Liberalizing Dependency

Avik Chaudhuri

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

Abstract. The dependency core calculus (DCC), a simple extension of the computational lambda calculus, 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, enhancing soundness of the latter and completeness of the former.

1 Introduction

The dependency core calculus (DCC) [2] is a simple extension of the computational lambda calculus [20], where each level ℓ in a lattice is associated with a type constructor T_ℓ 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 [22, 29] and noninterference [2, 16]. Roughly, DCC's type system guarantees that the computational effects of a program protected by some level ℓ can only be observed by programs protected by levels ℓ or higher in the lattice. Of course, such effects may include not only explicit effects due to data flow, but also implicit effects due to control flow. For example, consider the following functions.

$$f = \lambda x : T_\ell(s_1 + s_2).\ \mathsf{bind}\ y = x\ \mathsf{in}\ y$$
$$g = \lambda x : T_\ell(s_1 + s_2).\ \mathsf{bind}\ y = x\ \mathsf{in}\ \mathsf{case}\ y\ \mathsf{of}\ \mathsf{inj}_1(z).\ (\mathsf{inj}_1\ ())\ \|\ \mathsf{inj}_2(z).\ (\mathsf{inj}_2\ ())$$

The type of the argument x is an ℓ-protected sum type $(s_1 + s_2)$, denoted $T_\ell(s_1 + s_2)$. A value of this type is of the form $(\eta_\ell\ (\mathsf{inj}_i\ e_i))$, $i \in \{1, 2\}$, where e_i is an expression of type s_i, inj_i is a case constructor, and η_ℓ denotes some ℓ-protection mechanism (which can be undone with bind). The function f undoes the protection on x and returns it. The function g also undoes the protection on x, but returns only its case constructor. 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.

Traditionally, security experts have dismissed this notion of "safety" as unsound, since the attacker might be able to amplify the one-bit leak of information in g to leak all information on x, thereby making it as dangerous as f. However, for non-malicious code, such attacks are often complex and seem rare in practice [18]. Indeed, in the past few years several static analyses for security have focused on restricting effects due to data flow, while ignoring other effects [5, 7–13, 19, 25–28, 30, 32, 33, 35]. 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. This is the stance we take in this paper.

Previous work on downgrading and robustness [21, 34] is based on 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 p 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 p may still be robust, in the sense that the attacker cannot exploit the information released by p 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. As usual, robustness may still require that such 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 sum types. For instance, consider the following typing rule in DCC:

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

The variable x binds the result of e upon undoing 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 ℓ or higher. This is ensured by the side condition, which restricts t to be only of certain forms. (We will review the formal definition of this condition later.) In particular, t cannot be a sum 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, respectively, types $(s_1 + s_2)$ and $(\mathsf{unit} + \mathsf{unit})$.

In contrast, we study the following typing rule in DCC^d, a variant of DCC:

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

The type constructor $\overline{T}_\ell$ provides weaker protection than T_ℓ; 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 sum 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^ℓ is given to a value of type s that requires weak protection by level ℓ. 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 the argument) but f is not. We show that if a program is typable in DCC, then it remains typable in DCC^d by weakening types. Furthermore, we formalize the precise guarantee enforced by DCC^d. This guarantee is related to Volpano's definition of *weak security* as a safety property [31], and it eliminates (at least) Denning and Denning's *explicit flow* attacks [14]. For non-malicious code, *i.e.*, code that the attacker cannot fully control, such attacks are far more dangerous than implicit flow attacks [23]; thus DCC^d's guarantee is important for security of such code, at least from a practical perspective.

While the typing rules of 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 dynamic weaken primitive that allows values of type T_ℓ to be cast as values of type $\overline{T}_\ell$. This weakening may invalidate strong protection guarantees at levels ℓ and lower. However, weak protection guarantees should still hold at these levels, and strong protection guarantees should hold at all other levels. We show how these guarantees can be enforced by recycling DCC's types to carry *blames* for weakening. Specifically, we include the rule

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

  where β is some isomorphism from the lattice of levels to some lattice of blames. The behavior of the resulting system, DCC^{dc}, rests on the definition of β.
  - If β preserves joins and meets, then a program's type carries a blame $\beta(\ell)$ such that ℓ *upper-bounds* the levels of weakening on which its results may depend; thus, strong protection guarantees continue to hold at all levels not ℓ or lower.
  - If β exchanges joins and meets, then a program's type carries a blame $\beta(\ell)$ such that ℓ *lower-bounds* the levels of weakening on which its results may depend; thus, weak protection guarantees are robust against all levels not ℓ or higher.
- Conversely, we show how a DCC^d-style analysis can make DCC's dependency analysis more precise. Consider the following functions, which are clearly secure yet rejected by DCC because sum types are never considered protected:

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

  To typecheck such functions, we observe 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 sum 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 can 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 more liberal than DCC; in particular, it admits some new type-preserving optimizations.

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, enhancing soundness of the latter and completeness of the former. 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 focused on explicit information flow. We argue that this system provides the foundations for several recent static analyses for security that do not restrict implicit information flow (Section 3).
- We study a language primitive weaken that switches from DCC-style protection to DCC^d-style protection of programs at run time. 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 enhance the precision of DCC's typing rules. This technique (once again) relies on deconstructing information flow into explicit and implicit information flow (Section 5).

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

2 Background on DCC

Recall that the computational lambda calculus [20] extends the simply typed lambda calculus with a type constructor that is interpreted as a monad. The monad is used to systematically control effects in the language. DCC [2] carries this idea further by distinguishing computations at various "levels", and controlling effects across levels. Specifically, DCC includes a monadic type constructor for each level in a lattice, and has a special typing rule that restricts how computations at various levels may be composed based on the lattice. Let ℓ denote levels in such a lattice with ordering $\sqsubseteq$, join $\sqcup$, meet $\sqcap$, bottom $\bot$, and top $\top$. We focus on the following syntax for types and terms in DCC. (For brevity, we omit any discussion of pointed types and recursive programs; see Section 6 for further comments.)

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

Types include unit, product, sum, and function types, as well as types $T_\ell(s)$ for each level ℓ in the lattice. Terms include the introduction and elimination forms for these types; the introduction forms are considered values. In particular, $(\eta_\ell\ e)$ has type $T_\ell(s)$ whenever e has type s, and $(\mathsf{bind}\ x = (\eta_\ell\ e)\ \mathsf{in}\ e')$ reduces to $e'[e/x]$.

In practice, η_ℓ may represent any mechanism that provides "protection" at level ℓ, 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 ℓ. 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 ℓ", meaning that terms of type t cannot leak any information at level ℓ—in other words, terms of type t are indistinguishable to any level ℓ' that is not at least ℓ in the lattice. The following rules define this predicate: $\ell \preceq \mathsf{unit}$; $\ell \preceq (s \to t)$ iff $\ell \preceq t$; $\ell \preceq (s \times t)$ iff $\ell \preceq s$ and $\ell \preceq t$; and $\ell \preceq T_{\ell'}(s)$ iff $\ell \sqsubseteq \ell'$ or $\ell \preceq s$.

Significantly, this definition does not consider sum types to be protected. The broad reason is that any information in terms is ultimately conveyed by case constructors. (The other constructors—unit, tupling, function abstraction, and ℓ-protection—cannot convey any information since they are completely determined by the associated types.) For instance, a boolean may be encoded as either $(\mathsf{inj}_1\ ())$ or $(\mathsf{inj}_2\ ())$, thereby conveying one bit of information; so the sum type $(\mathsf{unit} + \mathsf{unit})$ can serve as an encoding of the datatype `boolean`. In general, complex datatypes can be encoded using sum types, and the only way of distinguishing terms of such types is by analyzing the case constructors used in those terms. Thus, it makes sense to require explicit protection on any term of a sum type. (However, we will show in Section 5 that this restriction can be relaxed.)

The typing rules for DCC derive judgments of the form $\Gamma; \Pi \vdash e : t$, where Γ contains type hypotheses for free variables and Π is a *protection context* [29], which indicates the maximum level of protection promised by the context. If e is closed, Γ is empty and Π is $\bot$, and we use the simpler notation $\vdash e : t$ for the typing judgment. In addition to standard rules for the simply typed lambda calculus with sum and product types (see the appendix), we have:

$$(\mathsf{T\text{-}ret}) \qquad \frac{\Gamma; \Pi \sqcup \ell \vdash e : s}{\Gamma; \Pi \vdash (\eta_\ell\ e) : T_\ell(s)}$$

$$(\mathsf{T\text{-}bind}) \qquad \frac{\Gamma; \Pi \vdash e : T_\ell(s) \qquad \Gamma, x : s; \Pi \vdash e' : t \qquad \ell \preceq T_\Pi(t)}{\Gamma; \Pi \vdash \mathsf{bind}\ x = e\ \mathsf{in}\ e' : t}$$

$(\mathsf{T\text{-}ret})$ states that $(\eta_\ell\ e)$ has type $T_\ell(s)$ whenever e has type s, assuming ℓ-protection by the context (as promised by joining ℓ with the protection context). $(\mathsf{T\text{-}bind})$ states that $(\mathsf{bind}\ x = e\ \mathsf{in}\ e')$ has type t whenever e has a type of the form $T_\ell(s)$, e' has type t assuming that x has type s, and the type $T_\Pi(t)$ is protected at ℓ. The latter condition means that either t is protected at ℓ, or Π is at least ℓ; this ensures that the result of e' cannot leak any information at ℓ, including any information on x, which is bound to the result of e upon undoing its ℓ-protection at run time.

The key property of this type system—ensuring a form of parametricity [22, 29] or noninterference [2, 16]—can be formalized using a type-directed indistinguishability relation over terms, $e \sim_\ell e' : s$, meaning that terms e and e' of type s are indistinguishable to level ℓ. In addition to standard rules for logical equivalence (see the appendix),

we have that $(\eta_{\ell'}\ e)$ and $(\eta_{\ell'}\ e')$ are indistinguishable to ℓ unless ℓ is at least ℓ'. In other words, we have that the encryptions of e and e' with a secret key at level ℓ' are indistinguishable to an observer at level ℓ as long as ℓ does not know any secrets at ℓ'.

For example, let $e_1 = (\eta_\ell\ (\mathsf{inj}_1\ ()))$ and $e_2 = (\eta_\ell\ (\mathsf{inj}_2\ ()))$, and suppose that $\ell \not\sqsubseteq \ell'$. Then $e_1 \sim_{\ell'} e_2 : T_\ell(\mathsf{unit} + \mathsf{unit})$. Now recall the functions f and g defined in Section 1 (and assume that $s_1 = s_2 = \mathsf{unit}$ for simplicity). Then $f \not\sim_{\ell'} f$, since $(f\ e_1)$ reduces to $(\mathsf{inj}_1\ ())$, $(f\ e_2)$ reduces to $(\mathsf{inj}_2\ ())$, and $(\mathsf{inj}_1\ ()) \not\sim_{\ell'} (\mathsf{inj}_2\ ())$. Similarly, we can show that $g \not\sim_{\ell'} g$. Fortunately, neither function is typable in DCC. Next consider $f' = \lambda x.\ (\eta_\ell\ (f\ x))$ and $g' = \lambda x.\ (\eta_\ell\ (g\ x))$. Then we can show that $f' \sim_{\ell'} f' : T_\ell(\mathsf{unit}+\mathsf{unit}) \to T_\ell(\mathsf{unit}+\mathsf{unit})$ and $g' \sim_{\ell'} g' : T_\ell(\mathsf{unit}+\mathsf{unit}) \to T_\ell(\mathsf{unit}+\mathsf{unit})$, and both functions are typable in DCC. Indeed, the type system guarantees that whenever a typed function is applied to ℓ-protected inputs, it always produces outputs that are indistinguishable to levels that are not at least ℓ.

Theorem 1 (DCC soundness, *cf.* [29]). *If* $\vdash e : T_\ell(s) \to t$, $\vdash e_1 : s$, *and* $\vdash e_2 : s$, *then for any* ℓ' *such that* $\ell \not\sqsubseteq \ell'$, $(e\ (\eta_\ell\ e_1)) \sim_{\ell'} (e\ (\eta_\ell\ e_2)) : t$.

3 Explicit flows and DCC^d

While DCC can adequately encode various analyses, the underlying notion of dependency can be overly sensitive in certain settings. In this section, we 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 restricts only *explicit* flows of information. We 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 [14]: "*. . . an explicit flow [of some information x] occurs whenever the operations generating it are independent of the value of x.*" Unfortunately, this definition has been largely ignored in the literature. The only related work seems to be Volpano's [31], which defines *weak security* as a trace-based (safety) property: a program is weakly secure if its traces induce secure "branch-free" programs. We observe that weak security implies the absence of explicit flow attacks, since information flows in a branch-free program cannot be generated by operations that depend on specific values. (It seems that this connection between Volpano's and Denning and Denning's definitions has not been articulated previously.)

These definitions deserve more attention, since they suggest exactly why explicit flow attacks are so interesting in practice. Explicit flow vulnerabilities are attractive to attackers, since they can be exploited parametrically. Conversely, such vulnerabilities often point to logical errors rather than implementation "artifacts", since the information-flow channels are abstract. Finally, various 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 other "definite vulnerabilities", but also liberates dynamic checks to serve as mechanisms for plugging those vulnerabilities.

This may explain why several recent analyses for security have by design ignored implicit flow attacks and focused on eliminating explicit flow attacks [5, 7–13, 19, 25–28, 30, 32, 33, 35]. Some of these analyses aim to verify the security of web applications [13, 26, 28, 30]. Many attacks in this context are ultimately due to code injection, and a common defense against such attacks is to sanitize values that may flow from inputs to outputs. The 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, 8, 9], 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.

3.2 DCC^d

Our system, DCC^d, is a simple variant of DCC where the type constructors T_ℓ are replaced by $\overline{T}_\ell$, and the protection mechanisms η_ℓ 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^ℓ, called an *open type*. Intuitively, the type s^ℓ is given to terms of type s that need to be (weakly) protected at level ℓ. Open types do not have any special introduction or elimination forms. Instead they *qualify* existing types [15], according to the following equations.

- $(s^\ell)^{\ell'} = s^{\ell \sqcup \ell'}$ and $s = s^\bot$ (protection requirements can be joined with $\sqcup$, and any type can be viewed as an open type with no protection requirement);
- $\mathsf{unit}^\ell = \mathsf{unit}$, $(s \to t)^\ell = s \to t^\ell$, $(s \times t)^\ell = (s^\ell \times t^\ell)$, and $\overline{T}_{\ell'}(s)^\ell = \overline{T}_{\ell'}(s^\ell)$ (protection requirements are redundant for the unit type, and can be structurally propagated for other non-sum types);
- $\overline{T}_{\ell'}(s)^\ell = \overline{T}_{\ell'}(s)$ iff $\ell \sqsubseteq \ell'$ (protection requirements can be dropped if there is adequate protection).

Note that there is no equation for sum types. In particular, it would be unsafe to equate the open sum type $(s + t)^\ell$ with the sum type $(s^\ell + t^\ell)$, for reasons similar to those discussed in Section 2. It suffices to see that such an equation would imply $(\mathsf{unit} + \mathsf{unit})^\ell = (\mathsf{unit}^\ell + \mathsf{unit}^\ell) = (\mathsf{unit} + \mathsf{unit})$. But recall that the type $(\mathsf{unit} + \mathsf{unit})$ can serve as an encoding of `boolean`; so the equation in question 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. Furthermore, 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) sum types. For the latter terms, we assume that they always have open sum types.

Our enforcement strategy with open types is rather simple. Upon undoing protection of a term of type $\overline{T}_\ell(s)$, we give it a type s^ℓ. We then demand that such a term be

protected back with a level ℓ' that is at least ℓ. The resulting term has type $\overline{T}_{\ell'}(s^\ell)$, which can be equated to $\overline{T}_{\ell'}(s)$, thereby dropping the protection requirement. To enforce this strategy, we define (as in DCC) a predicate $\ell \leq t$, read as "t is weakly protected at ℓ", with the following rules: $\ell \leq \mathsf{unit}$; $\ell \leq (s \to t)$ iff $\ell \leq t$; $\ell \leq (s \times t)$ iff $\ell \leq s$ and $\ell \leq t$; $\ell \leq (s + t)$ iff $\ell \leq s$ and $\ell \leq t$; and $\ell \leq \overline{T}_{\ell'}(s)$ iff $\ell \sqsubseteq \ell'$ or $\ell \leq s$.

Note that there is no rule for open types, since such types are not protected by definition. On the other hand, we include a rule for sum types. Such a rule is sound in this context because we are only interested in tracking data dependencies and not control dependencies. Indeed, terms of type $(s+t)$—which evaluate to values $(\mathsf{inj}_i\ e)$—cannot leak any data not already leaked by e (which has type either s or t). In particular, the constructors inj_i cannot leak any data, unless the values $(\mathsf{inj}_i\ e)$ already require protection and thus have a non-trivial open type (where the qualifier is not $\bot$)—which is impossible since by the equations above, $(s + t)$ cannot be equal to such a type.

Following DCC, the typing rules for DCC^d derive judgments of the form $\Gamma; \overline{\Pi} \vdash e : t$, where Γ contains type hypotheses for free variables and $\overline{\Pi}$ is a (weak) protection context. We show only the interesting rules. (The remaining rules are in the appendix.)

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

$$(\mathsf{T^D\text{-}ret}) \qquad \frac{\Gamma; \overline{\Pi} \sqcup \ell \vdash e : s}{\Gamma; \overline{\Pi} \vdash (\overline{\eta}_\ell\ e) : \overline{T}_\ell(s)}$$

$$(\mathsf{T^D\text{-}bind}) \quad \frac{\Gamma; \overline{\Pi} \vdash e : \overline{T}_\ell(s) \qquad \Gamma, x : s^\ell; \overline{\Pi} \vdash e' : t \qquad \ell \leq \overline{T}_{\overline{\Pi}}(t)}{\Gamma; \overline{\Pi} \vdash \mathsf{bind}\ x = e\ \mathsf{in}\ e' : t}$$

$(\mathsf{T^D\text{-}case})$ assumes that the case construction $(\mathsf{inj}_i\ e)$ has an open sum type, and propagates its protection requirement to the variable x bound to e at run time. This allows sensitive data to be safely destructed, without losing track of its protection requirements. All other rules are syntax-directed (thanks to normalization of types as mentioned above) and are analogous to those in DCC. In particular, $(\mathsf{T^D\text{-}ret})$ states that $(\overline{\eta}_\ell\ e)$ has type $\overline{T}_\ell(s)$ whenever e has type s, assuming (weak) ℓ-protection by the context. $(\mathsf{T^D\text{-}bind})$ states that $(\mathsf{bind}\ x = e\ \mathsf{in}\ e')$ has type t only if e has a type of the form $\overline{T}_\ell(s)$, e' has type t assuming that x has open type s^ℓ, and the type $\overline{T}_{\overline{\Pi}}(t)$ is (weakly) protected at ℓ. The latter condition means that either t is protected at ℓ, or $\overline{\Pi}$ is at least ℓ. This ensures that the result of e' cannot leak any data at ℓ, including any data in x, which is bound to the result of e upon undoing its ℓ-protection at run time.

We formalize the key property of this type system using a type-directed *safety* relation over terms, $e \rhd_\ell : s$, meaning that term e of type s is safe at level ℓ. Our safety relation relies on a semantics with "taint propagation". Thus, we extend the internal syntax with terms of the form e^ℓ, meaning e tainted with ℓ—intuitively, e^ℓ is similar to $(\mathsf{bind}\ x = (\overline{\eta}_\ell\ e)\ \mathsf{in}\ x)$ for fresh x. We define equations over tainted terms, closely following the equations over open types. Thus we have: $(e^\ell)^{\ell'} = e^{\ell \sqcup \ell'}$; $e = e^\bot$; $()^\ell = ()$; $(\lambda x.\ e)^\ell = \lambda x.\ e^\ell$; $\langle e_1, e_2 \rangle^\ell = \langle e_1^\ell, e_2^\ell \rangle$; $(\overline{\eta}_{\ell'}\ e)^\ell = (\overline{\eta}_{\ell'}\ e^\ell)$; and $(\overline{\eta}_{\ell'}\ e)^\ell = (\overline{\eta}_{\ell'}\ e)$ iff $\ell \sqsubseteq \ell'$. As usual, these equations let us normalize terms so that only terms of sum types carry taints. Finally, we extend the local reduction rules for bind and case as follows: $(\mathsf{bind}\ x = (\overline{\eta}_\ell\ e)\ \mathsf{in}\ e')$ reduces to $e'[e^\ell/x]$, and

$(\mathsf{case}\ (\mathsf{inj}_i\ e)^\ell\ \mathsf{of}\ \mathsf{inj}_1(x).\ e_1 \parallel \mathsf{inj}_2(x).\ e_2)$ reduces to $e_i[e^\ell/x]$. Thus, we taint a term upon undoing its protection, and propagate the taint on a term to its subterm upon pattern matching. Note that such taint propagation ignores implicit flows. We use this semantics in the derivation rules of our safety relation, as follows.

- $e \rhd_\ell : s$ iff e reduces to v and $v \rhd_\ell : s$
- $() \rhd_\ell : \mathsf{unit}$
- $v \rhd_\ell : (s \to t)$ iff for all e, if $e \rhd_\ell : s$ then $(v\ e) \rhd_\ell : t$
- $\langle e_1, e_2 \rangle \rhd_\ell : (s_1 \times s_2)$ iff $e_1 \rhd_\ell : s_1$ and $e_2 \rhd_\ell : s_2$
- $(\mathsf{inj}_i\ e_i) \rhd_\ell : (s_1 + s_2)$ iff $e_i \rhd_\ell : s_i$
- $(\overline{\eta}_{\ell'}\ e) \rhd_\ell : \overline{T}_{\ell'}(s)$ iff $\ell' \not\sqsubseteq \ell$ or $e \rhd_\ell : s$

Thus, safety is analogous to indistinguishability as defined in Section 2, except that we are concerned with properties of a single term rather than a pair of terms. As expected, tainted terms are unsafe, and $(\overline{\eta}_{\ell'}\ e)$ is safe at ℓ unless ℓ is at least ℓ'.

For example, let $e = (\overline{\eta}_\ell\ (\mathsf{inj}_i\ ()))$ for some $i \in \{1, 2\}$ and suppose that $\ell \not\sqsubseteq \ell'$. Then $e \rhd_{\ell'} : \overline{T}_\ell(\mathsf{unit} + \mathsf{unit})$. Now recall the functions f and g defined in Section 1 (and assume that $s_1 = s_2 = \mathsf{unit}$ for simplicity). Clearly $f \not\rhd_{\ell'}$, since $(f\ e)$ reduces to $(\mathsf{inj}_i\ ())^\ell$ and $(\mathsf{inj}_i\ ()) \not\rhd_{\ell'}$. Fortunately, f is not typable in DCC^d. In contrast, we can show that $g \rhd_{\ell'} : \overline{T}_\ell(\mathsf{unit} + \mathsf{unit}) \to (\mathsf{unit} + \mathsf{unit})$, and g is typable in DCC^d. Next consider $f' = \lambda x.\ (\overline{\eta}_\ell\ (f\ x))$. It is easy to check that $f' \rhd_{\ell'} : \overline{T}_\ell(\mathsf{unit} + \mathsf{unit}) \to \overline{T}_\ell(\mathsf{unit} + \mathsf{unit})$, and f' is typable in DCC^d. Indeed, the type system guarantees that whenever a typed function is applied to (weakly) ℓ-protected inputs, it always produces outputs that are safe at levels that are not at least ℓ.

Theorem 2 (DCC^d soundness). *If* $\vdash e : \overline{T}_\ell(s) \to t$ *and* $\vdash e' : s$*, then for any* ℓ' *such that* $\ell \not\sqsubseteq \ell'$*,* $(e\ (\overline{\eta}_\ell\ e')) \rhd_{\ell'} : t$*.*

Furthermore, we show that DCC^d's type system is at least as liberal than DCC's, by defining an appropriate encoding between the two systems. (In fact, it is strictly more liberal by the example above.)

Theorem 3 (DCC to DCC^d). *Let* $[\![\cdot]\!]$ *translate terms and types by replacing* $(\eta_\ell\ \cdot)$ *with* $(\overline{\eta}_\ell\ \cdot)$*, and* $T_\ell(\cdot)$ *with* $\overline{T}_\ell(\cdot)$*. If* $\vdash e : s$ *in DCC then* $\vdash [\![e]\!] : [\![s]\!]$ *in DCC^d.*

3.3 Remarks

Before we move on, let us try to carefully understand the guarantee provided by DCC^d.

DCC^d's semantics, based on taint propagation, is closely related to Volpano's execution monitor for weak security [31]. In fact, results of evaluation in DCC^d can be interpreted as branch-free DCC programs "induced by traces", and typing in DCC^d guarantees security of such programs in DCC.

Theorem 4 (DCC^d soundness, à la Volpano [31]). *Let* $\{\!\{\cdot\}\!\}$ *translate terms and types by replacing* $(\overline{\eta}_\ell\ \cdot)$ *with* $(\eta_\ell\ \cdot)$*,* $\overline{T}_\ell(\cdot)$ *with* $T_\ell(\cdot)$*, and* $(\cdot^\ell)$ *with* $(\mathsf{bind}\ x = (\eta_\ell\ \cdot)\ \mathsf{in}\ x)$ *for fresh* x*. If* $\vdash e : \overline{T}_\ell(s) \to t$*,* $\vdash e' : s$*, and* $(e\ (\overline{\eta}_\ell\ e'))$ *evaluates to* v *in DCC^d, and if no protection type occurs negatively in* t*, then* $\vdash \{\!\{v\}\!\} : \{\!\{t\}\!\}$ *in DCC.*

For example, consider the following function of type $\overline{T}_\ell(\mathsf{unit} + s) \to (\mathsf{unit} + \overline{T}_\ell(s))$:

$$k = \lambda x.\ \mathsf{bind}\ y = x\ \mathsf{in}\ \mathsf{case}\ y\ \mathsf{of}\ \mathsf{inj}_1(z).\ (\mathsf{inj}_1\ ())\ \|\ \mathsf{inj}_2(z).\ (\mathsf{inj}_2\ (\overline{\eta}_\ell\ z))$$

Let e be any term of type s; we have that $(k\ (\overline{\eta}_\ell\ (\mathsf{inj}_2\ e)))$ reduces to $(\mathsf{inj}_2\ (\overline{\eta}_\ell\ e^\ell))$, which translates via $\{\!\{\cdot\}\!\}$ to $(\mathsf{inj}_2\ (\eta_\ell\ \mathsf{bind}\ w = (\eta_\ell\ e)\ \mathsf{in}\ w))$. The latter is a branch-free, typed, DCC program. In fact, by the theorem above, all branch-free programs induced by traces of k are typed, and thus k is weakly secure. In contrast, if the protection $(\overline{\eta}_\ell\ \cdot)$ in the body of k is dropped, the induced branch-free program does not remain typable.

Furthermore, DCC's type system eliminates explicit flow attacks as characterized by Denning and Denning [14], since we have already argued that weak security implies the absence of such attacks. Note that an explicit flow attack can be camouflaged as an implicit flow attack by "deep copying", *i.e.*, by destructing a sensitive term all the way down with elimination forms and constructing it back from scratch with introduction forms. Formally, let erase be a function on types that erases the label qualifiers in open types. Thus, for any s, we have $\ell \leq \mathsf{erase}(s)$ for all ℓ; in other words, the side condition in (T^{D}-bind) is redundant for erased types. Now we can define a family of functions $\mathit{leak}_\ell(t) : \overline{T}_\ell(t) \to \mathsf{erase}(t)$ that behave just like $\lambda x : \overline{T}_\ell(t).\ \mathsf{bind}\ y = x\ \mathsf{in}\ y$, such that the former are typable in DCC^d, but the latter are not (see the appendix). Thus, in the limit we may be assured nothing even if DCC^d deems our program "secure"—while DCC^d guarantees that all explicit leaks are eliminated, these leaks may remain hidden in the guise of implicit leaks (which remain unrestricted). However, we argue that DCC^d still provides "pretty good protection", at least for code that the attacker cannot fully control. Indeed, for such non-malicious code, we may assume that the programmer does not try to intentionally circumvent our analysis. Under this assumption, prioritizing explicit flows over implicit flows is arguably reasonable, for several reasons:

- No sane programmer would copy all bits of some value indirectly, one at a time, instead of copying the value directly.
- As argued in [23], implicit leaks are largely harmless for non-malicious code, since such leaks cannot be exploited efficiently by the attacker.
- As shown in [18], checking for implicit flows can be costly to the programmer—typically lots of false alarms arise in systems that check for implicit flows.

4 Dynamic weakening in DCC^{dc}

While DCC^d-style protection is sufficient in some settings, DCC still enjoys better theoretical foundations and promises many desirable properties that DCC^d cannot. In practice, we should be able to mix DCC^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 DCC^{dc}.

4.1 DCC^{dc}

DCC^{dc}'s syntax and typing rules are obtained by merging those of DCC and DCC^d. The merge is mostly straightforward; we make a few adjustments to encourage the two

subsystems to interact. (The full system is available for reference in the appendix.) First, we carry both kinds of protection contexts in typing judgments, and modify the DCC rule (T-ret) as follows.

$$\frac{\Gamma; \Pi \sqcup \ell; \overline{\Pi} \sqcup \ell \vdash e : s}{\Gamma; \Pi; \overline{\Pi} \vdash (\eta_\ell\ e) : T_\ell(s)}$$

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: $\ell \preceq \overline{T}_{\ell'}(s)$ if $\ell \preceq s$; $\ell \leq T_{\ell'}(s)$ if $\ell \leq \overline{T}_{\ell'}(s)$; and $T_{\ell'}(s)^\ell = T_{\ell'}(s^\ell)$ if $\overline{T}_{\ell'}(s)^\ell = \overline{T}_{\ell'}(s^\ell)$. These rules internalize the fact that DCC's protection types subsume DCC^d's protection types, as shown in Theorem 3. In particular, these rules admit functions such as $\lambda x.\ \mathsf{bind}\ y = x\ \mathsf{in}\ (\eta_\ell\ y)$ of type $\overline{T}_\ell(s) \to T_\ell(s)$, that can be used to *strengthen* protection on terms. Finally, we unify the rules for non-protection types; in particular we have:

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

The definitions of indistinguishability and safety are similarly extended, and we can show that the respective guarantees of DCC and DCC^d are preserved in DCC^{dc}.

Theorem 5 (DCC^{dc} soundness, preliminary). *Theorems 1 and 2 also hold in DCC^{dc}.*

4.2 A weakening primitive

Next we include a weaken primitive in DCC^{dc}, which acts as a further bridge between the two subsystems (going in the opposite direction as the strengthening functions above). Our intention is that such a primitive should allow terms of type $T_\ell(s)$ to be viewed as terms of type $\overline{T}_\ell(s)$, possibly with some caution.

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

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

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

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

The function m can be typed $T_{\ell'}(s+t) \to T_\ell(\mathsf{unit}+\mathsf{unit})$ in DCC as long as $\ell' \sqsubseteq \ell$, and 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 $T_{\ell'}(s+t) \to (\mathsf{unit} + \mathsf{unit})$ is able to use m in combination with h to leak that bit.

As this example suggests, using weaken at level ℓ in a program may invalidate DCC-style guarantees for all types protected by levels ℓ and lower. However, weaker DCC^d-style guarantees should still hold for such types (because there is no way to get around 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 ℓ, will never delegate the responsibility of protection to the weakened types. In summary, 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 ℓ we have a blame $\beta(\ell)$, where β is some lattice isomorphism. Then, instead of the naïve typing rule for weaken above, we include the following rule:

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

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 $\beta(\ell)$ for facilitating such a view. While this allows us to get away with weaker protection requirements on such terms, it also forces some caution: the blame must be carried around whenever a result depends on those terms. Fortunately, DCC's typing rules can enforce this for free.

A reassuring interpretation of blames may be obtained through the lens of the Curry-Howard isomorphism, following a recent reading of DCC as an authorization logic [1]. Specifically, we can interpret the blame $\beta(\ell)$ as a principal that controls protection requirements at level ℓ, and rewrite the type of $(\mathsf{weaken}\ e)$ as $\beta(\ell)\ \mathsf{says}\ (\overline{T}_\ell(s))$. 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 β preserves joins and meets. In this scenario, the type of a program must carry a blame $\beta(\ell)$ such that ℓ upper-bounds the levels of weakening on which its results may depend. (This is because DCC's rules guarantee that $\beta(\ell)$ will upper-bound the levels of weakening on which the results of the program may depend.) In other words, DCC-style protection guarantees must hold at all levels not ℓ or lower.

Formally, we define the blame $\mathcal{B}(t)$ 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^{dc} soundness: strong protection). *If* $\vdash e : T_\ell(s) \to t$, $\vdash e_1 : s$, $\vdash e_2 : s$, *and* ℓ *is any label such that* $\ell \not\sqsubseteq \beta^{-1}(\mathcal{B}(t))$, *then for any* ℓ' *such that* $\ell \not\sqsubseteq \ell'$, $(e\ (\eta_\ell\ e_1)) \sim_{\ell'} (e\ (\eta_\ell\ e_2)) : t$. *Moreover, Theorem 2 holds as is in this system.*

As a simple example, consider the following well-typed program of type $T_{\beta(\ell)}(\mathsf{unit} + \mathsf{unit})$ (where $i \in \{1, 2\}$):

$$\mathsf{bind}\ x = (\mathsf{weaken}\ (\eta_\ell\ (\mathsf{inj}_i\ ()))) \ \mathsf{in}\ (\eta_{\beta(\ell)}\ x)$$

We have $\mathcal{B}(T_{\beta(\ell)}(\mathsf{unit} + \mathsf{unit})) = \beta(\ell)$, so we can be sure that this program does not (and cannot be used to) weaken DCC-style protection guarantees at ℓ' unless $\ell' \sqsubseteq \ell$.

An equally interesting scenario is where we flip the ordering in the blame lattice, so that β exchanges joins and meets. In this scenario, the type of a program must carry a blame $\beta(\ell)$ such that ℓ lower-bounds the levels of weakening on which its results may depend. (Again, this is because DCC's rules guarantee that $\beta(\ell)$ will upper-bound the levels of weakening on which the results of the program may depend.) In other words, DCC^d-style protection guarantees must be robust against all levels not ℓ or higher.

Formally we prove the following theorem, where $\mathcal{B}(t)$ is defined as earlier.

Theorem 7 (DCC^{dc} soundness: weak protection). *Suppose that* $\vdash e : T_\ell(s)$, $\vdash e' : \overline{T}_\ell(s) \to t$, *and* ℓ *is any label such that* $\beta^{-1}(\mathcal{B}(t)) \not\sqsubseteq \ell$. *Then it is impossible to derive* $\vdash (e'\ (\mathsf{bind}\ x = (\mathsf{weaken}\ e)\ \mathsf{in}\ x)) : t$.

Continuing the previous example, we can be sure that DCC^d-style guarantees for the program are not influenced by weakening at level ℓ' unless $\ell \sqsubseteq \ell'$.

5 Precise dependency analysis in DCC^{cd}

Just as a DCC-style analysis can strengthen protection guarantees in a hybrid system, in turns out that a DCC^d-style analysis can improve the coverage of such guarantees. In this section, we 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 sum types in DCC. The resulting system, DCC^{cd}, becomes more liberal than DCC without compromising its guarantees. We discuss the benefits of such an enhancement towards the end of the section.

5.1 DCC^{cd}

DCC conservatively assumes that case constructors may always convey sensitive information; thus, it restricts both explicit and implicit flows in one shot by requiring that sum types can never be considered protected (see the discussion on DCC's protection rules in Section 2). Unfortunately this restriction causes several benign programs 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

sum 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 is already somewhat evident in DCC^d, where we weaken DCC's protection rules to allow sum types to be considered protected (see Section 3). But by itself this is unsound, given the dangerous nature of case constructors—it admits both explicit and implicit flows. Thus, we also require open types—types with qualifiers to precisely track data flow through programs—and we use these qualifiers to restrict explicit flows in DCC^d. In particular, the typing rule for case analysis needs to accommodate terms with qualified sum types, because the qualifiers can be eliminated on all types other than sum types—they "stick" to sum types exactly because of the dangerous nature of case constructors. While in DCC^d we choose to ignore implicit flows caused by such case analysis, in DCC^{cd} we do 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* Σ in typing judgments, which indicates the minimum level of protection required by the context. For closed terms, Σ is $\top$.

The developments of Section 4 are orthogonal to our present purposes, so we drop terms of the form $(\mathsf{weaken}\ e)$ and $(\overline{\eta}_\ell\ e)$ in the language; indeed, on the surface we do not care about DCC^d-style protection at all, although DCC^d's type system is an important component of the system internally. Accordingly, we also drop weak protection contexts. The remaining system mostly inherits from DCC^{dc}; we make a few adjustments, discussed below. (The full system is available for reference in the appendix.)

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

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

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

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

Curiously, this rule looks similar to $(\mathsf{T^D}\text{-bind})$ in DCC^d, although functionally it is intended to be closer to $(\mathsf{T}\text{-bind})$ in DCC. Like $(\mathsf{T}\text{-bind})$, $(\mathsf{T^{CD}}\text{-bind-new})$ applies to terms of type $T_\ell(s)$ instead of $\overline{T}_\ell(s)$. On the other hand, like $(\mathsf{T^D}\text{-bind})$, 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 meet ℓ with the open context, deferring their actual restriction till we encounter case analysis at level ℓ.

The new rule for case analysis is as follows.

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

As in $(\mathsf{T^D}\text{-case})$, this rule requires—without loss of generality—that e have an open sum type, with some protection requirement ℓ. In addition, it requires that the open

context Σ be no lower than ℓ—so that any implicit flows at ℓ that may occur through the case analysis are irrelevant to (*i.e.*, cannot compromise) Σ. With these rules, we show that DCC^{cd} provides the same guarantees as DCC, and is at least as liberal.

Theorem 8 (DCC^{cd} soundness and completeness). *If* $\vdash e : T_\ell(s) \to t$, $\vdash e_1 : s$, *and* $\vdash e_2 : s$, *then for any* ℓ' *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*^{cd}.

In fact, DCC^{cd} accepts more programs than DCC. For example, the following functions—rejected by DCC (see Section 1)—have type $T_\ell(s) \to (\mathsf{unit} + \mathsf{unit})$ in DCC^{cd}:

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

As a more interesting example, consider the function *switch* below—rejected by DCC—which has type $T_\ell(\texttt{boolean}) \to \texttt{boolean} \to \texttt{option}(T_\ell(\texttt{boolean}))$ in DCC^{cd}. (We use the encodings $\texttt{boolean} = (\mathsf{unit} + \mathsf{unit})$, $\texttt{option}(\alpha) = (\mathsf{unit} + \alpha)$, $\texttt{false} = (\mathsf{inj}_1\ ())$, $\texttt{true} = (\mathsf{inj}_2\ ())$, $(\texttt{if}\ e\ \texttt{then}\ e_2\ \texttt{else}\ e_1) = (\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inj}_1(\_).\ e_1 \parallel \mathsf{inj}_2(\_).\ e_2)$, $\texttt{none} = (\mathsf{inj}_1\ ())$, and $(\texttt{some}\ e) = (\mathsf{inj}_2\ e)$.

$$\begin{aligned} \textit{switch} &= \lambda x.\ \lambda b.\ \mathsf{bind}\ b' = x\ \mathsf{in}\ ((\textit{match}\ b)\ b') \\ \textit{match} &= \lambda b.\ \lambda b'.\ \texttt{if}\ b\ \texttt{then}\ \texttt{none}\ \texttt{else}\ (\texttt{some}\ (\eta_\ell\ (\textit{not}\ b'))) \\ \textit{not} &= \lambda b'.\ \texttt{if}\ b'\ \texttt{then}\ \texttt{false}\ \texttt{else}\ \texttt{true} \end{aligned}$$

In general, undoing protection of terms early in the control-flow graph seems to cause problems in DCC, but not in DCC^{cd}.

5.2 Remarks

One may, of course, wonder whether our enhancement of DCC's type system is at all necessary. Indeed, DCC is designed to be a target language in which (type-based) program analyses can be encoded to prove their soundness: typing derivations in the source language are translated to typing derivations in DCC, and the soundness of the latter is used to reason about the soundness of the former. In this sense, in fact it is possible to encode DCC^{cd} in DCC: we compile DCC^{cd} programs to the SLam calculus [17] by erasing binds, and then use the well-known encoding of SLam in DCC [2]. Thus, DCC's status as a core calculus of dependency is not challenged. However, as in most such encodings, the translated DCC programs are not syntactically equivalent to the source programs. In particular, binds may be pushed inwards and duplicated across branches. Reasoning about the soundness of this translation requires exactly those observations that underlie the design of DCC^{cd}. Furthermore, the translated programs are inefficient. Indeed, in implementations of DCC in the polymorphic lambda calculus [29], binds are implemented as applications of secret keys (decryptions) to protection abstractions (encryptions)—and it makes sense to pull such applications as outwards as possible for efficiency. For source languages with DCC-like primitives, it is reasonable to expect that programs will be already be optimized; and we have shown that DCC-like typing rules do not preserve typability for such optimizations. In summary, we believe

that deconstructing information-flow analysis into explicit-flow and implicit-flow analysis, as in DCC^{cd}, provides a better guideline for designing type systems for DCC-like source languages, than placing an overall restriction on sum types, as in DCC. Other enhancements along such lines have been suggested previously [29].

6 Discussion

For brevity, in this paper we have omitted any discussion of pointed types and recursive programs, although they do appear in DCC [2]. However, we have checked that including these elements does not cause any problems in our results—which is hardly surprising since nontermination does not play an interesting role for weak security.

We have tried to remain close in spirit to Volpano's definition of weak security and Denning and Denning's characterization of explicit flows in our formal definition of DCC^{d}. However, inherent differences in the underlying languages make it difficult to establish a formal correspondence.

There is a huge body of research on noninterference-based security for languages; see [24] for a survey. However, there seems to be a disconnect between this research and most security tools implemented in practice, which ignore implicit flows. Some interesting previous studies have tried to explain why, and under what circumstances, it may make sense to ignore implicit flows in practical security [18, 23]. Unfortunately, we do not know of any work on formalizing the resulting safety guarantees of such tools, although [31] provides some valuable insights and several security type systems for process calculi have been designed around similar ideas [3, 6, 8, 9].

The idea of mixing strong and weak dependency analysis in mutually benefitial ways appears to be new. Indeed, our results suggest some interesting ways in which noninterference-based security may be reconciled with trace-based security within the same system, enhancing soundness of the latter and completeness of the former. 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.

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'99: Principles of Programming Languages*, pages 147–160. ACM, 1999.
3. M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. In *POP'02: Principles of Programming Languages*, pages 33–44. ACM, 2002.
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. P. Broadwell, M. Harren, and N. Sastry. Scrash: a system for generating secure crash information. In *SSYM'03: USENIX Security Symposium*, pages 19–30. USENIX Association, 2003.
6. L. Cardelli, G. Ghelli, and A. Gordon. Secrecy and group creation. *Information and Computation*, 196(2):127–155, 2005.
7. M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *OSDI'06: Operating Systems Design and Implementation*, pages 147–160. USENIX, 2006.
8. A. Chaudhuri. Language-based security on Android. In *PLAS'09: Programming Languages and Analysis for Security*, pages 1–7. ACM, 2009.
9. 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.
10. K. Chen and D. Wagner. Large-scale analysis of format string vulnerabilities in debian linux. In *PLAS'07: Programming languages and analysis for security*, pages 75–84. ACM, 2007.
11. J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA'07: International Symposium on Software Testing and Analysis*, pages 196–206. ACM, 2007.
12. M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *SOSP'05: Symposium on Operating Systems Principles*, pages 133–147. ACM, 2005.
13. M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. *SIGARCH Comput. Archit. News*, 35(2):482–493, 2007.
14. D. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7), 1977.
15. J. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. *ACM SIGPLAN Notices*, 34(5):192–203, 1999.
16. J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and privacy*, volume 12, 1982.
17. N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *POPL'98: Principles of Programming Languages*, pages 365–377. ACM, 1998.
18. 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.
19. M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. In *OOPSLA'05: Object-oriented programming, systems, languages, and applications*, pages 365–383. ACM, 2005.
20. E. Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.
21. A. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *CSFW'04: IEEE Computer Security Foundations Workshop*, pages 172–186. IEEE, 2004.
22. J. Reynolds. Types, abstraction and parametric polymorphism. *Information processing*, 83(513-523):1, 1983.
23. A. Russo, A. Sabelfeld, and K. Li. Implicit flows in malicious and nonmalicious code. *Marktoberdorf Lecture Notes*, 2009. See `http://www.cse.chalmers.se/~andrei/mod09.pdf`.
24. A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.
25. U. Shankar, T. Jaeger, and R. Sailer. Toward automated information-flow integrity verification for security-critical applications. In *NDSS'06: Network and Distributed System Security Symposium*. ISOC, 2006.
26. U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting format string vulnerabilities with type qaualifiers. In *USENIX Security Symposium*, page 16. USENIX Association, 2001.

27. G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS'04: Architectural Support for Programming Languages and Operating Systems*, pages 85–96. ACM, 2004.
28. O. Tripp, S. Fink, and O. Weisman. TAJ: effective taint analysis of web applications. In *PLDI'09: Programming Languages Design and Implementation*, pages 87–97. ACM, 2009.
29. S. Tse and S. Zdancewic. Translating dependency into parametricity. *ACM SIGPLAN Notices*, 39(9):115–125, 2004.
30. P. Vogt, F. Nentwich, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS'07: Network and Distributed System Security Symposium*. ISOC, 2007.
31. D. M. Volpano. Safety versus secrecy. In *SAS'99: Static Analysis Symposium*, pages 303–311. Springer-Verlag, 1999.
32. Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3):16, 2007.
33. H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS'07: Computer and Communications Security*, pages 116–127. ACM, 2007.
34. S. Zdancewic and A. Myers. Robust declassification. In *CSFW'01: IEEE Computer Security Foundations Workshop*, pages 15–23. IEEE, 2001.
35. X. Zhang, A. Edwards, and T. Jaeger. Using cqual for static analysis of authorization hook placement. In *USENIX Security Symposium*, pages 33–48. USENIX Association, 2002.

Appendix

We include full definitions of various systems described in this paper. (See next page.)

Typing rules (DCC)

$$\text{(T-var)} \quad \Gamma, x : s, \Gamma'; \Pi \vdash x : s$$

$$\text{(T-unit)} \quad \Gamma; \Pi \vdash () : \mathsf{unit}$$

$$\text{(T-abs)} \quad \frac{\Gamma, x : s; \Pi \vdash e : t}{\Gamma; \Pi \vdash \lambda x.\, e : (s \to t)}$$

$$\text{(T-app)} \quad \frac{\Gamma; \Pi \vdash e : s \to t \qquad \Gamma; \Pi \vdash e' : s}{\Gamma; \Pi \vdash (e\ e') : t}$$

$$\text{(T-pair)} \quad \frac{\Gamma; \Pi \vdash e_1 : s_1 \qquad \Gamma; \Pi \vdash e_2 : s_2}{\Gamma; \Pi \vdash \langle e_1, e_2 \rangle : (s_1 \times s_2)}$$

$$\text{(T-proj)} \quad \frac{\Gamma; \Pi \vdash e : (s_1 \times s_2)}{\Gamma; \Pi \vdash (\mathsf{proj}_i\ e) : s_i}$$

$$\text{(T-inj)} \quad \frac{\Gamma; \Pi \vdash e : s_i}{\Gamma; \Pi \vdash (\mathsf{inj}_i\ e) : (s_1 + s_2)}$$

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

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

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

Indistinguishability relation (DCC)

$$\text{(I-unit)} \quad () \sim_\ell () : \mathsf{unit}$$

$$\text{(I-function)} \quad \frac{\forall e, e'.\ e \sim_\ell e' : s \ \Rightarrow\ (v\ e) \sim_\ell (v'\ e') : t}{v \sim_\ell v' : (s \to t)}$$

$$\text{(I-product)} \quad \frac{e_1 \sim_\ell e_1' : s_1 \qquad e_2 \sim_\ell e_2' : s_2}{\langle e_1, e_2 \rangle \sim_\ell \langle e_1', e_2' \rangle : (s_1 \times s_2)}$$

$$\text{(I-sum)} \quad \frac{e_i \sim_\ell e_i' : s_i}{(\mathsf{inj}_i\ e_i) \sim_\ell (\mathsf{inj}_i\ e_i') : (s_1 + s_2)}$$

$$\text{(I-monad-1)} \quad \frac{\ell' \not\sqsubseteq \ell}{(\eta_{\ell'}\ e) \sim_\ell (\eta_{\ell'}\ e') : T_{\ell'}(s)}$$

$$\text{(I-monad-2)} \quad \frac{e \sim_\ell e' : s}{(\eta_{\ell'}\ e) \sim_\ell (\eta_{\ell'}\ e') : T_{\ell'}(s)}$$

$$\text{(I-eval)} \quad \frac{e \longrightarrow^\star v \qquad e' \longrightarrow^\star v' \qquad v \sim_\ell v' : s}{e \sim_\ell e' : s}$$

Typing rules (DCC^d)

$$(\mathsf{T^D}\text{-var}) \quad \Gamma, x : s, \Gamma'; \overline{\Pi} \vdash x : s$$

$$(\mathsf{T^D}\text{-unit}) \quad \Gamma \vdash () : \mathsf{unit}$$

$$(\mathsf{T^D}\text{-abs}) \quad \frac{\Gamma, x : s; \overline{\Pi} \vdash e : t}{\Gamma; \overline{\Pi} \vdash \lambda x.\, e : (s \to t)}$$

$$(\mathsf{T^D}\text{-app}) \quad \frac{\Gamma; \overline{\Pi} \vdash e : s \to t \qquad \Gamma; \overline{\Pi} \vdash e' : s}{\Gamma; \overline{\Pi} \vdash (e\ e') : t}$$

$$(\mathsf{T^D}\text{-pair}) \quad \frac{\Gamma; \overline{\Pi} \vdash e_1 : s_1 \qquad \Gamma; \overline{\Pi} \vdash e_2 : s_2}{\Gamma; \overline{\Pi} \vdash \langle e_1, e_2 \rangle : (s_1 \times s_2)}$$

$$(\mathsf{T^D}\text{-proj}) \quad \frac{\Gamma; \overline{\Pi} \vdash e : (s_1 \times s_2)}{\Gamma; \overline{\Pi} \vdash (\mathsf{proj}_i\ e) : s_i)}$$

$$(\mathsf{T^D}\text{-inj}) \quad \frac{\Gamma; \overline{\Pi} \vdash e : s_i}{\Gamma; \overline{\Pi} \vdash (\mathsf{inj}_i\ e) : (s_1 + s_2)}$$

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

$$(\mathsf{T^D}\text{-ret}) \quad \frac{\Gamma; \overline{\Pi} \sqcup \ell \vdash e : s}{\Gamma; \overline{\Pi} \vdash (\overline{\eta}_\ell\ e) : \overline{T}_\ell(s)}$$

$$(\mathsf{T^D}\text{-bind}) \quad \frac{\Gamma; \overline{\Pi} \vdash e : \overline{T}_\ell(s) \qquad \Gamma, x : s^\ell; \overline{\Pi} \vdash e' : t \qquad \ell \leq \overline{T}_{\overline{\Pi}}(t)}{\Gamma; \overline{\Pi} \vdash \mathsf{bind}\ x = e\ \mathsf{in}\ e' : t}$$

Leaking explicit flows via implicit flows: DCC^d

$$\begin{aligned}
\mathit{leak}_\ell(\mathsf{unit}) &= \lambda x : \overline{T}_\ell(\mathsf{unit}).\ () \\
\mathit{leak}_\ell(s \times t) &= \lambda x : \overline{T}_\ell(s \times t).\ \mathsf{bind}\ y = x\ \mathsf{in} \\
&\qquad \langle \mathit{leak}_\ell(s)(\overline{\eta}_\ell\ (\mathsf{proj}_1\ y)), \mathit{leak}_\ell(t)(\overline{\eta}_\ell\ (\mathsf{proj}_2\ y)) \rangle \\
\mathit{leak}_\ell((s+t)^{\ell'}) &= \lambda x : \overline{T}_\ell((s+t)^{\ell'}).\ \mathsf{bind}\ y = x\ \mathsf{in} \\
&\qquad \mathsf{case}\ y\ \mathsf{of} \\
&\qquad\quad \mathsf{inj}_1(z_1).\ (\mathsf{inj}_1\ (\mathit{leak}_{\ell \sqcup \ell'}(s)(\overline{\eta}_{\ell \sqcup \ell'}\ z_1))) \\
&\qquad \parallel \mathsf{inj}_2(z_2).\ (\mathsf{inj}_2\ (\mathit{leak}_{\ell \sqcup \ell'}(t)(\overline{\eta}_{\ell \sqcup \ell'}\ z_2))) \\
\mathit{leak}_\ell(s \to t) &= \lambda x : \overline{T}_\ell(s \to t). \\
&\qquad \lambda z : s.\ \mathsf{bind}\ f = x\ \mathsf{in}\ \mathit{leak}_\ell(t)(\overline{\eta}_\ell\ (f\ z)) \\
\mathit{leak}_\ell(\overline{T}_{\ell'}(s)) &= \lambda x : \overline{T}_\ell(\overline{T}_{\ell'}(s)).\ \mathsf{bind}\ y = x\ \mathsf{in} \\
&\qquad (\overline{\eta}_{\ell'}\ (\mathit{leak}_{\ell'}(s) y))
\end{aligned}$$

Syntax: DCC^{dc}

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

Typing rules: DCC^{dc}

(T^{DC}-var) $\quad \Gamma, x : s, \Gamma'; \Pi; \overline{\Pi} \vdash x : s$

(T^{DC}-unit) $\quad \Gamma; \Pi; \overline{\Pi} \vdash () : \mathsf{unit}$

(T^{DC}-abs) $\quad \dfrac{\Gamma, x : s; \Pi; \overline{\Pi} \vdash e : t}{\Gamma; \Pi; \overline{\Pi} \vdash \lambda x.\ e : (s \to t)}$

(T^{DC}-app) $\quad \dfrac{\Gamma; \Pi; \overline{\Pi} \vdash e : s \to t \qquad \Gamma; \Pi; \overline{\Pi} \vdash e' : s}{\Gamma; \Pi; \overline{\Pi} \vdash (e\ e') : t}$

(T^{DC}-pair) $\quad \dfrac{\Gamma; \Pi; \overline{\Pi} \vdash e_1 : s_1 \qquad \Gamma; \Pi; \overline{\Pi} \vdash e_2 : s_2}{\Gamma; \Pi; \overline{\Pi} \vdash \langle e_1, e_2 \rangle : (s_1 \times s_2)}$

(T^{DC}-proj) $\quad \dfrac{\Gamma; \Pi; \overline{\Pi} \vdash e : (s_1 \times s_2)}{\Gamma; \Pi; \overline{\Pi} \vdash (\mathsf{proj}_i\ e) : s_i}$

(T^{DC}-inj) $\quad \dfrac{\Gamma; \Pi; \overline{\Pi} \vdash e : s_i}{\Gamma; \Pi; \overline{\Pi} \vdash (\mathsf{inj}_i\ e) : (s_1 + s_2)}$

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

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

(T^{DC}-ret-2) $\quad \dfrac{\Gamma; \Pi; \overline{\Pi} \sqcup \ell \vdash e : s}{\Gamma; \Pi; \overline{\Pi} \vdash (\overline{\eta}_\ell\ e) : \overline{T}_\ell(s)}$

(T^{DC}-bind-1) $\quad \dfrac{\Gamma; \Pi; \overline{\Pi} \vdash e : T_\ell(s) \qquad \Gamma, x : s; \Pi; \overline{\Pi} \vdash e' : t \qquad \ell \preceq T_\Pi(t)}{\Gamma; \Pi; \overline{\Pi} \vdash \mathsf{bind}\ x = e\ \mathsf{in}\ e' : t}$

(T^{DC}-bind-2) $\quad \dfrac{\Gamma; \Pi; \overline{\Pi} \vdash e : \overline{T}_\ell(s) \qquad \Gamma, x : s^\ell; \Pi; \overline{\Pi} \vdash e' : t \qquad \ell \leq T_\Pi(t)}{\Gamma; \Pi; \overline{\Pi} \vdash \mathsf{bind}\ x = e\ \mathsf{in}\ e' : t}$

(T^{DC}-weaken) $\quad \dfrac{\Gamma; \Pi; \overline{\Pi} \vdash e : T_\ell(s)}{\Gamma; \Pi; \overline{\Pi} \vdash (\mathsf{weaken}\ e) : T_{\beta(\ell)}(\overline{T}_\ell(s))}$

Protection rules: DCC^{dc}

(P-unit) $\quad \ell \preceq \mathsf{unit}$

(P-product) $\quad \ell \preceq s \ \wedge\ \ell \preceq t \ \Rightarrow\ \ell \preceq (s \times t)$

(P-function) $\quad \ell \preceq t \ \Rightarrow\ \ell \preceq (s \to t)$

(P-monad-1, 2) $\quad \ell \sqsubseteq \ell' \ \Rightarrow\ \ell \preceq T_{\ell'}(s) \quad , \quad \ell \preceq s \ \Rightarrow\ \ell \preceq T_{\ell'}(s)$

(P-effect) $\quad \ell \preceq s \ \Rightarrow\ \ell \preceq \overline{T}_{\ell'}(s)$

(P^{D}-unit) $\quad \ell \leq \mathsf{unit}$

(P^{D}-product) $\quad \ell \leq s \ \wedge\ \ell \leq t \ \Rightarrow\ \ell \leq (s \times t)$

(P^{D}-function) $\quad \ell \leq t \ \Rightarrow\ \ell \leq (s \to t)$

(P^{D}-effect) $\quad \ell \sqsubseteq \ell' \ \Rightarrow\ \ell \leq T_{\ell'}(s) \quad , \quad \ell \leq s \ \Rightarrow\ \ell \leq \overline{T}_{\ell'}(s)$

(P^{D}-sum) $\quad \ell \leq s \ \wedge\ \ell \leq t \ \Rightarrow\ \ell \leq (s + t)$

(P^{D}-monad) $\quad \ell \leq \overline{T}_{\ell'}(s) \ \Rightarrow\ \ell \leq T_{\ell'}(s)$

Open type equations: DCC^{dc}

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

Syntax: DCC^{cd}

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

Typing rules: DCC^{cd}

$$\begin{array}{ll}
(\mathsf{T}^{\mathsf{CD}}\text{-var}) & \Gamma, x : s, \Gamma'; \Pi; \Sigma \vdash x : s \\
(\mathsf{T}^{\mathsf{CD}}\text{-unit}) & \Gamma; \Pi; \Sigma \vdash () : \mathsf{unit} \\
(\mathsf{T}^{\mathsf{CD}}\text{-abs}) & \dfrac{\Gamma, x : s; \Pi; \Sigma \vdash e : t}{\Gamma; \Pi; \Sigma \vdash \lambda x.\ e : (s \to t)} \\
(\mathsf{T}^{\mathsf{CD}}\text{-app}) & \dfrac{\Gamma; \Pi; \Sigma \vdash e : s \to t \qquad \Gamma; \Pi; \Sigma \vdash e' : s}{\Gamma; \Pi; \Sigma \vdash (e\ e') : t} \\
(\mathsf{T}^{\mathsf{CD}}\text{-pair}) & \dfrac{\Gamma; \Pi; \Sigma \vdash e_1 : s_1 \qquad \Gamma; \Pi; \Sigma \vdash e_2 : s_2}{\Gamma; \Pi; \Sigma \vdash \langle e_1, e_2 \rangle : (s_1 \times s_2)} \\
(\mathsf{T}^{\mathsf{CD}}\text{-proj}) & \dfrac{\Gamma; \Pi; \Sigma \vdash e : (s_1 \times s_2)}{\Gamma; \Pi; \Sigma \vdash (\mathsf{proj}_i\ e) : s_i} \\
(\mathsf{T}^{\mathsf{CD}}\text{-inj}) & \dfrac{\Gamma; \Pi; \Sigma \vdash e : s_i}{\Gamma; \Pi; \Sigma \vdash (\mathsf{inj}_i\ e) : (s_1 + s_2)} \\
(\mathsf{T}^{\mathsf{CD}}\text{-case}) & \dfrac{\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 \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inj}_1(x).\ e_1 \parallel \mathsf{inj}_2(x).\ e_2 : s} \\
(\mathsf{T}^{\mathsf{CD}}\text{-ret}) & \dfrac{\Gamma; \Pi \sqcup \ell; \Sigma \vdash e : s}{\Gamma; \Pi; \Sigma \vdash (\eta_\ell\ e) : T_\ell(s)} \\
(\mathsf{T}^{\mathsf{CD}}\text{-bind-1}) & \dfrac{\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 \mathsf{bind}\ x = e\ \mathsf{in}\ e' : t} \\
(\mathsf{T}^{\mathsf{CD}}\text{-bind-2}) & \dfrac{\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 \mathsf{bind}\ x = e\ \mathsf{in}\ e' : t}
\end{array}$$

Protection rules and Open type equations: DCC^{cd}

Same as those for DCC^{dc}.