac{\Gamma; \Pi \vdash e : s \rightarrow t \quad \Gamma; \Pi \vdash e' : s}{\Gamma; \Pi \vdash (e e') : t}$
(T-pair)	$\frac{\Gamma; \Pi \vdash e_1 : s_1 \quad \Gamma; \Pi \vdash e_2 : s_2}{\Gamma; \Pi \vdash \langle e_1, e_2 \rangle : (s_1 \times s_2)}$
(T-proj)	$\frac{\Gamma; \Pi \vdash e : (s_1 \times s_2)}{\Gamma; \Pi \vdash (\text{proj}_i e) : s_i}$
(T-inj)	$\frac{\Gamma; \Pi \vdash e : s_i}{\Gamma; \Pi \vdash (\text{inj}_i e) : (s_1 + s_2)}$
(T-case)	$\frac{\Gamma; \Pi \vdash e : (s_1 + s_2) \quad \Gamma, x : s_i; \Pi \vdash e_i : s}{\Gamma; \Pi \vdash \text{case } e \text{ of } \text{inj}_1(x). e_1 \parallel \text{inj}_2(x). e_2 : s}$
(T-ret)	$\frac{\Gamma; \Pi \sqcup \ell \vdash e : s}{\Gamma; \Pi \vdash (\eta_\ell e) : T_\ell(s)}$
(T-bind)	$\frac{\Gamma; \Pi \vdash e : T_\ell(s) \quad \Gamma, x : s; \Pi \vdash e' : t \quad \ell \preceq T_\Pi(t)}{\Gamma; \Pi \vdash \text{bind } x = e \text{ in } e' : t}$

that the result of  $e'$  will not leak any information at  $\ell$ , including any information on  $x$ , which will be bound to the result of  $e$  after removing its  $\ell$ -protection at run time.

The key property of this type system, ensuring a form of parametricity [14, 17] or noninterference [10, 2], can be formalized using a type-directed indistinguishability relation over terms,  $e \sim_\ell e' : s$ , defined by the rules below. Roughly,  $e \sim_\ell e' : s$  means that terms  $e$  and  $e'$  of type  $s$  are indistinguishable to level  $\ell$ .

### Indistinguishability relation

(I-unit)	$() \sim_\ell () : \text{unit}$
(I-intersection)	$\frac{e_1 \sim_\ell e'_1 : s_1 \quad e_2 \sim_\ell e'_2 : s_2}{\langle e_1, e_2 \rangle \sim_\ell \langle e'_1, e'_2 \rangle : (s_1 \times s_2)}$
(I-union)	$\frac{e_i \sim_\ell e'_i : s_i}{(\text{inj}_i e_i) \sim_\ell (\text{inj}_i e'_i) : (s_1 + s_2)}$
(I-function)	$\frac{\forall e, e'. e \sim_\ell e' : s \Rightarrow (v e) \sim_\ell (v' e') : t}{v \sim_\ell v' : (s \rightarrow t)}$
(I-monad-1, 2)	$\frac{\ell' \not\leq \ell \quad e \sim_\ell e' : s}{(\eta_{\ell'} e) \sim_\ell (\eta_{\ell'} e') : T_{\ell'}(s)} \quad \frac{e \sim_\ell e' : s}{(\eta_{\ell'} e) \sim_\ell (\eta_{\ell'} e') : T_{\ell'}(s)}$
(I-eval)	$\frac{e \xrightarrow{*} v \quad e' \xrightarrow{*} v' \quad v \sim_\ell v' : s}{e \sim_\ell e' : s}$

Other than (l-monad-1,2), these rules should be fairly straightforward. In addition, (l-monad-1,2) state that  $(\eta_{\ell'} e)$  and  $(\eta_{\ell'} e')$  are indistinguishable to  $\ell$  whenever  $\ell$  is not at least  $\ell'$ , or  $e$  and  $e'$  are indistinguishable to  $\ell$ .

Based on this definition, the type system guarantees that whenever a typed function is applied to  $\ell$ -protected inputs, it will always produce outputs that are indistinguishable to levels that are not at least  $\ell$ .

**Theorem 1 (DCC soundness, cf. [17]).** *If  $\vdash e : T_{\ell}(s) \rightarrow t$ ,  $\vdash e_1 : s$ , and  $\vdash e_2 : s$ , then for any  $\ell'$  such that  $\ell \not\sqsubseteq \ell'$ ,  $(e (\bar{\eta}_{\ell'} e_1)) \sim_{\ell'} (e (\bar{\eta}_{\ell'} e_2)) : t$ .*

### 3 Explicit flows and DCC<sup>D</sup>

While DCC’s guarantee has appealing consequences and is applicable in a variety of settings, in practice it is often difficult to satisfy, as argued in Section 1. Indeed, the notion of dependency captured by DCC can be overly sensitive in certain settings. In this section, we will design a variant of DCC with the aim of capturing a weaker notion of dependency—one that is sensitive to data dependencies but insensitive to control dependencies. Viewed through the lens of information flow, this system will restrict only *explicit* flows of information. We will make this guarantee precise, and argue why it may be useful for security in practice.

#### 3.1 Explicit flows

In their seminal paper on information flow security, Denning and Denning provided an intriguing characterization of explicit flows [8]:

... an explicit flow [of some information  $x$ ] occurs whenever the operations generating it are independent of the value of  $x$ .

Unfortunately, this “definition” was never formalized.<sup>4</sup> We believe that it deserves more attention, since it suggests exactly why explicit flow attacks may appear more interesting than other kinds of attacks in practice. Because the success of explicit flow attacks cannot actually rely on the specific values involved:

- Such attacks must exploit abstract information-flow channels in a program, which often point to logical errors rather than implementation “artifacts”. Indeed, useful dynamic checks—such as those for exception handling and access control—routinely cause implicit flows in practice. Ignoring these channels not only focuses our attention on “more obviously” serious attacks, but also encourages the use of these channels to prevent such attacks.
- Attackers may find explicit flow attacks more rewarding because they can be carried out reliably, without understanding the control structure of the program. For example, if the attacker can write some value to a trusted location, it can write any value to that location instead of worrying how to

<sup>4</sup> In fact, even in the original paper the distinction between explicit and implicit flows was mentioned only in passing, with no further reference in the rest of the paper.

influence that value. Conversely, if the attacker can read some value from a secret location, it can read any value from that location instead of worrying how to infer that value.

This may explain why several recent analyses for security have—by design—ignored implicit flow attacks and focused on eliminating explicit flow attacks [7, 16, 6]. Some of these analyses aim to verify the security of web applications [16]. Many attacks in this context are ultimately due to code injection, and a satisfactory defense against such attacks is to sanitize values that may flow from inputs to outputs; the sanitization is usually implemented by validating or transforming such values somewhere along the input/output path. Note that these sanitization mechanisms merely restrict explicit flows—they may well introduce implicit flows, but such flows are considered benign in this context. Some other analyses aim to formalize security guarantees provided by low-level systems such as file and operating systems [4, 7, 6], which are usually protected by dynamic access control mechanisms. Preventing explicit flow attacks with these mechanisms already requires some care, and it seems difficult and perhaps undesirable to expect stronger guarantees from such systems. That said, strong information-flow guarantees are necessary in many settings, and it is worthwhile to explore whether such guarantees can be enforced in combination with weaker guarantees as needed, in safe and hopefully useful ways.

### 3.2 DCC<sup>D</sup>

Our system, DCC<sup>D</sup>, is a simple variant of DCC where the type constructors  $T_\ell$  are replaced by  $\overline{T}_\ell$ , and the protection mechanisms  $\eta_\ell$  are replaced by  $\overline{\eta}_\ell$ . These replacements are intended to provide weaker guarantees than their counterparts in DCC, as discussed above; we enforce them with a slightly different set of rules, which require a new form of type  $s^\ell$ , called an open type.

#### Syntax

$\begin{aligned} \text{types } s, t &::= \text{unit} \mid (s \times t) \mid (s + t) \mid (s \rightarrow t) \mid \overline{T}_\ell(s) \mid s^\ell \\ \text{terms } e, v &::= () \mid \langle e, e' \rangle \mid (\text{proj}_i e) \mid (\text{inj}_i e) \mid \text{case } e \text{ of } \text{inj}_1(x). e_1 \parallel \text{inj}_2(x). e_2 \\ &\mid \lambda x. e \mid (e e') \mid (\overline{\eta}_\ell e) \mid \text{bind } x = e \text{ in } e' \end{aligned}$
--

Open types do not have any special introduction or elimination forms. Instead they *qualify* existing types [9], according to the following equations.

#### Open type equations

$\begin{aligned} \text{(E-open-1, 2)} \quad & (s^\ell)^{\ell'} = s^{\ell \sqcup \ell'} \quad , \quad s = s^\perp \\ \text{(E-unit)} \quad & \text{unit}^\ell = \text{unit} \\ \text{(E-intersection)} \quad & (s \times t)^\ell = (s^\ell \times t^\ell) \\ \text{(E-function)} \quad & (s \rightarrow t)^\ell = s \rightarrow t^\ell \\ \text{(E-effect-1, 2)} \quad & \overline{T}_{\ell'}(s)^\ell = \overline{T}_{\ell'}(s^\ell) \quad , \quad \ell \sqsubseteq \ell' \Rightarrow \overline{T}_{\ell'}(s)^\ell = \overline{T}_{\ell'}(s) \end{aligned}$
--

These rules may be explained as follows. Intuitively, the type  $s^\ell$  is given to terms of type  $s$  that need to be (weakly) protected at level  $\ell$ . (E-open-1) allows such protection requirements to be joined with  $\sqcup$ , and (E-open-2) allows any type to be viewed as an open type with no protection requirement. The remaining rules are analogous to DCC’s protection rules. For terms of open unit type  $\text{unit}^\ell$ , the protection requirement  $\ell$  is redundant—it may be dropped as needed. For terms of open intersection type  $(s \times t)^\ell$ , the protection requirement  $\ell$  can be propagated to their projections. Similarly, for terms of open function type  $(s \rightarrow t)^\ell$ , the protection requirement  $\ell$  can be propagated to their bodies. Finally, for terms of open protected type  $\overline{T}_{\ell'}(s)^\ell$ , the protection requirement  $\ell$  can be propagated to their payloads, and can be dropped if  $\ell'$  is at least  $\ell$ .

Continuing the analogy with DCC’s protection rules, there is no equation for (open) union types. In particular, it would be unsafe to equate the open union type  $(s + t)^\ell$  with the union type  $(s^\ell + t^\ell)$ , for reasons similar to those discussed in Section 2. It suffices to see that such an equation would imply the following:

$$(\text{unit} + \text{unit})^\ell = (\text{unit}^\ell + \text{unit}^\ell) = (\text{unit} + \text{unit})$$

As explained in Section 2, the type  $(\text{unit} + \text{unit})$  can serve as an encoding of boolean; so the equation above would allow protection requirements on booleans to be dropped as needed. In general, this would make protection requirements on any data redundant, and completely defeat the purpose of open types.

Note that by viewing the equations above as rewrite rules from left to right, it is possible to “normalize” types, effectively pushing the protection requirements that occur in those types as inwards as possible. Such normalization helps maintain syntax-directed types for most terms, except those that have (open) union types. For the latter terms, we assume they always have open union types, using (E-open-2) if needed.

Our enforcement strategy with open types is rather simple. After removing the protection from a term of type  $\overline{T}_\ell(s)$ , we will give it a type  $s^\ell$ . We will then demand that such a term be protected back with a level  $\ell'$  that is at least  $\ell$ . (See the rules below.) The resulting term will have type  $\overline{T}_{\ell'}(s^\ell)$ , which can be equated to  $\overline{T}_{\ell'}(s)$ , thereby removing the protection requirement.

For this purpose, as in DCC, we define a predicate  $\ell \leq t$ , read as “ $t$  is weakly protected at  $\ell$ ”. The rules closely follow those for  $\ell \preceq t$ , and further include a rule for union types. Intuitively, the latter rule is safe in this context, because we are only interested in tracking data dependencies and not control dependencies.

### Protection rules

---


$$\begin{aligned}
& \text{(P}^{\text{D}}\text{-unit)} \quad \ell \leq \text{unit} \\
& \text{(P}^{\text{D}}\text{-intersection)} \quad \ell \leq s \wedge \ell \leq t \Rightarrow \ell \leq (s \times t) \\
& \text{(P}^{\text{D}}\text{-function)} \quad \ell \leq t \Rightarrow \ell \leq (s \rightarrow t) \\
& \text{(P}^{\text{D}}\text{-effect)} \quad \ell \sqsubseteq \ell' \Rightarrow \ell \leq \overline{T}_{\ell'}(s) \quad , \quad \ell \leq s \Rightarrow \ell \leq \overline{T}_{\ell'}(s) \\
& \text{(P}^{\text{D}}\text{-union)} \quad \ell \leq s \wedge \ell \leq t \Rightarrow \ell \leq (s + t)
\end{aligned}$$


---

**Figure 2: Typing rules (DCC<sup>D</sup>)**

---

(T <sup>D</sup> -var)	$\Gamma, x : s, \Gamma'; \overline{\Pi} \vdash x : s$
(T <sup>D</sup> -unit)	$\Gamma \vdash () : \text{unit}$
(T <sup>D</sup> -abs)	$\frac{\Gamma, x : s; \overline{\Pi} \vdash e : t}{\Gamma; \overline{\Pi} \vdash \lambda x. e : (s \rightarrow t)}$
(T <sup>D</sup> -app)	$\frac{\Gamma; \overline{\Pi} \vdash e : s \rightarrow t \quad \Gamma; \overline{\Pi} \vdash e' : s}{\Gamma; \overline{\Pi} \vdash (e e') : t}$
(T <sup>D</sup> -pair)	$\frac{\Gamma; \overline{\Pi} \vdash e_1 : s_1 \quad \Gamma; \overline{\Pi} \vdash e_2 : s_2}{\Gamma; \overline{\Pi} \vdash \langle e_1, e_2 \rangle : (s_1 \times s_2)}$
(T <sup>D</sup> -proj)	$\frac{\Gamma; \overline{\Pi} \vdash e : (s_1 \times s_2)}{\Gamma; \overline{\Pi} \vdash (\text{proj}_i e) : s_i}$
(T <sup>D</sup> -inj)	$\frac{\Gamma; \overline{\Pi} \vdash e : s_i}{\Gamma; \overline{\Pi} \vdash (\text{inj}_i e) : (s_1 + s_2)}$
(T <sup>D</sup> -case)	$\frac{\Gamma; \overline{\Pi} \vdash e : (s_1 + s_2)^\ell \quad \Gamma, x : s_i^\ell; \overline{\Pi} \vdash e_i : s}{\Gamma; \overline{\Pi} \vdash \text{case } e \text{ of } \text{inj}_1(x). e_1 \parallel \text{inj}_2(x). e_2 : s}$
(T <sup>D</sup> -ret)	$\frac{\Gamma; \overline{\Pi} \sqcup \ell \vdash e : s}{\Gamma; \overline{\Pi} \vdash (\overline{\eta}_\ell e) : \overline{T}_\ell(s)}$
(T <sup>D</sup> -bind)	$\frac{\Gamma; \overline{\Pi} \vdash e : \overline{T}_\ell(s) \quad \Gamma, x : s^\ell; \overline{\Pi} \vdash e : t \quad \ell \leq \overline{T}_{\overline{\Pi}}(t)}{\Gamma; \overline{\Pi} \vdash \text{bind } x = e \text{ in } e' : t}$

---

The explanations for the rules other than (P<sup>D</sup>-union) are the same as those of their counterparts in Section 2, replacing the word “information” with “data”.<sup>5</sup> In addition, (P<sup>D</sup>-union) states that terms of type  $(s + t)$ —which evaluate to terms of the form  $(\text{inj}_i e)$ —cannot leak any more data than  $e$ , which has type  $s$  or  $t$ . On the surface, this seems to conflict with the view that case constructors are the (only) carriers of information in terms. Note, however, that we care only about explicit flows in DCC<sup>D</sup>—in particular, the constructors  $\text{inj}_i$  leak data only if they explicitly flow from protected contexts. However, this implies that the terms  $(\text{inj}_i e)$  themselves require protection, which in turn implies that those terms must have a non-trivial open type (where the qualifier is not  $\perp$ ); and this is impossible since by the equations above,  $(s + t)$  cannot be equal to such a type.

Finally, note that by assumption, open types are not protected, so there is no corresponding rule for such types.

Figure 2 shows the typing rules for DCC<sup>D</sup>. As in DCC, judgments are of the form  $\Gamma; \overline{\Pi} \vdash e : t$ , where  $\Gamma$  contains type hypotheses for free variables and  $\overline{\Pi}$  is a (weak) protection context. All rules other than (T<sup>D</sup>-case) are syntax-directed, thanks to normalization of types as mentioned above. (T<sup>D</sup>-case) is slightly non-standard. It assumes that the case construction  $(\text{inj}_i e)$  has an open union type, and propagates its protection requirement to the variable  $x$  bound to  $e$  at run

<sup>5</sup> Note that information leaked by explicit flow is, in fact, data.

time. This allows sensitive data to be safely deconstructed, without losing track of its protection requirements.

The other rules follow those in DCC. ( $\mathbb{T}^D$ -ret) states that  $(\overline{\eta}_\ell e)$  has type  $\overline{T}_\ell(s)$  whenever  $e$  has type  $s$ ; the latter derivation may assume that  $e$  will be (weakly) protected at  $\ell$  by its context, which is made evident by joining  $\ell$  with the protection context. ( $\mathbb{T}^D$ -bind) states that  $\text{bind } x = e \text{ in } e'$  has type  $t$  only if  $e$  has a type of the form  $\overline{T}_\ell(s)$  and  $e'$  has type  $t$  assuming that  $x$  has open type  $s^\ell$ ; furthermore, the type  $\overline{T}_{\overline{\Pi}}(t)$  must be (weakly) protected at  $\ell$ . This means that either  $t$  must be protected at  $\ell$ , or  $\overline{\Pi}$  must be at least  $\ell$ . In any case, this ensures that the result of  $e'$  will not leak any data at  $\ell$ , including any data in  $x$ , which will be bound to the result of  $e$  after removing its  $\ell$ -protection at run time. The remaining rules are standard.

We formalize the key property of this type system using a type-directed *underivability* relation over terms,  $e \triangleright_\ell : s$ , defined by the rules below. Roughly,  $e \triangleright_\ell : s$  means that term  $e$  of type  $s$  is underivable at level  $\ell$ .

### Underivability relation

(U-unit)	$() \triangleright_\ell : \text{unit}$
(U-intersection)	$\frac{e_1 \triangleright_\ell : s_1 \quad e_2 \triangleright_\ell : s_2}{\langle e_1, e_2 \rangle \triangleright_\ell : (s_1 \times s_2)}$
(U-union)	$\frac{e_i \triangleright_\ell : s_i}{(\text{inj}_i e_i) \triangleright_\ell : (s_1 + s_2)}$
(U-function)	$\frac{\forall e. e \triangleright_\ell : s \Rightarrow (v e) \triangleright_\ell : t}{v \triangleright_\ell : (s \rightarrow t)}$
(U-effect-1, 2)	$\frac{\ell' \not\sqsubseteq \ell}{(\overline{\eta}_{\ell'} e) \triangleright_\ell : \overline{T}_{\ell'}(s)} \quad \frac{e \triangleright_\ell : s}{(\overline{\eta}_{\ell'} e) \triangleright_\ell : \overline{T}_{\ell'}(s)}$
(U-eval)	$\frac{e \xrightarrow{*} v \quad v \triangleright_\ell : s}{e \triangleright_\ell : s}$

These rules closely follow those defining the indistinguishability relation in Section 2, except that here we are concerned with properties of a single term rather than a pair of terms. (U-monad-1,2) state that  $(\overline{\eta}_{\ell'} e)$  is underivable at  $\ell$  whenever  $\ell$  is not at least  $\ell'$ , or  $e$  is underivable at  $\ell$ .

Based on this definition, the type system guarantees that whenever a typed function is applied to a (weakly)  $\ell$ -protected input, it will always produce an output that is underivable at levels that are not at least  $\ell$ .

**Theorem 2 (DCC<sup>D</sup> soundness).** *If  $\vdash e : \overline{T}_\ell(s) \rightarrow t$  and  $\vdash e' : s$ , then for any  $\ell'$  such that  $\ell \not\sqsubseteq \ell'$ ,  $(e (\overline{\eta}_{\ell'} e')) \triangleright_{\ell'} : t$ .*

Furthermore, we show that DCC<sup>D</sup>'s type system is weaker than DCC's, by defining an appropriate translation between the two systems.

**Theorem 3 (DCC to DCC<sup>D</sup>).** *Let  $\|\cdot\|$  translate terms and types by replacing  $(\eta_\ell \cdot)$  by  $(\overline{\eta}_\ell \cdot)$ , and  $T_\ell(\cdot)$  by  $\overline{T}_\ell(\cdot)$ . If  $\vdash e : s$  in DCC then  $\vdash \|e\| : \|s\|$  in DCC<sup>D</sup>.*

### 3.3 A formal characterization of explicit flows in $DCC^D$

Whereas  $DCC^D$  restricts the flow of some data  $x$  without caring about possible leakage of information on  $x$  through case analysis, one may wonder to what extent this restriction actually captures the “definition” of explicit flow by Denning and Denning [8]—that an explicit flow of  $x$  must be generated by operations that are independent of the value of  $x$ . Before we close this section, let us try to make this connection more precise.

Note that the only operation that actually cares about the value of  $x$  is case analysis on  $x$ . Consider an alternative semantics of case analysis, that branches nondeterministically without caring about the relevant case constructor.

$$\text{case } (\text{inj}_- e) \text{ of } \text{inj}_1(z). e_1 \parallel \text{inj}_2(z). e_2 \longrightarrow e_i[e/z]$$

The resulting nondeterministic reduction relation over terms can be lifted to a deterministic reduction relation  $\Longrightarrow$  over sets of terms. If  $\vdash e : s$  (in either  $DCC$  or  $DCC^D$ ) and  $\{e\} \Longrightarrow^* \mathcal{E}$ , then we are guaranteed that for every  $e' \in \mathcal{E}$ , we will have  $\vdash e' : s$ . Moreover, we can lift the indistinguishability relation  $\sim$  to sets of terms as follows:

$$\mathcal{E} \approx_\ell \mathcal{E}' : s \text{ iff for all } e \in \mathcal{E} \text{ and } e' \in \mathcal{E}', e \approx_\ell e' : s$$

**Indistinguishability relation, lifted with  $\Longrightarrow$**

(  <sup>D</sup> -unit)	$() \approx_\ell () : \text{unit}$
(  <sup>D</sup> -intersection)	$\frac{e_1 \approx_\ell e'_1 : s_1 \quad e_2 \approx_\ell e'_2 : s_2}{\langle e_1, e_2 \rangle \approx_\ell \langle e'_1, e'_2 \rangle : (s_1 \times s_2)}$
(  <sup>D</sup> -union)	$\frac{e_i \approx_\ell e'_i : s_i}{(\text{inj}_i e_i) \approx_\ell (\text{inj}_i e'_i) : (s_1 + s_2)}$
(  <sup>D</sup> -function)	$\frac{\forall e, e'. e \approx_\ell e' : s \Rightarrow (v e) \approx_\ell (v' e') : t}{v \approx_\ell v' : (s \rightarrow t)}$
(  <sup>D</sup> -monad-1, 2)	$\frac{\ell' \not\sqsubseteq \ell \quad e \approx_\ell e' : s}{(\eta_{\ell'} e) \approx_\ell (\eta_{\ell'} e') : T_{\ell'}(s)} \quad \frac{e \approx_\ell e' : s}{(\eta_{\ell'} e) \approx_\ell (\eta_{\ell'} e') : T_{\ell'}(s)}$
(  <sup>D</sup> -eval)	$\frac{\{e\} \Longrightarrow^* \mathcal{E} \quad \{e'\} \Longrightarrow^* \mathcal{E}' \quad \mathcal{E} \approx_\ell \mathcal{E}' : s}{e \approx_\ell e' : s}$

We are now ready to define explicit flows. Intuitively, an information flow is explicit if it is not eliminated when we move to the alternative, nondeterministic semantics for case analysis. This captures the essence of Denning and Denning’s informal definition in [8].

**Definition 1 (Explicit flow).** *An information flow of  $x$  in  $e$  exists if  $e \sim e : s$  and there are terms  $e_1$  and  $e_2$  such that  $e[e_1/x] \not\sim_\ell e[e_2/x] : s$ . The flow is explicit if  $\vdash e[e_1/x] \not\sim_\ell e[e_2/x] : s$ .*

We prove that there is a tight correspondence between this definition of explicit flow and  $DCC^D$ ’s guarantee.

**Theorem 4 (Explicit flow and underivability).** *There is an explicit flow of  $x$  in  $e$  if and only if  $e \triangleright_\ell : s$  and there is a term  $e'$  such that  $e[e'/x] \not\triangleright_\ell : s$ .*

## 4 Dynamic weakening in $\text{DCC}^{\text{DC}}$

While  $\text{DCC}^{\text{D}}$ -style protection may be useful in some settings, we believe that it is best viewed as a “backdoor”, rather than an alternative to DCC-style protection. Indeed, DCC enjoys better theoretical foundations and promises many desirable properties that  $\text{DCC}^{\text{D}}$  cannot. In practice, we should be able to mix  $\text{DCC}^{\text{D}}$ -style protection carefully with DCC-style protection as needed, and still be able to reason precisely about the guarantees of the resulting systems, short of weakening all the guarantees provided by DCC-style protection. We investigate these issues in the setting of a hybrid language  $\text{DCC}^{\text{DC}}$ .

### 4.1 $\text{DCC}^{\text{DC}}$

$\text{DCC}^{\text{DC}}$ 's syntax and typing rules are obtained by merging those of DCC and  $\text{DCC}^{\text{D}}$ . The inheritance is mostly straightforward; we make a few adjustments to encourage the two subsystems to interact. (The full system is available for reference in Appendix A.) First, we carry both kinds of protection contexts in typing judgments, and modify the DCC rule (T-ret) as follows.

$$(\text{T}^{\text{DC}}\text{-ret-1}) \quad \frac{\Gamma; \Pi \sqcup \ell; \overline{\Pi} \sqcup \ell \vdash e : s}{\Gamma; \Pi; \overline{\Pi} \vdash (\eta_\ell e) : T_\ell(s)}$$

Thus, any DCC-style protection provided by the context is made evident not only in its usual protection context, but also in the weak protection context. Next, we add the following protection rules and open type equations.

$$\begin{array}{ll} (\text{P-effect}) & \ell \preceq s \Rightarrow \ell \preceq \overline{T}_{\ell'}(s) \\ (\text{P}^{\text{D}}\text{-monad}) & \ell \leq \overline{T}_{\ell'}(s) \Rightarrow \ell \leq T_{\ell'}(s) \\ (\text{E-monad}) & \overline{T}_{\ell'}(s)^\ell = \overline{T}_{\ell'}(s^\ell) \Rightarrow T_{\ell'}(s)^\ell = T_{\ell'}(s^\ell) \end{array}$$

These rules can be explained as follows. On the one hand, (P-effect) conservatively assumes that terms of type  $\overline{T}_{\ell'}(s)$ —which evaluate to weakly protected terms—can leak just as much information as their payloads, which are of type  $s$ . On the other hand, (P<sup>D</sup>-monad) considers the type  $T_{\ell'}(s)$  to be weakly protected whenever the weaker type  $\overline{T}_{\ell'}(s)$  is so protected; this internalizes the fact that  $\text{DCC}^{\text{D}}$ 's types provide weaker protection than DCC's types, as shown in Theorem 3. (E-monad) is based on similar reasoning. In particular, these rules admit functions of type  $\overline{T}_\ell(s) \rightarrow T_\ell(s)$ , that can be used to *strengthen* protection on terms:

$$\lambda x. \text{bind } y = x \text{ in } (\eta_\ell y)$$

Finally, we coalesce the introduction and elimination rules for unit, intersection, union, and function types; in particular we have:

$$(\text{T}^{\text{DC}}\text{-case}) \quad \frac{\Gamma; \Pi; \overline{\Pi} \vdash e : (s_1 + s_2)^\ell \quad \Gamma, x : s_i^\ell; \Pi; \overline{\Pi} \vdash e_i : s}{\Gamma; \Pi; \overline{\Pi} \vdash \text{case } e \text{ of } \text{inj}_1(x). e_1 \parallel \text{inj}_2(x). e_2 : s}$$

Fortunately, we are able to show that the resp. guarantees of DCC and  $\text{DCC}^D$  are preserved in  $\text{DCC}^{DC}$ . Of course, this requires adding the appropriate rules to the definitions of the indistinguishability and underivability relations, as follows.

$$\begin{array}{l}
\text{(I-effect)} \quad \frac{e \sim_{\ell} e' : s}{(\overline{\eta}_{\ell'} e) \sim_{\ell} (\overline{\eta}_{\ell'} e') : \overline{T}_{\ell'}(s)} \\
\text{(U-monad)} \quad \frac{(\overline{\eta}_{\ell'} e) \triangleright_{\ell} : \overline{T}_{\ell'}(s)}{(\eta_{\ell'} e) \triangleright_{\ell} : T_{\ell'}(s)}
\end{array}$$

**Theorem 5 ( $\text{DCC}^{DC}$  soundness, preliminary).** *Theorems 1 and 2 continue to hold in  $\text{DCC}^{DC}$ .*

## 4.2 A weakening primitive

Next we include a `weaken` primitive in  $\text{DCC}^{DC}$ , which will act as a further bridge between the two subsystems (going in the opposite direction as the strengthening functions above). Specifically, our intention is that `weaken` will allow terms of type  $T_{\ell}(s)$  to be viewed as terms of type  $\overline{T}_{\ell}(s)$ . (Later, we will force some overhead on this allowance.)

Unsurprisingly, using `weaken` may invalidate the protection guarantees provided by DCC's types. As a simple example, consider the following function:

$$h = \lambda x. \text{bind } y = (\text{weaken } x) \text{ in case } y \text{ of inj}_1(z). (\text{inj}_1 ()) \parallel \text{inj}_2(z). (\text{inj}_2 ())$$

Assuming a typing rule that allows `(weaken e)` to have type  $\overline{T}_{\ell}(s)$  whenever  $e$  has type  $T_{\ell}(s)$ , this function can be typed  $T_{\ell}(\text{unit} + \text{unit}) \rightarrow (\text{unit} + \text{unit})$ . However,  $h$  clearly has an information flow violation; formally, we have that  $(h (\eta_{\ell} (\text{inj}_1 ()))) \not\sim_{\ell} (h (\eta_{\ell} (\text{inj}_2 ())))$ , which contradicts Theorem 5. Worse,  $h$  can be used as an oracle to generate more complex counterexamples. Consider the following functions:

$$\begin{aligned}
m &= \lambda x. (\eta_{\ell} (\text{bind } y = x \text{ in case } y \text{ of inj}_1(z). (\text{inj}_1 ()) \parallel \text{inj}_2(z). (\text{inj}_2 ()))) \\
n &= \lambda x. (h (m x))
\end{aligned}$$

The function  $m$ , which can be typed  $T_{\ell'}(s+t) \rightarrow T_{\ell}(\text{unit} + \text{unit})$  even in DCC as long as  $\ell' \sqsubseteq \ell$ , does not leak information on  $x$  per se; it derives a bit of information on  $x$  and protects that bit before returning it. Still, the function  $n$  with type  $T_{\ell'}(s+t) \rightarrow (\text{unit} + \text{unit})$  is able to use  $m$  in combination with  $h$  to leak that bit.

As this example suggests, using `weaken` at level  $\ell$  in a program may invalidate DCC-style guarantees for all types protected by levels  $\ell$  and lower. However, weaker  $\text{DCC}^D$ -style guarantees should still hold for such types (because there is no way to get around  $\text{DCC}^D$ 's typing rules). Moreover, assuming that there are no other uses of `weaken` in the program, we expect that stronger DCC-style guarantees should remain valid for all other types. The reason is that such types, which are protected by levels higher or incomparable to  $\ell$ , will never delegate the responsibility of protection to the weakened types. In sum, we can precisely reason about protection in this system as long as we carefully track the uses of `weaken` in the program.

Curiously enough, such an analysis can be viewed as a special case of DCC’s dependency analysis, just like many other applications of DCC. Indeed, the original motivation for studying DCC was its ability to express various program analyses—including call tracking, slicing, partial evaluation, as well as information-flow control—in a uniform setting. Our analysis is similar in spirit, and can be expressed by recycling DCC’s types to carry *blames* for weakening.

Specifically, we consider a lattice of blames that is isomorphic to the lattice of levels, *i.e.*, for each level  $\ell$  we have a label  $\theta(\ell)$  that denotes a blame, where  $\theta$  is a fixed lattice isomorphism. Then, instead of the naïve typing rule for `weaken` above, we include the following rule:

$$(\mathbb{T}^{\text{DC}}\text{-weaken}) \quad \frac{\Gamma; \Pi; \overline{\Pi} \vdash e : T_\ell(s)}{\Gamma; \Pi; \overline{\Pi} \vdash (\text{weaken } e) : T_{\theta(\ell)}(\overline{T}_\ell(s))}$$

Intuitively, this means that whenever we use `weaken` to view terms of type  $T_\ell(s)$  as terms of type  $\overline{T}_\ell(s)$  in a program, we simultaneously blame  $\theta(\ell)$  for facilitating such a view. While this allows us to get away with weaker protection requirements on such terms, it also forces an overhead: the blame must be carried around whenever a result depends on those terms. Fortunately, DCC’s typing rules can enforce this overhead for free.

A reassuring interpretation of this overhead is obtained through the lens of the Curry-Howard isomorphism, following a recent reading of DCC as a logic for access control [1]. Specifically, we can interpret the blame  $\theta(\ell)$  as a principal that controls protection requirements at level  $\ell$ , and rewrite the type of `(weaken  $e$ )` as  $\theta(\ell)$  *says*  $(\overline{T}_\ell(s))$ . Using the logic, we can now pinpoint the principals whose statements may have influenced protection requirements in a program, resting assured that the protection guarantees at other levels will not be influenced by these statements.

### 4.3 Blame orderings

Note that we have not yet specified how the ordering in the blame lattice should be related to  $\sqsubseteq$ . One interesting scenario is where the ordering is the same, so that  $\theta$  preserves joins and meets over the relevant lattices. In this case, the type of a program must carry a blame  $\theta(\ell)$  such that  $\ell$  upper-bounds the levels of weakening on which its results may depend. In other words, DCC-style protection guarantees hold at all levels not  $\ell$  or lower.

Formally, we define the blame  $\Theta(t)$  carried by a program of type  $t$  as the join of all blames that appear in  $t$ . We then prove the following theorem.

**Theorem 6 (DCC<sup>DC</sup> soundness).** *If  $\vdash e : T_\ell(s) \rightarrow t$ ,  $\vdash e_1 : s$ ,  $\vdash e_2 : s$ , and  $\theta(\ell) \not\sqsubseteq \Theta(t)$ , then for any  $\ell'$  such that  $\ell \not\sqsubseteq \ell'$ ,  $(e \ (\eta_\ell \ e_1)) \sim_{\ell'} (e \ (\eta_\ell \ e_2)) : t$ . Moreover, Theorem 2 continues to hold as is in this system.*

An equally interesting scenario is where we flip the ordering in the blame lattice, so that  $\theta$  exchanges joins and meets over the relevant lattices. In this case, the type of a program must carry a blame  $\theta(\ell)$  such that  $\ell$  lower-bounds the levels of weakening on which its results may depend. In other words,  $\ell$  provides a measure of robustness for these guarantees.

## 5 Precise dependency analysis in $DCC^{CD}$

If we had to make a movie out of the story so far, we would likely not win any awards. We have a kingdom run by a blue-blooded tyrant,  $DCC$ ; a rustic rebel,  $DCC^D$ , conquers the kingdom and ushers in some liberalization, but it destabilizes the kingdom so much that the tyrant has to be rushed back in disguise to restore order! Indeed, what this story misses is a climax: can the rebel make a comeback, showing the tyrant that such liberalization will in fact help run his kingdom better?

More seriously, we have seen glimpses of an idea in  $DCC^D$  that can be used to improve  $DCC$ . The idea is to deconstruct information flow control in  $DCC$  into two separate problems: one of restricting explicit flows, and the other of restricting implicit flows. The former is already handled by  $DCC^D$ ; the latter, which is entirely due to case analysis, can be handled by reworking some of the rules for union types in  $DCC$ . Surprisingly, the resulting system,  $DCC^{CD}$ , becomes less conservative than  $DCC$  without compromising its guarantees.

### 5.1 $DCC^{CD}$

The main drawback we are trying to address in  $DCC$  is its extreme caution in the handling of case construction. Indeed, worried about the fact that case constructors may carry information,  $DCC$  restricts both explicit and implicit flows in one shot by placing a severe restriction on union types—namely, they can never be considered protected (see the discussion on  $DCC$ 's protection rules in Section 2). Unfortunately this causes several benign terms to be rejected by  $DCC$  simply because they use case construction.

We relax this restriction by observing that any information leak is ultimately due to either an explicit leak through data flow or an implicit leak through control flow. Specifically, evaluating a term of union type may reveal information about sensitive data only if that term either does a case analysis on sensitive data, or releases the sensitive data itself.

Technically, this separation of concerns was already somewhat evident in  $DCC^D$ . We weakened the protection rules to allow union types to be considered protected (see Section 3). By itself this would be too weak, given the dangerous nature of case constructors—it would allow both explicit and implicit flows. Thus in addition, we employed open types—essentially types with qualifiers to precisely track data flow through programs. In particular, these qualifiers allowed us to restrict explicit flows in the system.

We argue that the same qualifiers would also allow us to restrict implicit flows in the system. Indeed, the case analysis rule needed to accommodate terms with qualified union types, because the qualifiers could be eliminated on all types other than union types—they “stuck” to union types exactly because of the dangerous nature of case constructors! While in  $DCC$  we chose to ignore implicit flows caused by such case analysis, in  $DCC^{CD}$  we will not.

Note that in order to adjust the rule for case analysis to account for implicit flows, we must have some idea of the level of information that we are interested

in protecting—otherwise, we would have to conservatively ban any case analysis. For this purpose, we need to carry an *open context*  $\Sigma$  in typing judgments, which indicates the minimum level of protection required by the context.

To keep matters simple, we do not consider terms of the form (`weaken`  $e$ ) and ( $\overline{\eta}_\ell e$ ) in the language; indeed, on the surface we do not care about  $\text{DCC}^D$ -style protection at all, although  $\text{DCC}^D$ 's type system is an important component of the system internally. Accordingly, we also remove weak protection contexts. The remaining system mostly inherits from  $\text{DCC}^{DC}$ ; we make a few adjustments, discussed below. (The full system is available for reference in Appendix B.)

We now have two typing rules for `bind`, both offering DCC-style protection. The first rule is similar to that in DCC.

$$\text{(T}^{\text{CD}}\text{-bind-1)} \quad \frac{\Gamma; \Pi; \Sigma \vdash e : T_\ell(s) \quad \Gamma, x : s; \Pi; \Sigma \vdash e' : t \quad \ell \preceq T_\Pi(t)}{\Gamma; \Pi; \Sigma \vdash \text{bind } x = e \text{ in } e' : t}$$

The other rule is new, and captures the interaction of the two subsystems.

$$\text{(T}^{\text{CD}}\text{-bind-2)} \quad \frac{\Gamma; \Pi; \Sigma \vdash e : T_\ell(s) \quad \Gamma, x : s^\ell; \Pi; \Sigma \sqcap \ell \vdash e' : t \quad \ell \leq T_\Pi(t)}{\Gamma; \Pi; \Sigma \vdash \text{bind } x = e \text{ in } e' : t}$$

Curiously, this rule looks similar to ( $\text{T}^{\text{D}}\text{-bind}$ ) in  $\text{DCC}^D$ , although functionally it is intended to be closer to ( $\text{T}\text{-bind}$ ) in DCC. The similarities are as important as the differences. Since ( $\text{T}^{\text{CD}}\text{-bind-2}$ ) applies to terms of type  $T_\ell(s)$  instead of  $\overline{T}_\ell(s)$ , it is intended to be more restrictive than ( $\text{T}^{\text{D}}\text{-bind}$ ). Still, we use the weak protection predicate  $\leq$  instead of  $\preceq$ , while assuming an open type for  $x$ . This takes care of explicit flows, but not implicit flows. In addition, to handle implicit flows, we simply meet  $\ell$  with the open context, deferring their actual restriction till we encounter case analysis at level  $\ell$ .

Finally, the new rule for case analysis is as follows.

$$\text{(T}^{\text{CD}}\text{-case)} \quad \frac{\Gamma; \Pi; \Sigma \vdash e : (s_1 + s_2)^\ell \quad \ell \not\sqsubseteq \perp \Rightarrow \Sigma \not\sqsubseteq \ell \quad \Gamma, x : s_i^\ell; \Pi; \Sigma \vdash e_i : s}{\Gamma; \Pi; \Sigma \vdash \text{case } e \text{ of } \text{inj}_1(x). e_1 \parallel \text{inj}_2(x). e_2 : s}$$

As in ( $\text{T}^{\text{D}}\text{-case}$ ), this rule requires—without loss of generality—that  $e$  have an open union type, with some protection requirement  $\ell$ . In addition, it requires that the open context  $\Sigma$  be higher than or incomparable to  $\ell$ —so that any implicit flows at  $\ell$  that may occur through the case analysis are irrelevant to  $\Sigma$ . Note that this requirement does not apply if  $\ell$  is  $\perp$ . Indeed,  $\ell$  may be  $\perp$  for various reasons, and for none of those reasons is this requirement necessary:

- either  $e$  flows from a  $\perp$ -protected term, in which scenario we do not care about protecting  $e$ ;
- or  $e$  actually has type  $(s_1 + s_2)$ , which means that:
  - either it does not flow from a protected term;
  - or its protection requirement was omitted because ( $\text{T}^{\text{CD}}\text{-bind-1}$ ) was applied instead of ( $\text{T}^{\text{CD}}\text{-bind-2}$ ), in which scenario the side condition invoking DCC's protection rules is already sufficient to rule out implicit flows.

With these rules, we show that  $DCC^{CD}$  provides the same guarantees as DCC, and is at least as complete.

**Theorem 7 (DCC<sup>CD</sup> soundness and completeness).** *If  $\vdash e : T_\ell(s) \rightarrow t$ ,  $\vdash e_1 : s$ , and  $\vdash e_2 : s$ , then for any  $\ell'$  such that  $\ell \not\sqsubseteq \ell'$ ,  $(e (\overline{\eta}_\ell e_1)) \sim_{\ell'}$   $(e (\overline{\eta}_\ell e_2)) : t$ . Furthermore, if  $\vdash e' : s'$  in DCC then  $\vdash e' : s'$  in DCC<sup>CD</sup>.*

Of course, the point of defining DCC<sup>CD</sup> is to show that it accepts more programs than DCC. For example, the following functions—rejected by DCC (see Section 1)—have type  $T_\ell(s) \rightarrow (\text{unit} + \text{unit})$  in DCC<sup>CD</sup>:

$$\lambda x. \text{bind } y = x \text{ in } (\text{inj}_i ()) \quad i \in \{1, 2\}$$

Likewise, the following function has type  $T_\ell(s) \rightarrow (s_1 + s_2) \rightarrow (\text{unit} + \text{unit})$  in DCC<sup>CD</sup>, despite being rejected by DCC, and so on:

$$\lambda x. \lambda w. \text{bind } y = x \text{ in case } w \text{ of } \text{inj}_1(z). (\text{inj}_1 ()) \parallel \text{inj}_2(z). (\text{inj}_2 ())$$

## 6 Discussion

One limitation of this work is that we omit any discussion of pointed types and nonterminating programs, unlike DCC [2]. However, we have informally checked that including these elements does not cause any serious problems in our results.

Next, while we have tried to remain close in spirit to Denning and Denning’s characterization of explicit flow in our formal definition of DCC<sup>D</sup>, whether there is indeed perfect harmony is open to debate. In particular, if an insecure information flow is caused by a combination of “implicit” and “explicit” leaks, we prefer to consider such a flow explicit simply because we want DCC<sup>D</sup> to restrict such a flow. Indeed, we label only those flows implicit that can be eliminated by moving to a nondeterministic semantics for case analysis. Whether this classification “agrees” with Denning and Denning’s is, in the end, irrelevant to security.

Finally, it would seem that in a practical implementation of DCC<sup>DC</sup> the overhead of carrying blames would be too painful. However, note that it should not be too difficult to infer the blames required to type a program; and conversely, once a program is typed, the blames can be erased.

There is a huge body of research on noninterference-based security for languages; see [15] for a survey. However, there seems to be a disconnect between this research and most security tools implemented in practice, which ignore implicit flows [11, 16]. Unfortunately, we do not know of any work on formalizing the resulting, trace-based guarantees of such tools, although several security type systems for process calculi have been designed around similar ideas [5, 7, 6].

In the context of security, our results suggest some interesting ways in which noninterference-based security may be reconciled with trace-based security within the same system, improving soundness in one direction and completeness in the other. Specifically, in a system where protection may have been partially weakened, a strong blame analysis can be used to provide strong protection guarantees

for those parts of the system that are not affected by such weakening. Conversely, a weak flow analysis can be used to increase the coverage of such guarantees.

We hope that these results will spur further interest in bridging the gap between these two views of security.

*Acknowledgments* This work benefitted from discussions with Steve Zdancewic, Michael Hicks, Aslan Askarov, and David Mazières.

## References

1. M. Abadi. Access control in a core calculus of dependency. *Electronic Notes in Theoretical Computer Science*, 172:5–31, 2007.
2. M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *POPL*, pages 147–160. ACM, 1999.
3. A. Askarov and A. Sabelfeld. Catch me if you can: Permissive yet secure error handling. In *Workshop on Programming Languages and Analysis for Security*, pages 45–57. ACM, 2009.
4. B. Blanchet and A. Chaudhuri. Automated formal analysis of a protocol for secure file sharing on untrusted storage. In *IEEE Symposium on Security and Privacy*, pages 417–431. IEEE, 2008.
5. L. Cardelli, G. Ghelli, and A. Gordon. Secrecy and group creation. *Information and Computation*, 196(2):127–155, 2005.
6. A. Chaudhuri. Language-based security on Android. In *Workshop on Programming Languages and Analysis for Security*, pages 1–7. ACM, 2009.
7. A. Chaudhuri, P. Naldurg, and S. Rajamani. A type system for data-flow integrity on Windows Vista. *ACM SIGPLAN Notices*, 43(12):9–20, 2009.
8. D. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7), 1977.
9. J. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. *ACM SIGPLAN Notices*, 34(5):192–203, 1999.
10. J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and privacy*, volume 12, 1982.
11. D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can’t live with ’em, can’t live without ’em. In *International Conference on Information Systems Security*, pages 56–70. Springer, 2008.
12. E. Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.
13. A. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 172–186. IEEE, 2004.
14. J. Reynolds. Types, abstraction and parametric polymorphism. *Information processing*, 83(513-523):1, 1983.
15. A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.
16. O. Tripp, S. Fink, and O. Weisman. TAJ: effective taint analysis of web applications. In *PLDI*, pages 87–97. ACM, 2009.
17. S. Tse and S. Zdancewic. Translating dependency into parametricity. *ACM SIGPLAN Notices*, 39(9):115–125, 2004.
18. S. Zdancewic and A. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23. IEEE, 2001.

## Appendix

Below we provide the full definitions of  $\text{DCC}^{DC}$  and  $\text{DCC}^{CD}$  for reference.

### A $\text{DCC}^{DC}$

#### Syntax

---

types  $s, t ::= \text{unit} \mid (s \times t) \mid (s + t) \mid (s \rightarrow t) \mid T_\ell(s) \mid \overline{T}_\ell(s) \mid s^\ell$   
terms  $e, v ::= () \mid \langle e, e' \rangle \mid (\text{proj}_i e) \mid (\text{inj}_i e) \mid \text{case } e \text{ of } \text{inj}_1(x). e_1 \parallel \text{inj}_2(x). e_2$   
 $\mid \lambda x. e \mid (e e') \mid (\eta_\ell e) \mid (\overline{\eta}_\ell e) \mid \text{bind } x = e \text{ in } e'$

---

#### Typing rules

---

$(\text{T}^{\text{DC}}\text{-var})$	$\Gamma, x : s, \Gamma'; \Pi; \overline{\Pi} \vdash x : s$
$(\text{T}^{\text{DC}}\text{-unit})$	$\Gamma; \Pi; \overline{\Pi} \vdash () : \text{unit}$
$(\text{T}^{\text{DC}}\text{-abs})$	$\frac{\Gamma; \Pi; \overline{\Pi} \vdash e : (s \rightarrow t)}{\Gamma; \Pi; \overline{\Pi} \vdash \lambda x. e : (s \rightarrow t)}$
$(\text{T}^{\text{DC}}\text{-app})$	$\frac{\Gamma; \Pi; \overline{\Pi} \vdash e : s \rightarrow t \quad \Gamma; \Pi; \overline{\Pi} \vdash e' : s}{\Gamma; \Pi; \overline{\Pi} \vdash (e e') : t}$
$(\text{T}^{\text{DC}}\text{-pair})$	$\frac{\Gamma; \Pi; \overline{\Pi} \vdash e_1 : s_1 \quad \Gamma; \Pi; \overline{\Pi} \vdash e_2 : s_2}{\Gamma; \Pi; \overline{\Pi} \vdash \langle e_1, e_2 \rangle : (s_1 \times s_2)}$
$(\text{T}^{\text{DC}}\text{-proj})$	$\frac{\Gamma; \Pi; \overline{\Pi} \vdash e : (s_1 \times s_2)}{\Gamma; \Pi; \overline{\Pi} \vdash (\text{proj}_i e) : s_i}$
$(\text{T}^{\text{DC}}\text{-inj})$	$\frac{\Gamma; \Pi; \overline{\Pi} \vdash e : s_i}{\Gamma; \Pi; \overline{\Pi} \vdash (\text{inj}_i e) : (s_1 + s_2)}$
$(\text{T}^{\text{DC}}\text{-case})$	$\frac{\Gamma; \Pi; \overline{\Pi} \vdash e : (s_1 + s_2)^\ell \quad \Gamma, x : s_i^l; \Pi; \overline{\Pi} \vdash e_i : s}{\Gamma; \Pi; \overline{\Pi} \vdash \text{case } e \text{ of } \text{inj}_1(x). e_1 \parallel \text{inj}_2(x). e_2 : s}$
$(\text{T}^{\text{DC}}\text{-ret-1})$	$\frac{\Gamma; \Pi \sqcup \ell; \overline{\Pi} \sqcup \ell \vdash e : s}{\Gamma; \Pi; \overline{\Pi} \vdash (\eta_\ell e) : T_\ell(s)}$
$(\text{T}^{\text{DC}}\text{-ret-2})$	$\frac{\Gamma; \Pi; \overline{\Pi} \vdash (\overline{\eta}_\ell e) : \overline{T}_\ell(s)}{\Gamma; \Pi; \overline{\Pi} \vdash e : T_\ell(s)}$
$(\text{T}^{\text{DC}}\text{-bind-1})$	$\frac{\Gamma; \Pi; \overline{\Pi} \vdash e : T_\ell(s) \quad \Gamma, x : s; \Pi; \overline{\Pi} \vdash e' : t \quad \ell \preceq T_\Pi(t)}{\Gamma; \Pi; \overline{\Pi} \vdash \text{bind } x = e \text{ in } e' : t}$
$(\text{T}^{\text{DC}}\text{-bind-2})$	$\frac{\Gamma; \Pi; \overline{\Pi} \vdash e : \overline{T}_\ell(s) \quad \Gamma, x : s^\ell; \Pi; \overline{\Pi} \vdash e' : t \quad \ell \leq T_\Pi(t)}{\Gamma; \Pi; \overline{\Pi} \vdash \text{bind } x = e \text{ in } e' : t}$
$(\text{T}^{\text{DC}}\text{-weaken})$	$\frac{\Gamma; \Pi; \overline{\Pi} \vdash e : T_\ell(s)}{\Gamma; \Pi; \overline{\Pi} \vdash (\text{weaken } e) : T_{\theta(\ell)}(\overline{T}_\ell(s))}$

---

## Protection rules

---

$$\begin{aligned}
& \text{(P-unit)} \quad \ell \preceq \text{unit} \\
& \text{(P-intersection)} \quad \ell \preceq s \wedge \ell \preceq t \Rightarrow \ell \preceq (s \times t) \\
& \text{(P-function)} \quad \ell \preceq t \Rightarrow \ell \preceq (s \rightarrow t) \\
& \text{(P-monad-1,2)} \quad \ell \sqsubseteq \ell' \Rightarrow \ell \preceq T_{\ell'}(s) \quad , \quad \ell \preceq s \Rightarrow \ell \preceq T_{\ell'}(s) \\
& \text{(P-effect)} \quad \ell \preceq s \Rightarrow \ell \preceq \overline{T}_{\ell'}(s) \\
& \text{(P<sup>D</sup>-unit)} \quad \ell \leq \text{unit} \\
& \text{(P<sup>D</sup>-intersection)} \quad \ell \leq s \wedge \ell \leq t \Rightarrow \ell \leq (s \times t) \\
& \text{(P<sup>D</sup>-function)} \quad \ell \leq t \Rightarrow \ell \leq (s \rightarrow t) \\
& \text{(P<sup>D</sup>-effect)} \quad \ell \sqsubseteq \ell' \Rightarrow \ell \leq \overline{T}_{\ell'}(s) \quad , \quad \ell \leq s \Rightarrow \ell \leq \overline{T}_{\ell'}(s) \\
& \text{(P<sup>D</sup>-union)} \quad \ell \leq s \wedge \ell \leq t \Rightarrow \ell \leq (s + t) \\
& \text{(P<sup>D</sup>-monad)} \quad \ell \leq \overline{T}_{\ell'}(s) \Rightarrow \ell \leq T_{\ell'}(s)
\end{aligned}$$


---

## Open type equations

---

$$\begin{aligned}
& \text{(E-open-1,2)} \quad (s^\ell)^{\ell'} = s^{\ell \sqcup \ell'} \quad , \quad s = s^\perp \\
& \text{(E-unit)} \quad \text{unit}^\ell = \text{unit} \\
& \text{(E-intersection)} \quad (s \times t)^\ell = (s^\ell \times t^\ell) \\
& \text{(E-function)} \quad (s \rightarrow t)^\ell = s \rightarrow t^\ell \\
& \text{(E-effect-1,2)} \quad \overline{T}_{\ell'}(s)^\ell = \overline{T}_{\ell'}(s^\ell) \quad , \quad \ell \sqsubseteq \ell' \Rightarrow \overline{T}_{\ell'}(s)^\ell = \overline{T}_{\ell'}(s) \\
& \text{(E-monad)} \quad \overline{T}_{\ell'}(s)^\ell = \overline{T}_{\ell'}(s^\ell) \Rightarrow T_{\ell'}(s)^\ell = T_{\ell'}(s^\ell)
\end{aligned}$$


---

## B DCC<sup>CD</sup>

### Syntax

---

*types*  $s, t ::= \text{unit} \mid (s \times t) \mid (s + t) \mid (s \rightarrow t) \mid T_\ell(s) \mid \overline{T}_\ell(s) \mid s^\ell$   
*terms*  $e, v ::= () \mid \langle e, e' \rangle \mid (\text{proj}_i e) \mid (\text{inj}_i e) \mid \text{case } e \text{ of } \text{inj}_1(x). e_1 \parallel \text{inj}_2(x). e_2$   
 $\mid \lambda x. e \mid (e e') \mid (\eta_\ell e) \mid \text{bind } x = e \text{ in } e'$

---

## Typing rules

(T <sup>CD</sup> -var)	$\Gamma, x : s, \Gamma'; \Pi; \Sigma \vdash x : s$
(T <sup>CD</sup> -unit)	$\Gamma; \Pi; \Sigma \vdash () : \text{unit}$
(T <sup>CD</sup> -abs)	$\frac{\Gamma, x : s; \Pi; \Sigma \vdash e : t}{\Gamma; \Pi; \Sigma \vdash \lambda x. e : (s \rightarrow t)}$
(T <sup>CD</sup> -app)	$\frac{\Gamma; \Pi; \Sigma \vdash e : s \rightarrow t \quad \Gamma; \Pi; \Sigma \vdash e' : s}{\Gamma; \Pi; \Sigma \vdash (e e') : t}$
(T <sup>CD</sup> -pair)	$\frac{\Gamma; \Pi; \Sigma \vdash e_1 : s_1 \quad \Gamma; \Pi; \Sigma \vdash e_2 : s_2}{\Gamma; \Pi; \Sigma \vdash \langle e_1, e_2 \rangle : (s_1 \times s_2)}$
(T <sup>CD</sup> -proj)	$\frac{\Gamma; \Pi; \Sigma \vdash e : (s_1 \times s_2)}{\Gamma; \Pi; \Sigma \vdash (\text{proj}_i e) : s_i}$
(T <sup>CD</sup> -inj)	$\frac{\Gamma; \Pi; \Sigma \vdash e : s_i}{\Gamma; \Pi; \Sigma \vdash (\text{inj}_i e) : (s_1 + s_2)}$
(T <sup>CD</sup> -case)	$\frac{\Gamma; \Pi; \Sigma \vdash e : (s_1 + s_2)^\ell \quad \ell \not\sqsubseteq \perp \Rightarrow \Sigma \not\sqsubseteq \ell \quad \Gamma, x : s_i^\ell; \Pi; \Sigma \vdash e_i : s}{\Gamma; \Pi; \Sigma \vdash \text{case } e \text{ of } \text{inj}_1(x). e_1 \parallel \text{inj}_2(x). e_2 : s}$
(T <sup>CD</sup> -ret)	$\frac{\Gamma; \Pi \sqcup \ell; \Sigma \vdash e : s}{\Gamma; \Pi; \Sigma \vdash (\eta_\ell e) : T_\ell(s)}$
(T <sup>CD</sup> -bind-1)	$\frac{\Gamma; \Pi; \Sigma \vdash e : T_\ell(s) \quad \Gamma, x : s; \Pi; \Sigma \vdash e' : t \quad \ell \preceq T_\Pi(t)}{\Gamma; \Pi; \Sigma \vdash \text{bind } x = e \text{ in } e' : t}$
(T <sup>CD</sup> -bind-2)	$\frac{\Gamma; \Pi; \Sigma \vdash e : T_\ell(s) \quad \Gamma, x : s^\ell; \Pi; \Sigma \sqcap \ell \vdash e' : t \quad \ell \leq T_\Pi(t)}{\Gamma; \Pi; \Sigma \vdash \text{bind } x = e \text{ in } e' : t}$

The protection rules and open type equations are the same as those for  $\text{DCC}^{DC}$ .