


A Dependent Dependency Calculus

Pritam Choudhury[1]✉, Harley Eades III[2], and Stephanie Weirich[3]

[1] University of Pennsylvania, Philadelphia, PA, USA, pritam@seas.upenn.edu
[2] Augusta University, Augusta, GA, USA
[3] University of Pennsylvania, Philadelphia, PA, USA

Abstract. Over twenty years ago, Abadi et al. established the Dependency Core Calculus (DCC) as a general purpose framework for analyzing dependency in typed programming languages. Since then, dependency analysis has shown many practical benefits to language design: its results can help users and compilers enforce security constraints, eliminate dead code, among other applications. In this work, we present a Dependent Dependency Calculus (DDC), which extends this general idea to the setting of a dependently-typed language. We use this calculus to track both run-time and compile-time irrelevance, enabling faster type-checking and program execution.

Keywords: Dependent Types · Information Flow · Irrelevance

1 Dependency Analysis

Consider this judgment from a type system that has been augmented with *dependency analysis*.

$$x :^{L} \mathbf{Int}, y :^{H} \mathbf{Bool}, z :^{M} \mathbf{Bool} \vdash \mathbf{if}\ z\ \mathbf{then}\ x\ \mathbf{else}\ 3\ :^{M}\ \mathbf{Int}$$

In this judgment, L, M and H stand for low, medium and high security levels respectively. The computed value of the expression is meant to be a medium-security result. The inputs, x, y and z have been marked with their respective security levels. This expression type-checks because it is permissible for medium-security results to *depend* on both low and medium-security inputs. Note that the high-security boolean variable y is not used in the expression. However, if we replace z with y in the conditional, then the type checker would reject that expression. Even though the high-security input would not be returned directly, the medium-security result would still depend on it.

Dependency analysis, as we see above, is an *expressive* addition to programming languages. Such analyses allow languages to protect sensitive information [30,16], support run-time code generation [33], slice programs while preserving behavior [34], etc. Several existing dependency analyses were unified by Abadi et al. [1] in their Dependency Core Calculus (DCC). This calculus has served as a foundation for static analysis of dependencies in programming languages.

I. Sergey (Ed.): ESOP 2022, LNCS 13240, pp. 403–434, 2022.
https://doi.org/10.1007/978-3-030-99336-8_15

What makes DCC powerful is the parameterization of the type system by a *generic* lattice of dependency levels. Dependency analysis, in essence, is about ensuring secure information flow—that information never flows from more secure to less secure levels. Denning [13] showed that a lattice model, where increasing order corresponds to higher security, can be used to enforce secure flow of information. DCC integrates this lattice model with the computational λ-calculus [22] by grading the monad operator of the latter with elements of the former. This integration enables DCC to analyze dependencies in its type system.

However, even though many typed languages have included dependency analysis in some form, this feature has seen relatively little attention in the context of *dependently-typed* languages. This is unfortunate because, as we show in this paper, dependency analysis can provide an elegant foundation for compile-time and run-time irrelevance, two important concerns in the design of dependently-typed languages. Compile-time irrelevance identifies sub-expressions that are not needed for type checking while run-time irrelevance identifies sub-expressions that do not affect the result of evaluation. By ignoring or erasing such sub-expressions, compilers for dependently-typed languages increase the expressiveness of the type system, improve on compilation time and produce more efficient executables.

Therefore, in this work, we augment a dependently-typed language with a *primitive* notion of dependency analysis and use it to track compile-time and run-time irrelevance. We call this language DDC, for Dependent Dependency Calculus, in homage to DCC. Although our dependency analyses are structured differently, we show that DDC can faithfully embed the terminating fragment of DCC and support its many well-known applications, in addition to our novel application of tracking compile-time and run-time irrelevance.

More specifically, our work makes the following contributions:

- We design a language SDC, for Simple Dependency Calculus, that can analyze dependencies in a simply-typed language. We show that SDC is no less expressive than the terminating fragment of DCC. The structure of dependency analysis in SDC enables a relatively straightforward syntactic proof of non-interference. (Section 3)
- We extend SDC to a dependent calculus, $\mathrm{DDC}^{\top}$. Using this calculus, we analyze run-time irrelevance and show the analysis is correct using a non-interference theorem. $\mathrm{DDC}^{\top}$ contains SDC as a sub-language. As such, it can be used to track other forms of dependencies as well. (Section 4)
- We generalize $\mathrm{DDC}^{\top}$ to DDC. Using this calculus, we analyze both run-time and compile-time irrelevance and show that the analyses are correct. To the best of our knowledge, DDC is the only system that can distinguish run-time and compile-time irrelevance as separate modalities, necessary for the proper treatment of projection from irrelevant Σ-types. (Section 5)
- We have used the Coq proof assistant to mechanically verify the most important and delicate part of our designs, the non-interference and type sound-

[4] `https://github.com/sweirich/graded-haskell`

ness theorems for DDC. This mechanization is available online[4] and as a self-contained artifact [11].

2 Irrelevance and Dependent Types

Run-time irrelevance (sometimes called *erasure*) and *compile-time irrelevance* are two forms of *dependency* analyses that arise in dependent type theories. Tracking these dependencies helps compilers produce faster executables and makes type checking more flexible [27,19,6,20,3,18,4,24,32,23].

2.1 Run-time irrelevance

Parts of a program that are not required during run time are said to be run-time irrelevant. Our goal is to identify such parts. Let's consider some examples. We shall mark variables and arguments with $\top$ if they can be erased prior to execution and leave them unmarked if they should be preserved.

For example, the polymorphic identity function can be marked as:

```
id : Π x:⊤Type. x -> x
id = λ⊤x. λy. y
```

The first parameter, x, of the identity function is only needed during type checking; it can be erased before execution. The second parameter, y, though, is required during runtime. When we apply this function to arguments, as in `(id Bool⊤ True)`, we can erase the first argument, `Bool`, but the second one, `True`, must be retained.

Indexed data structures provide another example of run-time irrelevance.

Consider the `Vec` datatype for length-indexed vectors, as it might look in a core language inspired by GHC [31,35]. The `Vec` datatype has two parameters, `n` and `a`, that also appear in the types of the data constructors `Nil` and `Cons`. These parameters are relevant to `Vec`, but irrelevant to the data constructors. (In the types of the constructors, the equality constraints `(n ∼ Zero)` and `(n ∼ Succ m)` force `n` to be equal to the length of the vector.)

```
Vec  : Nat -> Type -> Type
Nil  : Π n:⊤Nat. Π a:⊤Type. (n ∼ Zero) => Vec n a
Cons : Π n:⊤Nat. Π a:⊤Type. Π m:⊤Nat. (n ∼ Succ m) => a -> Vec m a
    -> Vec n a
```

Now consider a function `vmap` that maps a given function over a given vector. The length of the vector and the type arguments are not necessary for running `vmap`; they are all erasable. So we assign them $\top$.

```
vmap : Π n:⊤Nat.Π a b:⊤Type.  (a -> b) -> Vec n a -> Vec n b
vmap = λ⊤ n a b. λ f xs.
          case xs of
            Nil -> Nil
            Cons m⊤ x xs -> Cons m⊤ (f x) (vmap m⊤ a⊤ b⊤ f xs)
```

Note that the ⊤-marked variables `m`, `a` and `b` appear in the definition of `vmap`, but only in ⊤ contexts. By requiring that 'unmarked' terms *don't depend* on terms marked with ⊤, we can track run-time irrelevance and guarantee safe erasure. Observe that even though these arguments are marked with ⊤ to describe their use in the *definition* of `vmap`, this marking does not reflect their usage in the *type* of `vmap`. In particular, we are free to use these variables with `Vec` in a relevant manner.

2.2 Compile-time Irrelevance

Some type constructors may have arguments which can be ignored during type checking. Such arguments are said to be *compile-time irrelevant*. For example, suppose we have a constant function that ignores its argument and returns a type.

```
phantom : Nat⊤ -> Type
phantom = λ⊤ x.  Bool
```

To type check `idp` below, we must show that `phantom 0` equals `phantom 1`. Without compile-time irrelevance, we need to β-reduce both sides to show that the input and output types are equal.

```
idp : phantom 0⊤ -> phantom 1⊤
idp = λ x. x
```

However, in the presence of compile-time irrelevance, we can use the dependency information contained in the type of a function to reason about it abstractly. Because the function `f` below ignores its argument, it is sound to equate the input and output types.

```
ida : Π f :⊤ (Nat⊤ -> Type).  f 0⊤ -> f 1⊤
ida = λ⊤ f. λ x. x
```

In the absence of compile-time irrelevance, we cannot type-check `ida`. So compile-time irrelevance makes type checking more flexible.

Compile-time irrelevance can also make type checking faster when the types contain expensive computation that can be safely ignored. For example, consider the following program that type checks without compile-time irrelevance. However, in that case, the type checker must show that `fib 28` reduces to `317811`, where `fib` represents the Fibonacci function.

```
idn : Π f :⊤ (Nat⊤ -> Type). f (fib 28)⊤ -> f 317811⊤
idn = λ⊤ f. λ x. x
```

So far, we have used two annotations on variables and terms: ⊤ for irrelevant ones and 'unmarked' for relevant ones. We used ⊤ to mark both arguments that can be erased at runtime and arguments that can be safely ignored by the type checker. However, sometimes we need a finer distinction.

2.3 Strong Irrelevant Σ-types

Consider the type Σ`m:`$^{\top}$`Nat. Vec m a`, which contains pairs whose first component is marked as irrelevant. This type might be useful, say, for the output of a `filter` function for vectors, where the length of the output vector cannot be calculated statically. If we never need to use this length at runtime, then it would be good to mark it with $\top$ so that it need not be stored.[5]

However, marking `m` with $\top$ means that the first component of the pair of this type must also be *compile-time* irrelevant. This results in a significant limitation for strong Σ types: we cannot project the second component from the pair. Say we have `ys:`Σ`m:`$^{\top}$`Nat. Vec m a`. The type of (π_1 `ys`) is a `Nat` that can only be used in irrelevant positions. However, note that the argument `n` in `Vec n a` must be compile-time relevant; otherwise the type checker would equate `Vec 0 a` with `Vec 1 a`, making the length index meaningless. The type of (π_2 `ys`) would then be `Vec (`π_1 `ys) a`, which is ill-formed because an irrelevant term (π_1 ys) appears in a relevant position.

Therefore, we don't want to mark the first component of the output of `filter` with $\top$. However, if we leave it unmarked, we cannot erase it at runtime, something that we might want to. A way out of this quandry comes by considering terms that are run-time irrelevant but not compile-time irrelevant. Such terms exist between completely irrelevant and completely relevant terms. They should not depend upon irrelevant terms and relevant terms should not depend upon them. We mark such terms with a new annotation, C, with the constraints that 'unmarked' terms do not depend on C and C terms do not depend on $\top$ terms. The three annotations, then, correspond to the three levels of a lattice modelling secure information flow, with $\bot < C < \top$, using $\bot$ in lieu of 'unmarked'. We call the lattice $\mathcal{L}_I$, for irrelevance lattice. Using this lattice, we can type check the following `filter` function.

```
filter : Πn:⊤Nat.Πa:⊤Type.(a -> Bool) -> Vec n a -> Σm:ᶜNat. Vec m a
filter = λ⊤ n a. λ f vec.
          case vec of
            Nil -> (Zeroᶜ, Nil)
            Cons n1⊤ x xs
              | f x  -> ((Succ (π1 ys))ᶜ, Cons (π1 ys)⊤ x (π2 ys))
                          where
                            ys = filter n1⊤ a⊤ f xs
              | _    -> filter n1⊤ a⊤ f xs
```

Eisenberg et al. [14] observe that, in Haskell, it is important to use projection functions to access the components of the pair that results from the recursive call (as in π_1 `ys` and π_2 `ys`) to ensure that `filter` is not excessively strict. If `filter` instead used pattern matching to eliminate the pair returned by the recursive

[5] It is, however, safe for `m` to be used in a relevant position in the body of the Σ-type even when it is marked with $\top$. This marking indicates how the first component of a pair having this type is used, not how the bound variable `m` is used in the body of the type.

call, it would have needed to filter the entire vector before returning the first successful value. This `filter` function demonstrates the practical utility of strong irrelevant Σ-types because it supports the same run-time behavior of the usual list `filter` function but with a more richly-typed data structure.

3 A Simple Dependency Analyzing Calculus

Our ultimate goal is a dependent dependency calculus. However, we first start with a simply-typed version so that we can explain our approach to dependency analysis and non-interference in a simplified setting.

We call the calculus of this section SDC, for Simple Dependency Calculus. This calculus is parameterized by a lattice of *labels* or *grades*, which can also be thought of as security *levels*.[6] An excerpt of this calculus appears in Figure 1; it is an extension of the simply-typed λ-calculus with a grade-indexed modal type $T^{\ell}\ A$. The modal type $T^{\ell}\ A$ can be thought of as putting a security barrier of grade ℓ around the values of A. The calculus itself is also *graded*, which means that in a typing judgment, the derived term and every variable in the context carries a label or grade. (The specification of the full system, which includes unit, products and sums, appears in the extended version of this paper [12].)

3.1 Type System

The typing judgment has the form $\Omega \vdash a :^{\ell} A$ which means that "ℓ is allowed to observe a" or that "a is visible at ℓ". Selected typing rules for SDC appear in Figure 1. Most rules are straightforward and propagate the level of the sub-terms to the expression.

The rule SDC-Var requires that the grade of the variable in the context must be less than or equal to the grade of the observer. In other words, an observer at level ℓ is allowed to use a variable from level k if and only if $k \leq \ell$. If the variable's level is too high, then this rule does not apply, ensuring that information can always flow to more secure levels but never to less secure ones. Abstraction rule SDC-Abs uses the current level of the expression for the newly introduced variable in the context. This makes sense because the argument to the function is checked at the same level in rule SDC-App.

The modal type, introduced and eliminated with rule SDC-Return and rule SDC-Bind respectively, manipulates the levels. The former says that, if a term is $(\ell \vee \ell_0)$-secure, then we can put it in an ℓ_0-secure box and release it at level ℓ. An ℓ_0-secure boxed term can be unboxed only by someone who has security clearance for ℓ_0, as we see in the latter rule. The join operation in rule SDC-Bind ensures that b can depend on a only if b itself is ℓ_0-secure or $\ell_0 \leq \ell$.

[6] We use the terms label, level and grade interchangeably.

(Grammar)

$$
\begin{array}{lll}
\textit{labels} & \ell, k & ::= \bot \mid \top \mid k \wedge \ell \mid k \vee \ell \mid \ldots \\
\textit{types} & A, B & ::= \mathbf{Unit} \mid A \to B \mid T^{\ell}\ A \\
\textit{terms} & a, b & ::= x \mid \lambda x\!:\!A.a \mid a\ b \qquad \textit{variables and functions} \\
 & & \mid\ \eta^{\ell}\ a \mid \mathbf{bind}^{\ell}\ x = a\ \mathbf{in}\ b \quad \textit{graded modality} \\
\textit{contexts} & \Omega & ::= \varnothing \mid \Omega, x\!:^{\ell} A
\end{array}
$$

$\boxed{\Omega \vdash a :^{\ell} A}$ *(Typing rules)*

$$\text{SDC-Var}\ \frac{\ell_0 \le \ell \qquad x\!:^{\ell_0} A \in \Omega}{\Omega \vdash x :^{\ell} A} \qquad \text{SDC-Abs}\ \frac{\Omega, x\!:^{\ell} A \vdash b :^{\ell} B}{\Omega \vdash \lambda x\!:\!A.b :^{\ell} A \to B} \qquad \text{SDC-App}\ \frac{\Omega \vdash b :^{\ell} A \to B \qquad \Omega \vdash a :^{\ell} A}{\Omega \vdash b\ a :^{\ell} B}$$

$$\text{SDC-Return}\ \frac{\Omega \vdash a :^{\ell \vee \ell_0} A}{\Omega \vdash \eta^{\ell_0}\ a :^{\ell} T^{\ell_0}\ A} \qquad \text{SDC-Bind}\ \frac{\Omega \vdash a :^{\ell} T^{\ell_0}\ A \qquad \Omega, x\!:^{\ell \vee \ell_0} A \vdash b :^{\ell} B}{\Omega \vdash \mathbf{bind}^{\ell_0}\ x = a\ \mathbf{in}\ b :^{\ell} B}$$

$\boxed{a \rightsquigarrow a'}$ *(Small step)*

$$\text{SDCStep-Beta}\ \frac{}{(\lambda x\!:^{\ell} A.a)\ b^{\ell} \rightsquigarrow a\{b/x\}} \qquad \text{SDCStep-BindBeta}\ \frac{}{\mathbf{bind}^{\ell}\ x = \eta^{\ell}\ a\ \mathbf{in}\ b \rightsquigarrow b\{a/x\}}$$

Fig. 1. Simple Dependency Calculus (Excerpt)

3.2 Meta-theoretic Properties

This type system satisfies the following properties related to levels.

First, we can always weaken our assumptions about the variables in the context. If a term is derivable with an assumption held at some grade, then that term is also derivable with that assumption held at any lower grade. Below, for any two contexts Ω_1, Ω_2, we say that $\Omega_1 \le \Omega_2$ iff they are the same modulo the grades and further, for any x, if $x\!:^{\ell_1} A \in \Omega_1$ and $x\!:^{\ell_2} A \in \Omega_2$, then $\ell_1 \le \ell_2$.

Lemma 1 (Narrowing). *If $\Omega' \vdash a :^{\ell} A$ and $\Omega \le \Omega'$, then $\Omega \vdash a :^{\ell} A$.*

Narrowing says that we can always downgrade any variable in the context. Conversely, we cannot upgrade context variables in general, but we can upgrade them to the level of the judgment.

Lemma 2 (Restricted Upgrading). *If $\Omega_1, x\!:^{\ell_0} A, \Omega_2 \vdash b :^{\ell} B$ and $\ell_1 \le \ell$, then $\Omega_1, x\!:^{\ell_0 \vee \ell_1} A, \Omega_2 \vdash b :^{\ell} B$.*

The restricted upgrading lemma is needed to show subsumption. Subsumption states that, if a term is visible at some grade, then it is also visible at all higher grades.

Lemma 3 (Subsumption). *If $\Omega \vdash a :^{\ell} A$ and $\ell \le k$, then $\Omega \vdash a :^{k} A$.*

Subsumption is necessary (along with a standard weakening lemma) to show that substitution holds for this language. For substitution, we need to ensure that the level of the variable matches up with that of the substituted expression.

Lemma 4 (Substitution). *If* $\Omega_1, x :^{\ell_0} A, \Omega_2 \vdash b :^{\ell} B$ *and* $\Omega_1 \vdash a :^{\ell_0} A$, *then* $\Omega_1, \Omega_2 \vdash b\{a/x\} :^{\ell} B$.

SDC terms are reduced using a call-by-name strategy. An excerpt of the small-step semantics appears in Figure 1. Note how the labels on the introduction form and the corresponding elimination form match up in rules SDCStep-Beta and SDCStep-BindBeta. Further, note that we could have also used a call-by-value strategy to reduce SDC terms; we chose a call-by-name strategy because our development is motivated by potential applications in Haskell.

For a call-by-name operational semantics, the above lemmas allow us to prove, a standard progress and preservation based type soundness result, which we omit here.

Next, we show that our type system is secure by proving non-interference.

3.3 A Syntactic Proof of Non-interference

When users with low-security clearance are oblivious to high-security data, we say that the system enjoys *non-interference*. Non-interference results from level-specific views of the world. The values η^H **True** and η^H **False** appear the same to an L-user while an H-user can differentiate between them. To capture this notion of a level-specific view, we design an indexed equivalence relation on open terms, $\sim_\ell$, called *indexed indistinguishability*, and shown in Figure 2. To define this relation, we need the labels of the variables in the context but not their types. So, we use grade-only contexts Φ, defined as $\Phi ::= \varnothing \mid \Phi, x : \ell$. These contexts are like graded contexts Ω without the type information on variables, also denoted by $|\Omega|$.

Informally, $\Phi \vdash a \sim_\ell b$ means that a and b appear the same to an ℓ-user. For example, η^H **True** $\sim_L \eta^H$ **False** but $\neg(\eta^H$ **True** $\sim_H \eta^H$ **False**$)$. We define this relation $\sim_\ell$ by structural induction on terms. We think of terms as ASTs annotated at various nodes with labels, say ℓ_0, that determine whether an observer ℓ is allowed to look at the corresponding sub-tree. If $\ell_0 \leq \ell$, then observer ℓ can start exploring the sub-tree; otherwise the entire sub-tree appears as a blob. So we can also read $\Phi \vdash a \sim_\ell b$ as: "a is syntactically equal to b at all parts of the terms marked with any label ℓ_0, where $\ell_0 \leq \ell$, but may be arbitrarily different elsewhere."

Note the rule SGEq-Return in Figure 2. It uses an auxiliary relation, $\Phi \vdash_\ell^{\ell_0} a_1 \sim a_2$. This auxiliary *extended equivalence* relation $\Phi \vdash_\ell^{\ell_0} a_1 \sim a_2$ formalizes

[7] Because this relation is untyped, its analogue for DDC is similar. For each lemma below, we include a reference to the location in the Coq development where it may be found for the dependent system.

$\boxed{\Phi \vdash a \sim_\ell b}$ *(Indexed Indistinguishability)*

$$\textsc{SGEq-Var}\ \frac{x : \ell_0\ \mathbf{in}\ \Phi \qquad \ell_0 \leq \ell}{\Phi \vdash x \sim_\ell x} \qquad \textsc{SGEq-Abs}\ \frac{\Phi, x : \ell \vdash b_1 \sim_\ell b_2}{\Phi \vdash \lambda x : A.b_1 \sim_\ell \lambda x : A.b_2} \qquad \textsc{SGEq-App}\ \frac{\Phi \vdash b_1 \sim_\ell b_2 \quad \Phi \vdash a_1 \sim_\ell a_2}{\Phi \vdash b_1\ a_1 \sim_\ell b_2\ a_2}$$

$$\textsc{SGEq-Return}\ \frac{\Phi \vdash^{\ell_0}_{\ell} a_1 \sim a_2}{\Phi \vdash \eta^{\ell_0}\ a_1 \sim_\ell \eta^{\ell_0}\ a_2} \qquad \textsc{SGEq-Bind}\ \frac{\Phi \vdash a_1 \sim_\ell a_2 \quad \Phi, x : \ell_0 \vee \ell \vdash b_1 \sim_\ell b_2}{\Phi \vdash \mathbf{bind}^{\ell_0}\ x = a_1\ \mathbf{in}\ b_1 \sim_\ell \mathbf{bind}^{\ell_0}\ x = a_2\ \mathbf{in}\ b_2}$$

$\boxed{\Phi \vdash^{\ell_0}_{\ell} a_1 \sim a_2}$

$$\textsc{SEq-Leq}\ \frac{\ell_0 \leq \ell \qquad \Phi \vdash a_1 \sim_\ell a_2}{\Phi \vdash^{\ell_0}_{\ell} a_1 \sim a_2} \qquad \textsc{SEq-Nleq}\ \frac{\neg(\ell_0 \leq \ell)}{\Phi \vdash^{\ell_0}_{\ell} a_1 \sim a_2}$$

Fig. 2. Indexed indistiguishability for SDC (Excerpt)

the idea discussed above: if $\ell_0 \leq \ell$, then a_1 and a_2 must be indistinguishable at ℓ; otherwise, they may be arbitrary terms.

Now, we explore some properties of the indistinguishability relation.[7]
If we remove the second component from an indistinguishability relation, $\Phi \vdash a \sim_\ell b$, we get a new judgment, $\Phi \vdash a : \ell$, called grading judgment. Now, corresponding to every indistinguishability rule, we define a grading rule where the indistinguishability judgments have been replaced with their grading counterparts. Terms derived using these grading rules are called well-graded. We can show that well-typed terms are well-graded.

Lemma 5 (Typing implies grading). *If* $\Omega \vdash a :^\ell A$ *then* $|\Omega| \vdash a : \ell$.

Lemma 6 (Equivalence). *Indexed indistinguishability at* ℓ *is an equivalence relation on well-graded terms at* ℓ.

The above lemma shows that indistinguishability is an equivalence relation. Observe that at the highest element of the lattice, $\top$, this equivalence degenerates to the identity relation.
Indistinguishability is closed under extended equivalence. The following is like a substitution lemma for the relation.

Lemma 7 (Indistinguishability under substitution). *If* $\Phi, x : \ell \vdash b_1 \sim_k b_2$ *and* $\Phi \vdash^\ell_k a_1 \sim a_2$ *then* $\Phi \vdash b_1\{a_1/x\} \sim_k b_2\{a_2/x\}$.

With regard to the above lemma, consider the situation when $\neg(\ell \leq k)$, for example, when $\ell = H$ and $k = L$. In such a situation, for any two terms a_1

and a_2, if $\Phi, x : \ell \vdash b_1 \sim_k b_2$, then $\Phi \vdash b_1\{a_1/x\} \sim_k b_2\{a_2/x\}$. Let us work out a concrete example. For a typing derivation $x :^H A \vdash b :^L \mathbf{Bool}$, we have, by lemmas 5 and 6, $x : H \vdash b \sim_L b$. Then, $\varnothing \vdash b\{a_1/x\} \sim_L b\{a_2/x\}$. This is almost non-interference in action. What's left to show is that the indistinguishability relation respects the small step semantics, written $a_1 \rightsquigarrow a_2$. The small-step relation is standard call-by-name reduction.

Theorem 1 (Non-interference). *If $\Phi \vdash a_1 \sim_k a_1'$ and $a_1 \rightsquigarrow a_2$ then there exists some a_2' such that $a_1' \rightsquigarrow a_2'$ and $\Phi \vdash a_2 \sim_k a_2'$.*

Since the step relation is deterministic, in the above lemma, there is exactly one such a_2' that a_1' steps to. Now, going back to our last example, we see that $b\{a_1/x\}$ and $b\{a_2/x\}$ take steps in tandem and they are L-indistinguishable after each and every step. Since the language itself is terminating, both the terms reduce to boolean values, values that are themselves L-indistinguishable as well. But the indistinguishability for boolean values is just the identity relation. This means that $b\{a_1/x\}$ and $b\{a_2/x\}$ reduce to the same value.

The indistinguishability relation gives us a syntactic method of proving non-interference for programs derived in SDC. Essentially, we show that a user with low-security clearance cannot distinguish between high security values just by observing program behavior.

Next, we show that SDC is no less expressive than the terminating fragment of DCC.

3.4 Relation with Sealing Calculus and Dependency Core Calculus

SDC is extremely similar to the sealing calculus $\lambda^{[]}$ of Shikuma and Igarashi [29]. Like SDC, $\lambda^{[]}$ has a label on the typing judgment.[8] But unlike SDC, $\lambda^{[]}$ uses standard ungraded typing contexts Γ. Both the calculi have the same types. As far as terms are concerned, there is only one difference. The sealing calculus has an **unseal** term whereas SDC uses **bind**. We present the rules for sealing and unsealing terms in $\lambda^{[]}$ below.[9]

$$\textsc{Sealing-Seal}\ \frac{\Gamma \vdash a :^{\ell \vee \ell_0} A}{\Gamma \vdash \eta^{\ell_0}\ a :^{\ell} T^{\ell_0}\ A} \qquad \textsc{Sealing-Unseal}\ \frac{\Gamma \vdash a :^{\ell} T^{\ell_0}\ A \quad \ell_0 \leq \ell}{\Gamma \vdash \mathbf{unseal}^{\ell_0}\, a :^{\ell} A}$$

Shikuma and Igarashi [29] have shown that $\lambda^{[]}$ is equivalent to $\mathrm{DCC}_{\mathrm{pc}}$, an extension of the terminating fragment of DCC. Therefore, we compare SDC to DCC by simulating $\lambda^{[]}$ in SDC. For this, we define a translation $\overline{\cdot}$, from $\lambda^{[]}$ to SDC. Most of the cases are handled inductively in a straightforward manner. For **unseal**, we have, $\overline{\mathbf{unseal}^{\ell}\, a} := \mathbf{bind}^{\ell}\, x = \overline{a}\, \mathbf{in}\, x$.

[8] Note that our labels correspond to observer levels of [29], which can be viewed as a lattice.

[9] We take the liberty of making small cosmetic changes in the presentation.

With this translation, we can give a forward and a backward simulation connecting the two languages. The reduction relation $\rightsquigarrow$ below is full reduction for both the languages, the reduction strategy used by Shikuma and Igarashi [29] for $\lambda^{[]}$. Full reduction is a non-deterministic reduction strategy whereby a β-redex in any sub-term may be reduced.

Theorem 2 (Forward Simulation). *If $a \rightsquigarrow a'$ in $\lambda^{[]}$, then $\overline{a} \rightsquigarrow \overline{a'}$ in SDC.*

Theorem 3 (Backward Simulation). *For any term a in $\lambda^{[]}$, if $\overline{a} \rightsquigarrow b$ in SDC, then there exists a' in $\lambda^{[]}$ such that $b = \overline{a'}$ and $a \rightsquigarrow a'$.*

The translation also preserves typing. In fact, a source term and its target have the same type. Below, for an ordinary context Γ, the graded context Γ^{ℓ} denotes Γ with the labels for all the variables set to ℓ.

Theorem 4 (Translation Preserves Typing). *If $\Gamma \vdash a :^{\ell} A$, then $\Gamma^{\ell} \vdash \overline{a} :^{\ell} A$.*

The above translation shows that the terminating fragment of DCC can be embedded into SDC. Therefore SDC is at least as expressive as the terminating fragment of DCC. Further, SDC lends itself nicely to syntactic proof techniques for non-interference. This approach generalizes to more expressive systems, as we shall see in the next section, where we extend SDC to a general dependent dependency calculus.

4 A Dependent Dependency Analyzing Calculus

$$
\begin{array}{lll}
a, A, b, B ::= s \mid \mathbf{unit} \mid \mathbf{Unit} & & \textit{sorts and unit} \\
\quad \mid \Pi x :^{\ell} A.B \mid x \mid \lambda x :^{\ell} A.a \mid a\ b^{\ell} & & \textit{dependent functions} \\
\quad \mid \Sigma x :^{\ell} A.B \mid (a^{\ell}, b) \mid \mathbf{let}\ (x^{\ell}, y)\ =\ a\ \mathbf{in}\ b & & \textit{dependent pairs} \\
\quad \mid A + B \mid \mathbf{inj}_1\ a \mid \mathbf{inj}_2\ a \mid \mathbf{case}\ a\ \mathbf{of}\ b_1; b_2 & & \textit{disjoint unions}
\end{array}
$$

Fig. 3. Dependent Dependency Calculus Grammar (Types and Terms)

Here and in the next section, we present dependently-typed languages, with dependency analysis in the style of SDC. The first extension, called $\mathrm{DDC}^{\top}$ is a straightforward integration of labels and dependent types. This system subsumes SDC, and so can be used for the same purposes. Here, we show how it can be used to analyze *run-time irrelevance*. Then, in Section 5, we generalize this system to DDC, which allows definitional equality to ignore unnecessary sub-terms, thus also enabling *compile-time irrelevance*. We present the system in this way both

to simplify the presentation and to show that DDC$^\top$ is an intermediate point in the design space.

Both DDC$^\top$ and DDC are pure type systems [5]. They share the same syntax, shown in Figure 3, combining terms and types into the same grammar. They are parameterized by a set of sorts s, a set of axioms $\mathcal{A}(s_1, s_2)$ which is a binary relation on sorts, and a set of rules $\mathcal{R}(s_1, s_2, s_3)$ which is a ternary relation on sorts. For simplicity, we assume, without loss of generality, that for every sort s_1, there is some sort s_2, such that $\mathcal{A}(s_1, s_2)$.[10]

We annotate several syntactic forms with grades for dependency analysis. The dependent function type, written $\Pi x :^\ell A.B$, includes the grade of the argument to a function having this type. Similarly, the dependent pair type, written $\Sigma x :^\ell A.B$, includes the grade of the first component of a pair having this type. [11] We can interpret these types as a fusion of the usual, ungraded dependent types and the graded modality $T^\ell\ A$ we saw earlier. In other words, $\Pi x :^\ell A.B$ acts like the type $\Pi y : (T^\ell\ A).\mathbf{bind}\, x = y\, \mathbf{in}\, B$ and $\Sigma x :^\ell A.B$ acts like the type $\Sigma y : (T^\ell\ A).\mathbf{bind}\, x = y\, \mathbf{in}\, B$. Because of this fusion, we do not need to add the graded modality type as a separate form—we can define $T^\ell\ A$ as $\Sigma x :^\ell A.\mathbf{Unit}$. Using $\Pi x :^\ell A.B$ instead of $\Pi y : (T^\ell\ A).\mathbf{bind}\, x = y\, \mathbf{in}\, B$ has an advantage: the former allows x to be held at differing grades while type checking B and the body of a function having this Π-type while the latter requires x to be held at the same grade in both the cases. We utilize this flexibility in Section 5.

4.1 DDC$^\top$: Π-types

The core typing rules for DDC$^\top$ appear in Figure 4. As in the simple type system, the variables in the context are labelled and the judgement itself includes a label ℓ. Rule DCT-Var is similar to its counterpart in the simply-typed language: the variable being observed must be graded less than or equal to the level of the observer. Rule DCT-Pi propagates the level of the expression to the subterms of the Π-type. Note that this type is annotated with an arbitrary label ℓ_0: the purpose of this label ℓ_0 is to denote the level at which the argument to a function having this type may be used.

In rule DCT-Abs, the parameter of the function is introduced into the context at level $\ell_0 \vee \ell$ (akin to rule SDC-Bind). In rule DCT-App, the argument to the function is checked at level $\ell_0 \vee \ell$ (akin to rule SDC-Return). Note that the Π-type is checked at $\top$ in rule DCT-Abs. In DDC$^\top$, level $\top$ corresponds to 'compile time' observers and motivates the superscript $\top$ in the language name.

Rule DCT-Conv converts the type of an expression to an equivalent type. The judgment $|\Omega| \vdash A \equiv_\top B$ is a label-indexed definitional equality relation

[10] This assumption does not lead to any loss in generality because given a pure type system (S', A', R') that does not meet the above condition, we can provide another pure type system (S'', A'', R''), where $S'' = S' \cup \{\hexagon\}$ (given $\hexagon \notin S'$) and $A'' = A' \cup \{(s, \hexagon) | s \in S''\}$ and $R'' = R'$, such that there exists a straightforward bisimulation between the two systems.

[11] We use standard abbreviations when x is not free in B: we write ${}^\ell A \to B$ for $\Pi x :^\ell A.B$

$\boxed{\Omega \vdash a :^{\ell} A}$ *(Typing)*

DCT-Var
$$\frac{\ell_0 \leq \ell \qquad x :^{\ell_0} A \in \Omega}{\Omega \vdash x :^{\ell} A}$$

DCT-Type
$$\frac{\mathcal{A}(s_1, s_2)}{\Omega \vdash s_1 :^{\ell} s_2}$$

DCT-Pi
$$\frac{\Omega \vdash A :^{\ell} s_1 \qquad \Omega, x :^{\ell} A \vdash B :^{\ell} s_2 \qquad \mathcal{R}(s_1, s_2, s_3)}{\Omega \vdash \Pi x :^{\ell_0} A.B :^{\ell} s_3}$$

DCT-Abs
$$\frac{\Omega, x :^{\ell_0 \vee \ell} A \vdash b :^{\ell} B \qquad \Omega \vdash (\Pi x :^{\ell_0} A.B) :^{\top} s}{\Omega \vdash \lambda x :^{\ell_0} A.b :^{\ell} \Pi x :^{\ell_0} A.B}$$

DCT-App
$$\frac{\Omega \vdash b :^{\ell} \Pi x :^{\ell_0} A.B \qquad \Omega \vdash a :^{\ell_0 \vee \ell} A}{\Omega \vdash b\ a^{\ell_0} :^{\ell} B\{a/x\}}$$

DCT-Conv
$$\frac{\Omega \vdash a :^{\ell} A \qquad |\Omega| \vdash A \equiv_{\top} B \qquad \Omega \vdash B :^{\top} s}{\Omega \vdash a :^{\ell} B}$$

Fig. 4. DDC$^{\top}$ type system (core rules)

instantiated to $\top$. This relation is the closure of the indexed indistinguishability relation (Section 3.3) under small-step call-by-name evaluation. When instantiated to $\top$, the relation degenerates to β-equivalence. So the rule DCT-Conv is essentially casting a term to a β-equivalent type; however, in the next section, we utilize the flexibility of label-indexing to cast a term to a type that may not be β-equivalent. Also, note that the equality relation itself is untyped. As such, we need the third premise to guarantee that the new type is well-formed.

4.2 DDC$^{\top}$: Σ-types

The language DDC$^{\top}$ includes Σ types, as specified by the rules below.

DCT-WSigma
$$\frac{\Omega \vdash A :^{\ell} s_1 \qquad \Omega, x :^{\ell} A \vdash B :^{\ell} s_2 \qquad \mathcal{R}(s_1, s_2, s_3)}{\Omega \vdash \Sigma x{:}^{\ell_0} A.B :^{\ell} s_3}$$

DCT-WPair
$$\frac{\Omega \vdash a :^{\ell_0 \vee \ell} A \qquad \Omega \vdash b :^{\ell} B\{a/x\} \qquad \Omega \vdash \Sigma x{:}^{\ell_0} A.B :^{\top} s}{\Omega \vdash (a^{\ell_0}, b) :^{\ell} \Sigma x{:}^{\ell_0} A.B}$$

Like Π-types, Σ-types include a grade that is not related to how the bound variable is used in the body of the type. The grade indicates the level at which the first component of a pair having the Σ-type may be used. In rule DCT-WPair, we check the first component a of the pair at a level raised by ℓ_0, the level annotating the type, akin to rule SDC-Return. The second component b is checked at the current level.

and ${}^{\ell}A \times B$ for $\Sigma x{:}^{\ell} A.B$.

$$\textsc{DCT-LetPair}\quad \frac{\Omega \vdash a :^{\ell} \Sigma x{:}^{\ell_0} A.B \qquad \Omega, x{:}^{\ell_0 \vee \ell} A, y{:}^{\ell} B \vdash c :^{\ell} C\{(x^{\ell_0}, y)/z\} \qquad \Omega, z{:}^{\top} (\Sigma x{:}^{\ell_0} A.B) \vdash C :^{\top} s}{\Omega \vdash \mathbf{let}\ (x^{\ell_0}, y)\ =\ a\ \mathbf{in}\ c :^{\ell} C\{a/z\}}$$

The rule DCT-LetPair eliminates pairs using dependently-typed pattern matching. The pattern variables x and y are introduced into the context while checking the body c. Akin to rule SDC-Bind, the level of the first pattern variable, x, is raised by ℓ_0. The result type C is refined by the pattern match, informing the type system that the pattern (x^{ℓ_0}, y) is equal to the scrutinee a.

Because of this refinement in the result type, we can define the projection operations through pattern matching. In particular, the first projection, $\pi_1^{\ell_0}\ a := \mathbf{let}\ (x^{\ell_0}, y)\ =\ a\ \mathbf{in}\ x$ while the second projection, $\pi_2^{\ell_0} a := \mathbf{let}\ (x^{\ell_0}, y)\ =\ a\ \mathbf{in}\ y$. These projections can be type checked according to the following derived rules:

$$\textsc{DCT-Proj1}\quad \frac{\Omega \vdash a :^{\ell} \Sigma x{:}^{\ell_0} A.B \qquad \ell_0 \leq \ell}{\Omega \vdash \pi_1^{\ell_0}\ a :^{\ell} A} \qquad\qquad \textsc{DCT-Proj2}\quad \frac{\Omega \vdash a :^{\ell} \Sigma x{:}^{\ell_0} A.B}{\Omega \vdash \pi_2^{\ell_0} a :^{\ell} B\{\pi_1^{\ell_0}\ a/x\}}$$

Note that the derived rule DCT-Proj1 limits access to the first component through the premise $\ell_0 \leq \ell$, akin to rule Sealing-Unseal. This condition makes sense because it aligns the observability of the first component of the pair with the label on the Σ-type.

4.3 Embedding SDC into DDC$^{\top}$

Here, we show how to embed SDC into DDC$^{\top}$.

We define a translation function, $\overline{\cdot}$, that takes the types and terms in SDC to terms in DDC$^{\top}$. For types, the translation is defined as: $\overline{\overline{A \to B}} := {}^{\perp}\overline{\overline{A}} \to \overline{\overline{B}}$, $\overline{\overline{A \times B}} := {}^{\perp}\overline{\overline{A}} \times \overline{\overline{B}}$ and $\overline{\overline{T^{\ell}\ A}} := \Sigma x{:}^{\ell} \overline{\overline{A}}.\mathbf{Unit}$. For terms, the translation is straightforward except for the following cases: $\overline{\overline{\eta^{\ell}\ a}} := (\overline{\overline{a}}^{\ell}, \mathbf{unit})$ and $\overline{\overline{\mathbf{bind}^{\ell}\ x = a\ \mathbf{in}\ b}} := \mathbf{let}\ (x^{\ell}, y)\ =\ \overline{\overline{a}}\ \mathbf{in}\ \overline{\overline{b}}$, where y is a fresh variable. By lifting the translation to contexts, we show that translation preserves typing.

Theorem 5 (Trans. Preserves Typing). *If $\Omega \vdash a :^{\ell} A$, then $\overline{\overline{\Omega}} \vdash \overline{\overline{a}} :^{\ell} \overline{\overline{A}}$.*

Next, assuming a standard call-by-name small-step semantics for both the languages, we can provide a bisimulation.

Theorem 6 (Forward Simulation). *If $a \rightsquigarrow a'$ in SDC, then $\overline{\overline{a}} \rightsquigarrow \overline{\overline{a'}}$ in* DDC$^{\top}$.

Theorem 7 (Backward Simulation). *For any term a in SDC, if $\overline{\overline{a}} \rightsquigarrow b$ in* DDC$^{\top}$*, then there exists a' in SDC such that $b = \overline{\overline{a'}}$ and $a \rightsquigarrow a'$.*

Hence, SDC can be embedded into DDC$^{\top}$, preserving meaning. As such, DDC$^{\top}$ can analyze dependencies in general.

4.4 Run-time Irrelevance

Next, we show how to track run-time irrelevance using $\mathrm{DDC}^{\top}$. We use the two element lattice $\{\bot, \top\}$ with $\bot < \top$ such that $\bot$ and $\top$ correspond to run-time relevant and run-time irrelevant terms respectively. So, we need to erase terms marked with $\top$. However, we first define a general indexed erasure function, $\lfloor \cdot \rfloor_\ell$, on $\mathrm{DDC}^{\top}$ terms, that erases everything an ℓ-user should not be able to see. The function is defined by straightforward recursion in most cases. For example, $\lfloor x \rfloor_\ell := x$ and $\lfloor \Pi x{:}^{\ell_0} A.B \rfloor_\ell := \Pi x{:}^{\ell_0} \lfloor A \rfloor_\ell . \lfloor B \rfloor_\ell$ and $\lfloor \lambda^{\ell_0} x.b \rfloor_\ell := \lambda^{\ell_0} x. \lfloor b \rfloor_\ell$. The interesting cases are:
$\lfloor b\ a^{\ell_0} \rfloor_\ell := (\lfloor b \rfloor_\ell\ \lfloor a \rfloor_\ell^{\ell_0})$ if $\ell_0 \leq \ell$ and $(\lfloor b \rfloor_\ell\ \mathbf{unit}^{\ell_0})$ otherwise,
$\lfloor (a^{\ell_0}, b) \rfloor_\ell := (\lfloor a \rfloor_\ell^{\ell_0}, \lfloor b \rfloor_\ell)$ if $\ell_0 \leq \ell$ and $(\mathbf{unit}^{\ell_0}, \lfloor b \rfloor_\ell)$ otherwise.
They are so defined because if $\neg(\ell_0 \leq \ell)$, an ℓ-user should not be able to see a, so we replace it with **unit**.

This erasure function is closely related to the indistinguishability relation, we saw in Section 3.3, extended to a dependent setting. (This definition appears in the extended version of this paper [12].) The erasure function maps the equivalence classes formed by the indistinguishability relation to their respective canonical elements. We have verified the following lemmas using the Coq proof assistant. Footnotes mark the file and lemma name of the corresponding mechanized results.

Lemma 8 (Canonical Element[12]). *If $\Phi \vdash a_1 \sim_\ell a_2$, then $\lfloor a_1 \rfloor_\ell = \lfloor a_2 \rfloor_\ell$.*

Further, a well-graded term and its erasure are indistinguishable.

Lemma 9 (Erasure Indistinguishability[13]). *If $\Phi \vdash a : \ell$, then $\Phi \vdash a \sim_\ell \lfloor a \rfloor_\ell$.*

Next, we can show that erased terms simulate the reduction behavior of their unerased counterparts.

Lemma 10 (Erasure Simulation[14]). *If $\Phi \vdash a : \ell$ and $a \rightsquigarrow b$, then $\lfloor a \rfloor_\ell \rightsquigarrow \lfloor b \rfloor_\ell$. Otherwise, if a is a value, then so is $\lfloor a \rfloor_\ell$.*

This lemma follows from Lemma 9 and the non-interference theorem (Theorem 1). Therefore, it is safe to erase, before run time, all sub-terms marked with $\top$.

This shows that we can correctly analyze run-time irrelevance using $\mathrm{DDC}^{\top}$. However, supporting compile-time irrelevance requires some changes to the system. We take them up in the next section.

[12] `erasure.v:Canonical_element`. [13] `erasure.v:Erasure_Indistinguishability`
[14] `erasure.v:Step_erasure,Value_erasure`

5 DDC: Run-time and Compile-time Irrelevance

5.1 Towards Compile-time Irrelevance

Recall that terms which may be safely ignored while checking for type equality are said to be compile-time irrelevant. In DDC$^\top$, the conversion rule DCT-Conv checks for type equality at $\top$.

$$\frac{\Omega \vdash a :^{\ell} A \qquad |\Omega| \vdash A \equiv_{\top} B \qquad \Omega \vdash B :^{\top} s}{\Omega \vdash a :^{\ell} B}\ \text{DCT-Conv}$$

The equality judgment used in this rule $\Phi \vdash a \equiv_{\top} b$ is an instantiation of the general judgment $\Phi \vdash a \equiv_{\ell} b$, which is the closure of the indistinguishability relation at ℓ under β-equivalence. When ℓ is $\top$, indistinguishability is just identity. As such, the equality relation at $\top$ degenerates to standard β-equivalence. So, rule DCT-Conv does not ignore any part of the terms when checking for type equality.

To support compile-time irrelevance then, we need the conversion rule to use equality at some grade strictly less than $\top$ so that $\top$-marked terms may be ignored. For the irrelevance lattice $\mathcal{L}_I$, the level C can be used for this purpose. For any other lattice $\mathcal{L}$, we can add two new elements, C and $\top$, above every other existing element, such that $\mathcal{L} < C < \top$, and thereafter use level C for this purpose. So, for any lattice, we can support compile-time irrelevance by equating types at C.

Referring back to the examples in Section 2.2, note that for `phantom : Nat`$^\top$ `-> Type`, we have `phantom 0`$^\top \equiv_C$ `phantom 1`$^\top$. With this equality, we can type-check `idp : phantom 0`$^\top$ `-> phantom 1`$^\top$ `= λ x. x`, even without knowing the definition of `phantom`.

Now, observe that in rule DCT-Conv, the new type B is also checked at $\top$. If we want to check for type equality at C, we need to make sure that the types themselves are checked at C. However, checking types at C would rule out variables marked at $\top$ from appearing in them. This would restrict us from expressing many examples, including the polymorphic identity function.

To move out of this impasse, we take inspiration from EPTS [20,21]. The key idea, adapted from [20], is to use a judgment of the form $C \wedge \Omega \vdash a :^C A$ instead of a judgment of the form $\Omega \vdash a :^{\top} A$. The operation $C \wedge \Omega$ takes the point-wise meet of the labels in the context Ω with C, essentially reducing any label marked as $\top$ to C, making it available for use in a C-expression. This operation, called *truncation*, makes $\top$ marked variables available at C. Other systems also use similar mechanisms for tracking irrelevance — for example, we can see a relation between this idea and analogous ones in [27] and [3]. In these systems, "context resurrection" operation makes proof variables and irrelevant variables in the context available for use, similar to how $C \wedge \Omega$ makes $\top$-marked variables in the context available for use.

5.2 DDC: Basics

Next, we design a general dependency analyzing calculus, DDC, that takes advantage of compile-time irrelevance in its type system. DDC is a generalization of DDC$^\top$ and EPTS$^\bullet$ [20]. When C equals $\top$, DDC degenerates to DDC$^\top$, that does not use compile-time irrelevance. When C equals $\bot$, DDC degenerates to EPTS$^\bullet$, that identifies compile-time and run-time irrelevance. A crucial distinction between EPTS$^\bullet$ and DDC is that while the former is tied to a two element lattice, the latter can use any lattice. Thus, not only can DDC distinguish between run-time and compile-time irrelevance, but also it can simultaneously track other dependencies.

$\boxed{\Omega \vdash a :^{\ell} A}$ *(*DDC *core typing rules)*

$$\textsc{T-Var}\ \frac{\ell_0 \leq \ell \qquad x :^{\ell_0} A \in \Omega \qquad \ell \leq C}{\Omega \vdash x :^{\ell} A} \qquad \textsc{T-Type}\ \frac{\ell \leq C \qquad \mathcal{A}(s_1, s_2)}{\Omega \vdash s_1 :^{\ell} s_2} \qquad \textsc{T-Pi}\ \frac{\Omega \vdash A :^{\ell} s_1 \qquad \Omega, x :^{\ell} A \vdash B :^{\ell} s_2 \qquad \mathcal{R}(s_1, s_2, s_3)}{\Omega \vdash \Pi x :^{\ell_0} A.B :^{\ell} s_3}$$

$$\textsc{T-AbsC}\ \frac{\Omega, x :^{\ell_0 \vee \ell} A \vdash b :^{\ell} B \qquad \Omega \Vdash (\Pi x :^{\ell_0} A.B) :^{\top} s}{\Omega \vdash \lambda x :^{\ell_0} A.b :^{\ell} \Pi x :^{\ell_0} A.B} \qquad \textsc{T-AppC}\ \frac{\Omega \vdash b :^{\ell} \Pi x :^{\ell_0} A.B \qquad \Omega \Vdash a :^{\ell_0 \vee \ell} A}{\Omega \vdash b\ a^{\ell_0} :^{\ell} B\{a/x\}} \qquad \textsc{T-ConvC}\ \frac{\Omega \vdash a :^{\ell} A \qquad |C \wedge \Omega| \vdash A \equiv_C B \qquad \Omega \Vdash B :^{\top} s}{\Omega \vdash a :^{\ell} B}$$

$\boxed{\Omega \Vdash a :^{\ell} A}$ *(Truncate at* $\top$*)*

$$\textsc{CT-Leq}\ \frac{\Omega \vdash a :^{\ell} A \qquad \ell \leq C}{\Omega \Vdash a :^{\ell} A} \qquad \textsc{CT-Top}\ \frac{C \wedge \Omega \vdash a :^{C} A \qquad C < \ell}{\Omega \Vdash a :^{\ell} A}$$

Fig. 5. Dependent type system with compile-time irrelevance (core rules)

The core typing rules of DDC appear in Figure 5. Compared to DDC$^\top$, this type system maintains the invariant that for any $\Omega \vdash a :^{\ell} A$, we have $\ell \leq C$. To ensure that this is the case, rule T-Type and rule T-Var include this precondition. This restriction means that we cannot really derive any term at $\top$ in DDC. We can get around this restriction by deriving $C \wedge \Omega \vdash a :^{C} A$ in place of $\Omega \vdash a :^{\top} A$.

Wherever DDC$^\top$ uses $\top$ as the observer level on a typing judgment, DDC uses truncation and level C instead. If DDC$^\top$ uses some grade other than $\top$ as the observer level, DDC leaves the derivation as such. So a DDC$^\top$ judgment

$\Omega \vdash a :^{\ell} A$ is replaced with a *truncated-at-top judgment*, $\Omega \Vdash a :^{\ell} A$ which can be read as: if $\ell = \top$, use the truncated version $C \wedge \Omega \vdash a :^{C} A$; otherwise use the normal version $\Omega \vdash a :^{\ell} A$, as we see in Figure 5. In the typing rules, uses of this new judgment have been highlighted in gray to emphasize the modification with respect to DDC$^{\top}$.

5.3 Π-types

Rule T-Pi is unchanged. The lambda rule T-AbsC now checks the type at C after truncating the variables in the context to C. The application rule T-AppC checks the argument using the truncated-at-top judgment. Note that if $\ell_0 = \top$, the term a can depend upon any variable in Ω. Such a dependence is allowed since information can always flow from relevant to irrelevant contexts.

To see how irrelevance works in this system, let's consider the definition and use of the polymorphic identity function.

```
id : Π x:⊤Type. x -> x
id = λ⊤x. λ y. y
```

In DDC$^{\top}$, the type `Π x:⊤Type. x -> x` is checked at $\top$. However, here it must be checked at level C, which requires the premise `x:`^{C}`Type ⊢ x -> x :`^{C} `Type`. Note that if we used the same grade for the bound variable x in rule T-Pi and rule T-AbsC, we would have been in trouble because variable `x` is compile-time relevant while we check the type, even though it is irrelevant in the term.[15]

Finally, observe that rule T-ConvC uses the definitional equality at C instead of $\top$ and that the new type is checked after truncation.

5.4 Σ-types

$$\text{T-WPairC}\quad \frac{\Omega \Vdash a :^{\ell_0 \vee \ell} A \qquad \Omega \vdash b :^{\ell} B\{a/x\} \qquad \Omega \Vdash \Sigma x{:}^{\ell_0} A.B :^{\top} s}{\Omega \vdash (a^{\ell_0}, b) :^{\ell} \Sigma x{:}^{\ell_0} A.B}$$

$$\text{T-LetPairC}\quad \frac{\Omega \vdash a :^{\ell} \Sigma x{:}^{\ell_0} A.B \qquad \Omega, x :^{\ell_0 \vee \ell} A, y :^{\ell} B \vdash c :^{\ell} C\{(x^{\ell_0}, y)/z\} \qquad \Omega, z :^{\top} (\Sigma x{:}^{\ell_0} A.B) \Vdash C :^{\top} s}{\Omega \vdash \mathbf{let}\ (x^{\ell_0}, y)\ =\ a\ \mathbf{in}\ c :^{\ell} C\{a/z\}}$$

We also need to modify the typing rules for Σ types accordingly. In particular, when we create a pair, we check the first component using the truncated-at-top judgment. This is akin to how we check the argument in rule T-AppC. Note that if $\ell_0 = \top$, the first component a is compile-time irrelevant. In such a situation, we cannot type-check the second projection since it requires the first projection, as we see in the derived[16] projection rules below. So pairs having type $\Sigma x{:}^{\top} A.B$

[15] This is why we fuse the graded modality with the dependent types. If they were separated, and we had to bind here, it would be a problem since a dependent function and its type have different restrictions vis-à-vis the bound variable.

[16] `strong_exists.v:T_wproj1,T_wproj2`

can only be eliminated via pattern matching if B mentions x. However, pairs having type $\Sigma x{:}^C A.B$ can be eliminated via projections.

For example, for an output of the `filter` function, `ys` $:\Sigma$`m`$:^C$`Nat. Vec m Bool`, we have π_1 `ys` $:^C$ `Nat` and π_2 `ys : Vec (`π_1 `ys) Bool`. Note that (π_1 `ys`) is visible at C and is used in the type of (π_2 `ys`). We can substitute (π_1 `ys`) for `m` in (`Vec m Bool`) because `m` $:^C$`Nat` $\vdash$ `Vec m Bool` $:^C$ `Type`. However, (π_1 `ys`) cannot be used at $\bot$, so it will be erasable then.

$$\text{T-Proj1C}\quad \frac{\Omega \vdash a :^{\ell} \Sigma x{:}^{\ell_0} A.B \qquad \ell_0 \leq \ell}{\Omega \vdash \pi_1^{\ell_0}\ a :^{\ell} A} \qquad \text{T-Proj2C}\quad \frac{\Omega \vdash a :^{\ell} \Sigma x{:}^{\ell_0} A.B \qquad \ell_0 \leq C}{\Omega \vdash \pi_2^{\ell_0}\, a :^{\ell} B\{\pi_1^{\ell_0}\ a/x\}}$$

5.5 Non-interference

DDC satisfies an analogous noninterference theorem to the one presented for SDC, using suitable definitions for the *grading* relation, written $\Phi \vdash a : \ell$, and *indexed indistiguishability*, written $\Phi \vdash b_1 \sim_\ell b_2$. The complete definition of these judgements appears in the extended version of this paper [12].

Lemma 11 (Typing implies grading[17]). *If* $\Omega \vdash\ a :^{\ell} A$ *then* $|\Omega| \vdash a : \ell$.

Lemma 12 (Equivalence[18]). *Indexed indistinguishability at* ℓ *is an equivalence relation on well-graded terms at* ℓ.

Lemma 13 (Indistinguishability under substitution[19]). *If* $\Phi, x : \ell \vdash b_1 \sim_k b_2$ *and* $\Phi \vdash^{\ell}_{k} a_1 \sim a_2$ *then* $\Phi \vdash b_1\{a_1/x\} \sim_k b_2\{a_2/x\}$.

Theorem 8 (Non-interference for DDC[20]). *If* $\Phi \vdash a_1 \sim_k a_1'$ *and* $a_1 \rightsquigarrow a_2$ *then there exists some* a_2' *such that* $a_1' \rightsquigarrow a_2'$ *and* $\Phi \vdash a_2 \sim_k a_2'$.

5.6 Consistency of Equality

The equality relation of DDC incorporates compile-time irrelevance. To show that the type system is sound, we need to show that the equality relation is consistent. Consistency of definitional equality means that there is no derivation that equates two types having different head forms. For example, it should not equate **Nat** with **Unit**.

Note that if $\top$ inputs can interfere with C outputs, the equality relation cannot be consistent. To see why, let $x :^{\top} A \vdash\ b :^C$ **Bool** and for $a_1, a_2 : A$, let the terms $b\{a_1/x\}$ and $b\{a_2/x\}$ reduce to **True** and **False** respectively. Now, $(\lambda^{\top} x.\mathbf{if}\ b\ \mathbf{then\ Nat\ else\ Unit})\ {a_1}^{\top} \equiv_C (\lambda^{\top} x.\mathbf{if}\ b\ \mathbf{then\ Nat\ else\ Unit})\ {a_2}^{\top}$. But then, by β-equivalence **Nat** $\equiv_C$ **Unit**.

To prove consistency, we construct a standard parallel reduction relation and show that this relation is confluent. Thereafter, we prove that if two terms

[17] `typing.v:Typing_Grade` [18] `geq.v:GEq_refl,GEq_symmetry,GEq_trans`
[19] `subst.v:CEq_GEq_equality_substitution` [20] `geq.v:CEq_GEq_respects_Step`

are definitionally equal at ℓ, then they are joinable at ℓ, meaning they reduce, through parallel reduction, to two terms that are indistinguishable at ℓ. Next, we show that joinability at ℓ implies consistency. Therefore, we conclude that for any ℓ, the equality relation at ℓ is consistent. This implies that the equality relation at C, that ignores sub-terms marked with $\top$, is sound. Hence, DDC tracks compile-time irrelevance correctly. Note that DDC can track run-time irrelevance the same way as DDC$^\top$.

We formally state consistency in terms of *head forms*, i.e. syntactic forms that correspond to types such as sorts s, **Unit**, $\Pi x{:}^\ell A.B$, etc.

Theorem 9 (Consistency[21]). *If $\Phi \vdash a \equiv_\ell b$, and a and b both are head forms, then they have the same head form.*

5.7 Soundness theorem

DDC is type sound and we have checked this and other results using the Coq proof assistant. Below, we give an overview of the important lemmas in this development.

The properties below are stated for DDC, but they also apply to DDC$^\top$ since DDC degenerates to DDC$^\top$ whenever $C = \top$. First, we list the properties related to grading that hold for all judgments: indexed indistinguishability, definitional equality, and typing. (We only state the lemmas for typing, their counterparts are analogous.) These lemmas are similar to their simply-typed counterparts in Section 3.2.

Lemma 14 (Narrowing[22]). *If $\Omega \vdash a :^\ell A$ and $\Omega' \leq \Omega$, then $\Omega' \vdash a :^\ell A$*

Lemma 15 (Weakening[23]). *If $\Omega_1, \Omega_2 \vdash a :^\ell A$ then $\Omega_1, \Omega, \Omega_2 \vdash a :^\ell A$.*

Lemma 16 (Restricted upgrading[24]). *If $\Omega_1, x{:}^{\ell_0} A, \Omega_2 \vdash b :^\ell B$ and $\ell_1 \leq \ell$ then $\Omega_1, x{:}^{\ell_0 \vee \ell_1} A, \Omega_2 \vdash b :^\ell B$.*

Next, we list some properties that are specific to the typing judgment. For any typing judgment in DDC, the observer grade ℓ is at most C. Further, the observer grade of any judgment can be raised up to C.

Lemma 17 (Bounded by C[25]). *If $\Omega \vdash a :^\ell A$ then $\ell \leq C$.*

Lemma 18 (Subsumption[26]). *If $\Omega \vdash a :^\ell A$ and $\ell \leq k$ and $k \leq C$ then $\Omega \vdash a :^k A$*

Note that we don't require contexts to be well-formed in the typing judgment; we add context well-formedness constraints, as required, to our lemmas. The following lemmas are true for well-formed contexts. A context Ω is well-formed, expressed as $\vdash \Omega$, iff for any assumption $x :^\ell A$ in Ω, we have $\Omega' \Vdash A :^\top s$, where Ω' is the prefix of Ω that appears before the assumption.

[21] `consist.v:DefEq_Consistent` [22] `narrowing.v:Typing_narrowing`
[23] `weakening.v:Typing_weakening` [24] `pumping.v:Typing_pumping`
[25] `pumping.v:Typing_leq_C` [26] `typing.v:Typing_subsumption`

Lemma 19 (Substitution[27]). *If* $\Omega_1, x :^{\ell_0} A, \Omega_2 \vdash b :^{\ell} B$ *and* $\vdash \Omega_1$ *and* $\Omega_1 \Vdash a :^{\ell_0} A$ *then* $\Omega_1, \Omega_2\{a/x\} \vdash b\{a/x\} :^{\ell} B\{a/x\}$

Next, if a term is well-typed in our system, the type itself is also well-typed.

Lemma 20 (Regularity[28]). *If* $\Omega \vdash a :^{\ell} A$ *and* $\vdash \Omega$ *then* $\Omega \Vdash A :^{\top} s$.

Finally, we have the two main lemmas proving type soundness.

Lemma 21 (Preservation[29]). *If* $\Omega \vdash a :^{\ell} A$ *and* $\vdash \Omega$ *and* $a \rightsquigarrow a'$, *then* $\Omega \vdash a' :^{\ell} A$.

Lemma 22 (Progress[30]). *If* $\varnothing \vdash a :^{\ell} A$ *then either* a *is a value or there exists some* a' *such that* $a \rightsquigarrow a'$.

Hence, DDC is type sound. We have seen earlier that it tracks run-time and compile-time irrelevance correctly.

DDC is parameterized by a generic pure type system and a generic lattice. When the parameterizing pure type system is strongly normalizing, such as the Calculus of Constructions, type-checking is decidable. In the next section, we provide a demonstration.

6 Type Checking

As a pure type system, not all instances of DDC admit decidable type checking. For example, in the presence of the type:type axiom, the system includes non-terminating computations via Girard's paradox. As as a result, we cannot decide equality in that system, so type checking will be undecidable. However, if the sorts, axioms and rules are chosen such that the language is strongly normalizing, then we can define a decidable type checking algorithm. This algorithm is standard, but relies on a decision procedure for the equality judgement.

Our consistency proof, described in Section 5.6, gives us a start. This proof uses an auxiliary binary relation called *joinability*, which holds when two terms can use multiple steps of parallel reduction to reach two simpler terms that differ only in their unobservable components. Joinability and definitional equality induce the same relation on DDC terms. We can show that two DDC terms are definitionally equal if and only if they are joinable[31], which means that a decision procedure based on joinability will be sound and complete for DDC's labeled definition of equivalence.

Therefore, the decidability of type checking reduces to showing strong normalization. If we select the sorts, axioms and rules of DDC to match those of the Calculus of Constructions [5], we believe that this result holds, but leave a direct proof for future work. However, by translating this instance of DDC to ICC*, we can show that a sublanguage of this instance is strongly normalizing.

[27] typing.v:Typing_substitution_CTyping [28] typing.v:Typing_regularity
[29] typing.v:Typing_preservation [30] progress.v:Typing_progress
[31] consist.v:DefEq_Joins,Joins_DefEq

ICC* [6], is a version of the Implicit Calculus of Constructions with annotations that support decidable type checking, but because it includes only (relevant and irrelevant) Π-types, so we must restrict our attention to the corresponding fragment of DDC.

We define the following translation, written $\widetilde{\cdot}$, that converts DDC terms to ICC* terms. The key parts of this translation map arguments labeled C and below to relevant arguments, and those labeled greater than C, such as $\top$, to irrelevant arguments.[32]

$$\widetilde{x} = x \qquad \widetilde{s} = s \qquad \widetilde{\Pi x :^{\ell} A.B} = \begin{cases} \Pi(x : \widetilde{A}).\widetilde{B} & \text{if } \ell \leq C \\ \Pi[x : \widetilde{A}].\widetilde{B} & \text{otherwise} \end{cases}$$

$$\widetilde{\lambda x :^{\ell} A.b} = \begin{cases} \lambda(x : \widetilde{A}).\widetilde{b} & \text{if } \ell \leq C \\ \lambda[x : \widetilde{A}].\widetilde{b} & \text{otherwise} \end{cases} \qquad \widetilde{b\ a^{\ell}} = \begin{cases} \widetilde{b}\ (\widetilde{a}) & \text{if } \ell \leq C \\ \widetilde{b}\ [\widetilde{a}] & \text{otherwise} \end{cases}$$

Note that ICC* compares terms for equality after an erasure operation, written $\cdot^*$, that removes all irrelevant arguments. Now, we can show that the above translation preserves definitional equality and typing. Here, $\widetilde{\Omega}$ denotes Ω with the labels at the variable bindings omitted.

Lemma 23 (Translation preservation). *If* $\Phi \vdash A \equiv_C B$, *then* $\widetilde{A}^* \cong_{\beta\eta} \widetilde{B}^*$. *If* $\Omega \vdash a :^{\ell} A$, *then* $\widetilde{\Omega} \vdash \widetilde{a} : \widetilde{A}$.

Next, observe that because β-reductions are preserved by the translation, any parallel reduction in DDC between terms a and b at level C, where $a \neq b$, would correspond to a sequence of reduction steps $\widetilde{a} \to^{+}_{\beta_{ie}} \widetilde{b}$ in ICC*. That means that an infinite sequence of parallel reductions a_0, a_1, ..., where each term differs from the previous, corresponds to an infinite sequence of reductions $\widetilde{a_0}$, $\widetilde{a_1}$ ... in ICC*. Therefore, as all well-typed ICC* terms are strongly normalizing, we can conclude that this is so for this instance of DDC.

Non-terminating instances of DDC. For pure type systems that are not strongly normalizing, such as the type:type language, there is an alternative approach to developing a calculus with decidable type checking, following Weirich et al. [35]. The key idea is to develop an annotated version of DDC that book-keeps additional information from typing and equality derivations. In such an annotated version, the conversion rule would include an explicit coercion annotation that witnesses the equality between the concerned types, thus avoiding the need for normalization.

[32] The syntax of ICC* uses parentheses to indicate usual (relevant) arguments and square brackets to indicate arguments that are irrelevant at both run time and compile time.

7 Discussions and Related Work

7.1 Irrelevance in Dependent Type Theories

Overall, compile-time and run-time irrelevance is a well-studied topic in the design of dependent type systems. In some systems, the focus is only on support for run-time irrelevance: see [18,4,8,19,20,32]. In other systems, the focus is on compile-time irrelevance: see [27,3]. Some systems support both, but require them to overlap, such as [6,21,35,24]. The system of Mooon et al. [23] does not require them to overlap but their type system does not make use of compile-time irrelevance in the conversion rule.

To compare, system $\mathrm{DDC}^{\top}$, presented here, can support run-time irrelevance only and is similar to the core language of Tejiščák [32]. However, note that $\mathrm{DDC}^{\top}$ can track dependencies in general while the system in [32] tracks run-time irrelevance alone. DDC, on the other hand, is the only system that we are aware of that tracks run-time and compile-time irrelevance separately and makes use of the latter in the conversion rule. Further, DDC tracks these irrelevances in the presence of strong Σ-types with erasable first components, something which, to the best of our knowledge, no prior work has been able to.

Prior work has identified the difficulty in handling strong Σ-types with erasable first components in a setting that tracks compile-time irrelevance. Abel and Scherer [3] point out that strong irrelevant Σ-types make their theory inconsistent. Similarly, $\mathrm{EPTS}^{\bullet}$ [21] cannot define the projections for pairs having such Σ-types. The reason behind this is that $\mathrm{EPTS}^{\bullet}$ is hard-wired to work with a two-element lattice which identifies compile-time and run-time irrelevance. As such, projections from such pairs lead to type unsoundness. For example, considering the first components to be run-time irrelevant, the pairs $(\mathbf{Int}, \mathbf{unit})$ and $(\mathbf{Bool}, \mathbf{unit})$ are run-time equivalent. Since $\mathrm{EPTS}^{\bullet}$ identifies run-time and compile-time irrelevance, these pairs are also compile-time equivalent. Then, taking the first projections of these pairs, one ends up with **Int** and **Bool** being compile-time equivalent. We resolve this problem by distinguishing between run-time and compile-time irrelevance, thus requiring a lattice with three elements.

Next, we compare our work with existing literature with respect to the equality relation. We analyze compile-time irrelevance to enable the equality relation to ignore unnecessary sub-terms. However, since our equality relation is untyped, we cannot include type-dependent rules in our system, such as η-equivalence for the `Unit` type. Several prior works on irrelevance [19,6,21,32] use an untyped equality relation. However, some prior work, such as [27,3], do consider compile-time irrelevance in the context of typed-directed equality. But such systems require irrelevant arguments to functions appear only irrelevantly in the codomain type of the function, thus ruling out several examples including the polymorphic identity function.

7.2 Quantitative Type Systems

Our work is closely related to quantitative type systems [26,15,9,18,4,25,2,10,23]. Such systems provide a fine-grained accounting of coeffects, viewed as resources,

for example, variable usage, linearity, liveness, etc. A typical judgment from a quantitative type system [10] may look like:

$$x :^1 \mathbf{Bool}, y :^1 \mathbf{Int}, z :^0 \mathbf{Bool} \vdash \mathbf{if}\, x\, \mathbf{then}\, y + 1\, \mathbf{else}\, y - 1 \;:^1\; \mathbf{Int}$$

The variable x is used once in the condition, the variable y is used once in each of the branches while the variable z is not used at all. As such, they are marked with these quantities in the context.

This form of judgment is very similar to our typing judgments with quantities appearing in place of levels. However, there is a crucial difference: to the right of the turnstile, while any level may appear in our judgments, only the quantity 1 can appear in typing judgments of quantitative systems. A quantitative system that allows an arbitrary quantity to the right of the turnstile is not closed under substitution [18,4]. As such, quantitative systems are tied to a fixed reference while our systems can view programs from different reference levels. This difference in form results from the difference in the purposes the two kinds of systems serve: quantitative systems count while our systems compare. Counting requires a fixed standard or reference whereas comparison does not. Applications that require counting, like linearity tracking, are handled well by quantitative systems while applications that require comparison, like ensuring secure information flow, are handled well by systems of our kind.

From a type-theoretic standpoint, in general, quantitative systems cannot eliminate pairs through projections. This is so because there is no general way to split the resources of the context that type-checks a pair. Eliminating pairs through projections is straightforward in our systems because the grade on the typing judgment can control where the projections are visible.

7.3 Dependency Analysis and Dependent Type Theory

Dependency analysis and dependent type theories have come together in some existing work.

Like our system, Prost [28] extends the λ-cube so that it may track dependencies. However, unlike our system, this work uses sorts to track dependencies. It is inspired by the distinction between sorts in the Calculus of Constructions where computationally relevant and irrelevant terms live in sorts `Set` and `Prop` respectively. As Mishra-Linger [21] points out, such an approach ties up two distinct language features, sorts and dependency analysis, which can be treated in a more orthogonal manner.

Bernardy and Guilhem's type-theory in color [7] is very related to our work. This type-theory uses colors to erase terms while we use grades. Colors and grades both form a lattice structure and their usage in the respective type systems are quite similar. However, in type-theory in color, internalized parametricity is used to reason about erasure; so it is important that the type-theory be logically consistent. Our work does not rely on the normalizing nature of the theory; we take a direct route to analyzing erasure.

Like our work, Lourenço and Caires [17] track information flow in a dependent type system. But Lourenço and Caires [17] focus on more imperative features,

like modeling of state while we focus on irrelevance. A distinguishing feature of their system is that they allow security labels to depend upon terms, something that we don't attempt here.

8 Conclusion

We started with the aim of designing a dependent calculus that can analyze dependencies in general, and run-time and compile-time irrelevance in particular. Towards this end, we designed a simple dependency calculus, SDC, and then extended it to two dependent calculi, $\text{DDC}^{\top}$ and DDC. $\text{DDC}^{\top}$ can track run-time irrelevance while DDC can track both run-time and compile-time irrelevance along with other dependencies.

In future, we would like to explore how irrelevance interacts with other dependencies. We also want to explore whether our systems can be integrated with existing graded type systems, especially quantitative type systems. Yet another interesting direction for research is that how they compare with graded effect systems.

Our work lies in the intersection of dependency analysis and irrelevance tracking in dependent type systems. Both these areas have rich literature of their own. We hope that the connections established in this paper will be mutually beneficial and help in the future exploration of dependencies and irrelevance in dependent type systems.

9 Acknowledgments

The first two authors were supported by the National Science Foundation under Grant Nos. 1703835 and 1521539. The second author was supported by the National Science Foundation under Grant No. 2104535.

References

1. Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.: A core calculus of dependency. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 147–160. POPL '99, Association for Computing Machinery, New York, NY, USA (1999). https://doi.org/10.1145/292540.292555
2. Abel, A., Bernardy, J.P.: A unified view of modalities in type systems. Proc. ACM Program. Lang. **4**(ICFP) (Aug 2020). https://doi.org/10.1145/3408972
3. Abel, A., Scherer, G.: On irrelevance and algorithmic equality in predicative type theory. Logical Methods in Computer Science **8**(1) (mar 2012). https://doi.org/10.2168/lmcs-8(1:29)2012
4. Atkey, R.: Syntax and semantics of quantitative type theory. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science. p. 56–65. LICS '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3209108.3209189
5. Barendregt, H.P.: Lambda Calculi with Types, p. 117–309. Oxford University Press, Inc., USA (1993)
6. Barras, B., Bernardo, B.: The implicit calculus of constructions as a programming language with dependent types. In: Amadio, R. (ed.) Foundations of Software Science and Computational Structures. pp. 365–379. FOSSACS 2008, Springer Berlin Heidelberg, Budapest, Hungary (2008)
7. Bernardy, J.P., Guilhem, M.: Type-theory in color. SIGPLAN Not. **48**(9), 61–72 (Sep 2013). https://doi.org/10.1145/2544174.2500577
8. Brady, E.: Idris 2: Quantitative Type Theory in Practice. In: Møller, A., Sridharan, M. (eds.) 35th European Conference on Object-Oriented Programming (ECOOP 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 194, pp. 9:1–9:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.ECOOP.2021.9
9. Brunel, A., Gaboardi, M., Mazza, D., Zdancewic, S.: A core quantitative coeffect calculus. In: Shao, Z. (ed.) Programming Languages and Systems. pp. 351–370. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
10. Choudhury, P., Eades III, H., Eisenberg, R.A., Weirich, S.: A graded dependent type system with a usage-aware semantics. Proc. ACM Program. Lang. **5**(POPL) (Jan 2021). https://doi.org/10.1145/3434331
11. Choudhury, P., Eades III, H., Weirich, S.: Artifact associated with "A Dependent Dependency Calculus" (Jan 2022). https://doi.org/10.5281/zenodo.5903726
12. Choudhury, P., Eades III, H., Weirich, S.: A dependent dependency calculus (extended version) (2022), `https://arxiv.org/abs/2201.11040`
13. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (May 1976). https://doi.org/10.1145/360051.360056
14. Eisenberg, R.A., Duboc, G., Weirich, S., Lee, D.: An existential crisis resolved: Type inference for first-class existential types. Proc. ACM Program. Lang. **5**(ICFP) (Aug 2021), `https://richarde.dev/papers/2021/exists/exists.pdf`
15. Ghica, D.R., Smith, A.I.: Bounded linear types in a resource semiring. In: European Symposium on Programming Languages and Systems. pp. 331–350. Springer (2014)
16. Heintze, N., Riecke, J.G.: The slam calculus: Programming with secrecy and integrity. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 365–377. POPL '98, Association for Computing Machinery, New York, NY, USA (1998). https://doi.org/10.1145/268946.268976

17. Lourenço, L., Caires, L.: Dependent information flow types. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 317–328. POPL '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2676726.2676994
18. McBride, C.: I Got Plenty o' Nuttin', pp. 207–233. Springer International Publishing, Cham (2016)
19. Miquel, A.: The implicit calculus of constructions extending pure type systems with an intersection type binder and subtyping. In: Abramsky, S. (ed.) Typed Lambda Calculi and Applications. pp. 344–359. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
20. Mishra-Linger, N., Sheard, T.: Erasure and polymorphism in pure type systems. In: Proceedings of the Theory and Practice of Software, 11th International Conference on Foundations of Software Science and Computational Structures. p. 350–364. FOSSACS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008)
21. Mishra-Linger, R.N.: Irrelevance, Polymorphism, and Erasure in Type Theory. Ph.D. thesis, Portland State University, Department of Computer Science (2008). https://doi.org/10.15760/etd.2669
22. Moggi, E.: Notions of computation and monads. Information and Computation **93**(1), 55–92 (1991), `https://www.sciencedirect.com/science/article/pii/0890540191900524`, selections from 1989 IEEE Symposium on Logic in Computer Science
23. Moon, B., Eades III, H., Orchard, D.: Graded modal dependent type theory. In: Yoshida, N. (ed.) Programming Languages and Systems. pp. 462–490. Springer International Publishing, Cham (2021)
24. Nuyts, A., Devriese, D.: Degrees of relatedness: A unified framework for parametricity, irrelevance, ad hoc polymorphism, intersections, unions and algebra in dependent type theory. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018. pp. 779–788. ACM (2018). https://doi.org/10.1145/3209108.3209119, `https://doi.org/10.1145/3209108`
25. Orchard, D., Liepelt, V.B., Eades III, H.: Quantitative program reasoning with graded modal types. Proc. ACM Program. Lang. **3**(ICFP) (Jul 2019). https://doi.org/10.1145/3341714
26. Petricek, T., Orchard, D., Mycroft, A.: Coeffects: A calculus of context-dependent computation. In: Proceedings of International Conference on Functional Programming. ICFP 2014 (2014)
27. Pfenning, F.: Intensionality, extensionality, and proof irrelevance in modal type theory. In: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science. pp. 221–. LICS '01, IEEE Computer Society, Washington, DC, USA (2001), `http://dl.acm.org/citation.cfm?id=871816.871845`
28. Prost, F.: A static calculus of dependencies for the λ-cube. In: Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332). pp. 267–276 (2000). https://doi.org/10.1109/LICS.2000.855775
29. Shikuma, N., Igarashi, A.: Proving noninterference by a fully complete translation to the simply typed λ-calculus. In: Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues. p. 301–315. ASIAN'06, Springer-Verlag, Berlin, Heidelberg (2006)
30. Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 355–364. POPL '98, Association for Computing Machinery, New York, NY, USA (1998). https://doi.org/10.1145/268946.268975

31. Sulzmann, M., Chakravarty, M.M.T., Jones, S.P., Donnelly, K.: System f with type equality coercions. In: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation. p. 53–66. TLDI '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1190315.1190324
32. Tejiščák, M.: A dependently typed calculus with pattern matching and erasure inference. Proc. ACM Program. Lang. **4**(ICFP) (Aug 2020). https://doi.org/10.1145/3408973
33. Thiemann, P.: A unified framework for binding-time analysis. In: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development. p. 742–756. TAPSOFT '97, Springer-Verlag, Berlin, Heidelberg (1997)
34. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages **3** (1995)
35. Weirich, S., Voizard, A., de Amorim, P.H.A., Eisenberg, R.A.: A specification for dependent types in Haskell. Proc. ACM Program. Lang. **1**(ICFP), 31:1–31:29 (Aug 2017). https://doi.org/10.1145/3110275