A Dependent Dependency Calculus

ANONYMOUS AUTHOR(S)

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 tracking has shown many practical benefits to language design: its results can help users and compilers protect sensitive information, trace data provenance, and produce faster code, among other applications. In this work, we present a Dependent Dependency Calculus (DDC), which extends this general and powerful idea to the setting of a dependently typed language. We use this mechanism to support both run-time and compile-time irrelevance in a uniform setting, enabling faster type-checking and program execution.

ACM Reference Format:
Anonymous Author(s). 2021. A Dependent Dependency Calculus. 1, 1 (July 2021), 28 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 DEPENDENCY-TRACKING IN TYPE SYSTEMS

Consider this typing judgement from a type system that has been augmented with *dependency tracking*.

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

In this judgement, the expression has been marked as a medium-security result. It has several inputs, x, y and z, each of which 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 y is not used in the expression. However, if we replace z with y in the conditional, then the type checker would reject this code. Even though the high-security input would not be returned directly, the medium-security result would still depend on this boolean.

Dependency tracking, as above, is an *expressive* addition to typed programming languages. This mechanism allows typed languages to protect sensitive information [Smith and Volpano 1998], trace data provenance and integrity [Zdancewic and Myers 2001], support run-time code generation [Thiemann 1997], erase types during compilation [Brady et al. 2003], regulate zero-cost coercions [Weirich et al. 2019] and reason efficiently using proof irrelevance [Pfenning 2001]. Abadi et al. [1999] unified some of these applications in the Dependency Core Calculus (DCC) which has served as a foundational framework for static analysis of information flow in programming languages.

What makes dependency tracking powerful is the parametrization of the type system by a *generic* lattice of dependency levels. Dependency tracking, at its core, is about guaranteeing non-interference—that if $\neg(k \leq \ell)$ in the lattice, then inputs marked with k do not affect results at level ℓ. The design of the type system and the non-interference result are independent of any specific lattice or application.

XXXX-XXXX/2021/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

At the same time, dependency tracking is *simple*. In essence, the type system need only ensure that a variable is used at a level at least as secure as its own assigned one. In the above example, z, a medium-security boolean, may be used in either in a medium-security computation or in a high-security computation, but never in a low-security one. The key additional component required in tracking dependency is a graded monad, which allows high-security data to be safely boxed and then released to a low-security world [Abadi et al. 1999; Shikuma and Igarashi 2006].

However, even though many typed languages have included dependency tracking in some form, this feature has seen relatively little attention in the context of *dependently-typed* programming languages and type systems. The reason for this omission is that it is possible to encode dependency tracking using type abstraction and derive non-interference as a corollary to parametricity [Algehed and Bernardy 2019; Bowman and Ahmed 2015; Shikuma and Igarashi 2006; Tse and Zdancewic 2004]. Using these ideas, dependency tracking for secure information flow has been embedded into the Haskell language using arrows [Li and Zdancewic 2010] and monads [Russo et al. 2008]. These embeddings are also possible in richer contexts, such as Coq, Agda and Idris.

However, in this work we take a more direct approach and augment a dependently-typed language with a *primitive* notion of dependency tracking. We call this language DDC, for Dependent Dependency Calculus. We find that this direct approach has several benefits.

First, using a direct approach means that our non-interference result is not derived from parametricity. Instead, non-interference can be proved directly via simple syntactic techniques. As a result, it is available in contexts where parametricity is difficult to prove or does not hold.

Second, by directly incorporating dependency tracking, the type system itself can take advantage of this mechanism. In particular, dependency tracking directly supports run-time irrelevance, or the identification of parts of expressions that are only needed for type checking and can be erased prior to execution. It can also be adapted to support compile-time irrelevance, or the identification of parts of expressions that are not required when checking equivalence statically.

In this work, we make the following contributions:

- We first isolate the essence of dependency tracking in the context of a simply-typed language. We call this system SDC, for Simple Dependency Calculus. We show that SDC is sound and at least as expressive as the terminating fragment of DCC. (Section 3)
- We define a level-indexed version of equality that obscures parts of expressions that are unobservable at a particular level. We show that this equality is preseved by the operational semantics of the language, giving us a syntactic proof of non-interference. (Section 3.2)
- We adapt the ideas of SDC into two versions of DDC, the general dependent dependency calculus. The first, and simpler, version (DDC$^{\top}$) uses dependency levels to track run-time irrelevance. Using a lattice of at least two security levels, this system can use the highest level of the lattice ($\top$) to track specificational arguments that are not needed at run time. (Section 4.3)
- The second, and full, version of DDC supports compile-time irrelevance. Using a lattice with at least three points, we show how to distinguish between expressions that are relevant at run time (level $\bot$), erasable but relevant at compile time (level C), and irrelevant at both phases (level $\top$). To take advantage of compile-time irrelevance in type-checking, we use level-indexed equality as a basis for conversion, relating types at the coarser equivalence level C rather than $\top$. (Section 5)
- We show that SDC can be considered as a graded effect system by showing that its categorical model in classified sets has the structure of a category of graded algebras. The latter category is abstracted into a new multicategory that captures the base structure of fully graded type systems, and is called a grade-indexed multicategory. (Section 6)

- We provide a detailed comparison of DDC against existing mechanisms for compile-time and run-time irrelevance analysis, including those based on quantitative type theories and other approaches. DDC is the only system that combines separate quantifiers for compile-time and run-time irrelevance in the same framework. In particular, we observe that the distinctions made by DDC are necessary for a proper treatment of erasable projections from strong Σ types. (Section 7)
- We have formalized and mechanized the type soundness proof of full DDC using the Coq proof assistant. The mechanization gives us confidence about the technical correctness of our development. It can also be used for experimentation and extension. Our proof scripts are available to reviwers as anonymized supplementary material and we plan to make this development publicly available. This anonymous supplementary material also includes the full specification of the system.

2 BACKGROUND

In this section, we briefly sketch out the background. We first motivate the generic lattice structure of dependency. Thereafter, we delve into run-time and compile-time irrelevance in dependently-typed languages.

2.1 Lattice Structure of Dependency

Let us consider a world of objects sharing information with one another. For any given ordered pair (O_1, O_2) of objects, we may either allow or block information flow from O_1 to O_2. For example, if O_1 is general public and O_2 is secret services agent, we allow information flow from O_1 to O_2 but we block flow along the other way. Such flow constraints induce an ordering $(\leq)$ on the objects sharing information, with $O_1 \leq O_2$ meaning information can flow from O_1 to O_2.

The ordering is generally reflexive and transitive. We expect information to flow from object O to itself. We also expect that, if information can flow from O_1 to O_2 and also from O_2 to O_3, then it can flow from O_1 to O_3. These properties give us a pre-order. Now, if information can flow from O_1 to O_2 and also from O_2 to O_1, it makes sense to merge the two objects together into a single level. A set of objects between which information may freely flow is called a level. With this grouping, we arrive at a partial ordering of levels, reflecting an ordering on the nature of information itself. There are many examples of partially-ordered information: a) An order with two levels: Public or low security, L and Private or high security, H with $L \leq H$. b) An order with four levels: General Staff, Marketing Director, Sales Manager and CEO with General Staff at the Bottom and CEO at the top while Marketing Director and Sales Manager are in between and incomparable to one another.

Now, given any finite partial ordering of information, we can convert it into a unique lattice by adding the missing 'joins' and 'meets'. This idea of a lattice model of information flow, introduced in Denning [1976], has been used extensively in the design of secure information flow systems. Many programming languages [Denning and Denning 1977; Heintze and Riecke 1998; Smith and Volpano 1998; Thiemann 1997, etc.] use the lattice model to statically enforce information flow constraints in programs. Abadi et al. [1999] unified some of these languages through the Dependency Core Calculus (DCC). Over the years, DCC has served as a foundational framework for static analysis of information flow in programming languages. In this paper, we extend the Dependency Core Calculus to dependent types and use operational methods to reason about information flow properties. To the best of our knowledge, our paper is the first of its kind in this respect. Further, we use dependency to track irrelevance in dependent type systems.

2.2 Run-time and Compile-time irrelevance

Run-time irrelevance (sometimes called *erasure*) and *compile-time irrelevance* are two forms of *dependency* analysis arising in dependent type theories. Tracking these dependencies helps compilers produce leaner code, type-check and run programs faster and helps users reason more easily about programs [Abel and Scherer 2012; Atkey 2018; Barras and Bernardo 2008; McBride 2016; Miquel 2001; Mishra-Linger and Sheard 2008; Nuyts and Devriese 2018; Pfenning 2001; Tejiščák 2020].

In a dependently typed language, not all arguments to a function are needed at run time. For example, consider the following function, written in the Agda programming language, that takes a bounded number and returns the same number but with a larger bound [The Agda Team 2021]. (The `Fin n` type contains exactly `n` values.)

```
inject+ : ∀ n m → (fin : Fin n) → Fin (n + m)
inject+ _ _  zero      = zero
inject+ _ _ (suc fin) = suc (inject+ _ _ fin)
```

So `inject+ 5 9 2 = 2` and `inject+ 10 42 2 = 2`. The first two arguments to this function, `n` and `m`, are only used to specify its type. Though the type of the output depends on them, the value *does not depend* on either of them. Therefore, to save time and space, we can erase both the programs to (`inject+ _ _ 2`) before running them. Since `n` and `m` are not required during run time, we say that they are *run-time irrelevant*. Erasing sub-terms irrelevant to run-time computation makes programs run faster.

However, with dependently typed languages, type-checking also requires us to compute — with type-expressions. For example, `inject+` with all parameters explicit, looks like:

```
inject+ : ∀ n m → (fin : Fin n) → Fin (n + m)
inject+ (suc n)   m   (zero {n})      = zero {n + m}
inject+ (suc n)   m   (suc {n} fin)  = suc {n + m} (inject+ n m fin)
```

Note that in the above definition, in both the lines, the result should have type (`Fin (suc n + m)`) but both the outputs have type (`Fin (suc (n + m))`). We need to reduce (`suc (n + m)`) to (`suc n + m`) to know that the types are equal. Here, we need just a single step to check type equality but type reductions may get quite complex at times. Below, we show an example that demonstrates how analysing dependency helps us reduce complexity and make type-checking faster.

Consider the following contrived example:

```
phantom : ∀ (a : Set) → Set
phantom a = if (fib 28 == 317810) then Nat else Bool
```

The function `phantom` *does not depend* on its argument `a`. It just returns either `Nat` or `Bool` based on a conditional. The conditional checks whether the Fibonacci number F_{28} equals 317810. Next, we use this function to specify a type:

```
conv : ∀ (a b : Set) → phantom a -> phantom b
conv _ _ x = x
```

The function `conv` is essentially an identity. We can immediately see that `conv` should type-check. Since `phantom` ignores its argument, `phantom a = phantom b`. But, if we have no way of communicating this information to the type-checker, it may actually reduce `phantom a` and `phantom b` to find out whether they are the same type. With a computationally-intensive conditional, such a reduction may take a really long time. In fact, on a modern laptop, Agda takes about a minute and a half to type-check `conv`. But we could do this in no time just by informing the compiler that it can ignore the argument to `phantom` while checking for equality between types. Ignoring such arguments makes type-checking faster. Arguments that can be ignored while checking type equality at compile time are called *compile-time irrelevant*.

So we see that tracking both run-time irrelevance and compile-time irrelevance are useful. Generally, compile-time irrelevance implies run-time irrelevance. But the converse is not always true. A sub-term may be run-time irrelevant but needed during equality checking. Consider the following definition of complex numbers:

```
Complex : Σ[ α ∈ Set ]
              ((ℂ : Real × Real → α) × (_+_ : α × α → α) × (|_| : α → Real))
Complex = ( Real × Real , (id , sum , mod)) where

              sum : (Real × Real) → (Real × Real) → (Real × Real)
              sum (m , n) (p , q) = ((m + n) , (p + q))

              mod : Real × Real → Real
              mod (m , n) = sqrt (m^2 + n^2)
```

`Complex` has a Σ-type. Its first component gives the underlying representation while the second component defines some functions on complex numbers: we can construct (ℂ) a complex number from a pair of real numbers, we can add (_+_) two complex numbers and we can find the absolute value (|_|) of a complex number. Here, a complex number is represented in cartesian form as a pair of real numbers. But we don't want any user program to depend on this information. This *non-dependence* gives the implementer the flexibility to unnoticeably change the representation to a different, for example, polar form. In other words, changing the representation to a polar form would not change the run-time behavior of any user program. So we may say that the first component of `Complex` is run-time irrelevant. It is not compile-time irrelevant though: we cannot ignore it while deriving type equality.

We use our dependency calculus to analyse irrelevance. The constraint we must abide by is that information should never flow from irrelevant to relevant contexts. Our calculus ensures that this flow constraint is satisfied. This means, once we show our calculus is correct, we know that a) erasing any irrelevant term at run time is safe and b) ignoring compile-time irrelevant terms while checking for type equality is sound.

With the background set up, let us move to the calculi.

3 A SIMPLE DEPENDENCY-TRACKING CALCULUS

(Grammar)

labels	ℓ, k	$::=$	$\bot \mid \top \mid k \wedge \ell \mid k \vee \ell \mid \ldots$	
types	A, B	$::=$	$\mathbf{Unit} \mid A \to B \mid A \times B \mid A + B \mid T^{\ell}\ A$	
terms	a, b	$::=$	$x \mid \lambda x{:}A.a \mid a\ b$	*variables and functions*
		$\mid$	$\mathbf{unit} \mid (a, b) \mid \pi_1\ a \mid \pi_2\ a$	*unit and products*
		$\mid$	$\mathbf{inj}_1\ a \mid \mathbf{inj}_2\ a \mid \mathbf{case}\ a\ \mathbf{of}\ b_1; b_2$	*sums*
		$\mid$	$\eta^{\ell}\ a \mid \mathbf{bind}^{\ell}\ x = a\ \mathbf{in}\ b$	*graded modality*
contexts	Ω	$::=$	$\varnothing \mid \Omega, x :^{\ell} A$	

Fig. 1. Simple Dependency Calculus

$\boxed{\Omega \vdash a :^{\ell} A}$ *(Simple Dependency-Tracking Calculus)*

$$\textsc{SDC-Var}\quad \frac{\ell_0 \leq \ell \qquad x :^{\ell_0} A \in \Omega}{\Omega \vdash x :^{\ell} A}
\qquad
\textsc{SDC-Unit}\quad \frac{}{\Omega \vdash \mathbf{unit} :^{\ell} \mathbf{Unit}}$$

$$\textsc{SDC-Abs}\quad \frac{\Omega, x :^{\ell} A \vdash b :^{\ell} B}{\Omega \vdash \lambda x{:}A.b :^{\ell} A \to B}
\qquad
\textsc{SDC-App}\quad \frac{\Omega \vdash b :^{\ell} A \to B \qquad \Omega \vdash a :^{\ell} A}{\Omega \vdash b\ a :^{\ell} B}$$

$$\textsc{SDC-Pair}\quad \frac{\Omega \vdash a_1 :^{\ell} A_1 \qquad \Omega \vdash a_2 :^{\ell} A_2}{\Omega \vdash (a_1, a_2) :^{\ell} A_1 \times A_2}
\qquad
\textsc{SDC-Proj1}\quad \frac{\Omega \vdash a :^{\ell} A_1 \times A_2}{\Omega \vdash \pi_1\ a :^{\ell} A_1}
\qquad
\textsc{SDC-Proj2}\quad \frac{\Omega \vdash a :^{\ell} A_1 \times A_2}{\Omega \vdash \pi_2\ a :^{\ell} A_2}$$

$$\textsc{SDC-Inj1}\quad \frac{\Omega \vdash a_1 :^{\ell} A_1}{\Omega \vdash \mathbf{inj}_1\ a_1 :^{\ell} A_1 + A_2}
\qquad
\textsc{SDC-Inj2}\quad \frac{\Omega \vdash a_2 :^{\ell} A_2}{\Omega \vdash \mathbf{inj}_2\ a_2 :^{\ell} A_1 + A_2}$$

$$\textsc{SDC-Case}\quad \frac{\Omega \vdash a :^{\ell} A_1 + A_2 \qquad \Omega, x :^{\ell} A_1 \vdash b_1 :^{\ell} B \qquad \Omega, y :^{\ell} A_2 \vdash b_2 :^{\ell} B}{\Omega \vdash \mathbf{case}\ a\ \mathbf{of}\ x \hookrightarrow b_1 \mid y \hookrightarrow b_2 :^{\ell} B}$$

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

Fig. 2. Typing rules for SDC

To explain the essence of dependency tracking, we first design the Simple Dependency Calculus (SDC). This language is parametrized over a lattice of *labels* or *grades*, which can also be thought of as security *levels*.[1] The syntax of this calculus appears in Figure 1 and is an extension of the simply typed λ-calculus with a label-indexed modal type $T^{\ell}\ A$. The calculus itself is also graded, meaning both the term and every variable in the context carries a label or grade. The typing judgement has the form $\Omega \vdash a :^{\ell} A$ which means that "ℓ is allowed to observe a" or that "a is visible at ℓ".

3.1 Type System

The typing rules for SDC appear in Figure 2. Most rules are straightforward and propagate the level of the subterms 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 level of the expression. In other words, any observer at level ℓ is allowed to use a variable at level k, for $k \leq \ell$. 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. The abstraction rule (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 rule SDC-Return says that if a term is $(\ell \vee \ell_0)$-secure, then we can put it in a ℓ_0-secure box and release it at level ℓ. A ℓ_0-secure boxed term can be unboxed only

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

by someone who has security clearance ℓ_0, as we see in rule SDC-Bind.[2] The join operation in rule SDC-Bind ensures that b can depend on a only if b itself is ℓ_0-secure or if $\ell_0 \leq \ell$.

This type system satisfies the following properties related to grades.

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 $\Omega_1 \leq \Omega_2$ iff they are the same modulo the grades and for any x, if $x :^{\ell_1} A \in \Omega_1$ and $x :^{\ell_2} A \in \Omega_2$, then $\ell_1 \leq \ell_2$.

Lemma 3.1 (Narrowing). *If $\Omega' \vdash a :^{\ell} A$ and $\Omega \leq \Omega'$, 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 variables to the level of the judgement.

Lemma 3.2 (Pumping). *If $\Omega_1, x :^{\ell_0} A, \Omega_2 \vdash b :^{\ell} B$ and $\ell_1 \leq \ell$, then $\Omega_1, x :^{\ell_0 \vee \ell_1} A, \Omega_2 \vdash b :^{\ell} B$.*

The pumping lemma is needed to show subsumption, i.e. if a term is visible at some grade, then it is also visible at all higher grades.

Lemma 3.3 (Subsumption). *If $\Omega \vdash a :^{\ell} A$ and $\ell \leq k$, then $\Omega \vdash a :^{k} A$.*

Subsumption is necessary (along with a standard weakening lemma) to show that substitution holds for this language. In the substitution lemma, we must match up the levels of the variable in the context and the expression being substituted in.

Lemma 3.4 (Substitution). *If $\Omega_1, x :^{\ell_0} A, \Omega_2 \vdash b :^{\ell} B$ and $\Omega_1 \vdash a :^{\ell_0} A$, then $\Omega_1, \Omega_2 \vdash b\{a/x\} : B$.*

These lemmas allows us to prove a standard progress and preservation based type soundness result, which we omit here. Type soundness holds with respect to both call-by-value and call-by-name semantics. We now move on to non-interference.

3.2 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 in level-specific views of the world. The values η^H **True** and η^H **False** must appear the same to an L-user while an H-user differentiates between them. To capture this notion of a level-specific view, we design a relation between terms, $\sim_\ell$, called *guarded equality*, and shown in Figure 3.

Informally, $a \sim_\ell b$ means that a and b appear the same to an ℓ-user. For example, η^H **True** $\sim_L$ η^H **False** but $\neg(\eta^H$ **True** $\sim_H \eta^H$ **False**$)$. We define this relation $\sim_\ell$ by structural recursion on terms. We think of terms as ASTs annotated at various nodes with labels, 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 just as a blob. So we can also read $a \sim_\ell b$ as a is syntactically equal to b at all parts of the terms marked with any label ℓ_0, where $\ell_0 \leq \ell$. The terms a and b are allowed to be arbitrarily different at corresponding sub-trees that are marked by ℓ_0 where $\neg(\ell_0 \leq \ell)$.

Looking at the relation, rule SGEq-Return is interesting. It uses an auxiliary relation, $\Phi \vdash^{\ell_0}_{\ell} a_1 \sim a_2$. This auxiliary *extended equivalence* relation $\Phi \vdash^{\ell_0}_{\ell} a_1 \sim a_2$ formalizes the idea discussed above: if $\ell_0 \leq \ell$, then a_1 and a_2 must be guarded equal at ℓ; otherwise, they may be arbitrary terms.

Now, we explore some properties of the guarded equality relation.[3]

[2]Our expressions have the same names as their DCC counterparts. This is done purposefully to make the connection clearer.
[3]Because this relation is untyped, its dependent analogue is similar. For each lemma below, we include a reference to the

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

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

$$\boxed{\Phi \vdash a \sim_\ell b} \qquad \textit{(Guarded Equality)}$$

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

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

Fig. 3. Guarded Equality for SDC (Excerpt)

Lemma 3.5 (Equivalence[4]). *Guarded equality is an equivalence relation.*

We call the diagonal of this equivalence relation "grading", because it ensures that any variable visible to an observer ℓ has a grade k in the context where $k \leq \ell$. This grading property is implied by the typing relation. Here, $|\Omega|$ extracts the grades from the context Ω.

Lemma 3.6 (Typing implies grading[5]). *If $\Omega \vdash a :^{\ell} A$ then $|\Omega| \vdash a \sim_\ell a$.*

The guarded equality relation is closed under extended equivalence. The following is like a substitution lemma for guarded equality.

Lemma 3.7 (Guarded equality extended substitution[6]). *If $\Phi, x\!: \ell \vdash b_1 \sim_k b_2$ and $\Phi \vdash^{\ell}_{k} a_1 \sim a_2$ then $\Phi \vdash b_1\{a_1/x\} \sim_k b_2\{a_2/x\}$.*

Consider the situation when $\neg(\ell \leq k)$, for example, when $\ell = H$ and $k = L$. Then, for any two terms a_1 and a_2, if $\Phi, x\!: \ell \vdash b_1 \sim_k b_2$, then $\Phi \vdash b_1\{a_1/x\} \sim_k b_2\{a_2/x\}$. To understand better, let us work out a concrete example. For a typing derivation $x :^{H} A \vdash b :^{L} \mathbf{Bool}$, we have the corresponding grading derivation $x\!: H \vdash b \sim_L b$. Also, let $\varnothing \vdash a_1 :^{H} A$ and $\varnothing \vdash a_2 :^{H} A$. 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 guarded equality relation respects the step semantics. We use a call-by-value reduction here.[7]

Lemma 3.8 (Guarded equality respects step[8]). *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, there is exactly one such a_2' that a_1' steps to. 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 guarded equal after each and every step. Since the language itself is terminating, both the terms reduce to boolean values, values that are guarded equal. Now, guarded equality 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.

location in the Coq development where it may be found for the dependent system.
[4]geq.v:GEq_refl,GEq_symmetry,GEq_trans [5]typing.v:Typing_Grade [6]subst.v:CEq_GEq_equality_substitution
[7]Since the dependent system is non-terminating, there we use call-by-name. [8]geq.v:CEq_GEq_respects_Step

The guarded equality relation gives us a syntactic method of proving non-interference for programs derived in SDC. Further, since the guarded equality relation respects the step semantics, we can conclude that information never flows in the wrong direction in our system. In other words, programs derived in our calculus are secure by design.

3.3 Relation with Sealing Calculus and the Dependency Core Calculus

SDC is *extremely* similar to the sealing calculus $\lambda^{[]}$ of Shikuma and Igarashi [2006]. Like SDC, $\lambda^{[]}$ has a label on the typing judgement. (Note that our labels correspond to observer levels of Shikuma and Igarashi [2006], which can be viewed as a lattice.) But unlike SDC, the assumptions in the contexts are not labelled in $\lambda^{[]}$. 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 where SDC uses **bind**. We present the rules for sealing and unsealing terms in $\lambda^{[]}$. (We take the liberty of making some cosmetic changes in the presentation.)

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

Shikuma and Igarashi [2006] have previously shown that $\lambda^{[]}$ is equivalent to DCC$_{\text{pc}}$, an extension of the terminating fragment of DCC. Therefore, we compare SDC to DCC by simulating $\lambda^{[]}$ in SDC. We do so by defining a translation $\overline{\cdot}$, from $\lambda^{[]}$ to SDC. Most of the cases are defined inductively in a straightforward manner. For **unseal**, we have, $\overline{\mathbf{unseal}^{\ell} a} = \mathbf{bind}^{\ell}\ x = \overline{a}\ \mathbf{in}\ x$.

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 form used by Shikuma and Igarashi [2006] for $\lambda^{[]}$. Full reduction is a non-deterministic reduction strategy whereby a β-redex in any sub-term may be reduced.

Lemma 3.9 (Forward Simulation). *If $a \rightsquigarrow a'$ in $\lambda^{[]}$, then $\overline{a} \rightsquigarrow \overline{a'}$ in SDC.*

Lemma 3.10 (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 3.11 (Translation Preserves Typing). *If $\Gamma; \ell \vdash a : A$, then $\Gamma^{\ell} \vdash \overline{a} :^{\ell} A$.*

The above translation shows that DCC can be embedded into SDC. So SDC is at least as expressive as the terminating fragment of DCC. Further, SDC lends itself nicely to syntactic methods; we presented a relatively easy syntactic proof of non-interference. Syntactic methods generally carry forward smoothly to the dependent setting. Therefore, we use SDC as a basis for exploring applications of dependency tracking in dependent types.

4 A DEPENDENT DEPENDENCY-TRACKING CALCULUS: PREPARING THE STAGE

4.1 Lattice Structure for Tracking Irrelevance

We want our dependent type system to track dependency, especially irrelevance. We saw two kinds of irrelevance: compile-time irrelevance and run-time irrelevance. Which lattice structure is best suited to represent both these kinds of irrelevance? Here, we investigate this question.

Let us start with just two levels: $\{\top, \bot\}$ with $\bot \leq \top$ and see what we can do with them. The idea is to use $\top$ for a compile-time observer and $\bot$ for a runtime observer. Can we, then, distinguish the erasable parts of programs (to be marked by $\top$) from the parts necessary at run time (to be marked by $\bot$)? Let's look at an example: consider the polymorphic identity function and its type.

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

In this section, we treat runtime observer to be the default. If the label on a variable or term is left off, it is understood to be $\bot$. Coming back to the example, the first parameter of the identity function is only needed during type checking; it can be erased before execution. The $\top$ on the parameter signifies this fact. While we apply this function, as in **id Bool**$^\top$ **True**, we can erase the first argument, **Bool**, but the second one, **True**, is needed at runtime.

Note that though the argument `x` is irrelevant in the body of the function (i.e. in `λy.y`), it is relevant in the body of the type (i.e. in `x -> x`). Can a variable be irrelevant in a term but be relevant in its type? This question has no one answer. Some authors [Abel and Scherer 2012; Pfenning 2001] do not allow this form of irrelevant quantification while others [Barras and Bernardo 2008; Mishra-Linger and Sheard 2008] do. We discuss this design choice further in Section 7.1.

The above example may lead one to think that though irrelevant variables may appear in types, they never appear in terms. But that is not the full story: irrelevant variables can appear in terms, as long as they do in irrelevant contexts. Let us understand this through an example.

Consider the `Vec` datatype for length-indexed vectors and an associated mapping operation, as they might look in a core language inspired by GHC [Sulzmann et al. 2007; Weirich et al. 2017]. Below, the syntax notates that the `Vec` datatype has two parameters, `n` and `a`, that appear in the types of the data constructors. Such parameters are relevant to the type `Vec`, but irrelevant to the constructors `Nil` and `Cons`. (In the types of the data constructors, the equality constraints `(n ~ Zero)` and `(n ~ Succ m)` force `n` to be equal to the length of the vector. We won't show these constraints explicitly in the rest of this paper, but they are explicit in GHC's internal language.)

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

Since `m` is irrelevant to `cons`, we label `m` with $\top$. We would like this value to be erasable and not stored with the vector at run time.

Now consider a mapping function `vmap` on vectors which maps a given function over a given vector. The length of the vector and the type arguments are not really necessary while 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 all of the variables `m`, `a` and `b` appear in the definition of `vmap`, but since they appear in $\top$ contexts, we are fine. So a two element lattice gives us the ability to distinguish between erasable and relevant terms.

We can use this analysis to produce leaner code and run programs faster. But there are at least two other things we can do with irrelevance analysis: type-check faster and track erasable but compile-time relevant programs, as in **Complex** example. We shall consider faster type-checking later; for now, let's look at erasable but compile-time relevant programs.

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

Fig. 4. Dependent Dependency Calculus Grammar

Till now, we were just differentiating between erasable and relevant programs and a two level lattice served us good. With erasable but compile-time relevant programs joining the game, we would need another level. Let's call the level C; it's between $\top$ and $\bot$. We now consider an example that puts our lattice to test: the **`filter`** function for length-indexed vectors.

The **`filter`** function filters a vector based on a predicate. The difficulty with this function is that we cannot statically predict the length of the vector that will be returned, so we package the result in a Σ-type.

```
filter  : Π n:^⊤ Nat. Π a:^⊤ Type. (a -> Bool) -> Vec n a -> Σ m:^C Nat. Vec m a
filter  =   λ^⊤ n a. λ f vec.
            case vec of
              Nil -> (Zero^C, Nil)
              Cons n1^⊤ x xs
                 | f x        -> let ys : Σ m:^C Nat. Vec m a
                                     ys = filter n1^⊤ a^⊤ f xs
                                 in ((Succ (π₁ ys))^C, Cons (π₁ ys)^⊤ x (π₂ ys))
                 | otherwise  -> filter n1^⊤ a^⊤ f xs
```

Since `n` and `a` are not required at runtime, we assign them $\top$. But what do we assign to `m`? Surely, it is also not required during runtime; but, it is required for type-checking, if we want to get the output vector using projection. So it should not be assigned $\bot$, it should not be assigned $\top$ either; we assign it C.

Eisenberg et al. [2021] observe that in Haskell it is important to use projection functions to access the components of the Σ type that results from the recursive call (as in π_1 `ys`, π_2 `ys`) to ensure that this function is not excessively strict. If **`filter`** had been written with pattern matching to eliminate the Σ-type instead, it would then have to filter the entire vector before returning the first successful value.

So with three levels, we can track run-time relevant, run-time irrelevant but compile-time relevant and compile-time irrelevant programs. But can we also make type checking faster? Short answer: yes. How? Our story shall follow this question as we go along. Let us now move on to the language basics.

4.2 Dependent Dependency Tracking: The Basics

We progressively extend the simple dependency-tracking system to include dependent types. The first extension, called DDC$^{\top}$ and described in Section 4.3, uses dependency tracking for run-time irrelevance. Then, in Section 5, we present the full language DDC, which generalizes the first version and supports both run-time and compile-time irrelevance. We present the system in this way both to simplify the presentation and to provide a simpler point in the design space for languages that do not need compile-time irrelevance.

$\boxed{\Omega \vdash a :^{\ell} A}$ *(Simple Dependent Types)*

$$\text{SDT-Var}\quad \frac{\ell_0 \leq \ell \qquad x :^{\ell_0} A \in \Omega}{\Omega \vdash x :^{\ell} A} \qquad \text{SDT-Type}\quad \frac{}{\Omega \vdash \mathbf{Type} :^{\ell} \mathbf{Type}} \qquad \text{SDT-Pi}\quad \frac{\Omega \vdash A :^{\ell} \mathbf{Type} \quad \Omega, x :^{\ell} A \vdash B :^{\ell} \mathbf{Type}}{\Omega \vdash \Pi x :^{\ell_0} A.B :^{\ell} \mathbf{Type}}$$

$$\text{SDT-Abs}\quad \frac{\Omega, x :^{\ell_0 \vee \ell} A \vdash b :^{\ell} B \quad \Omega \vdash (\Pi x :^{\ell_0} A.B) :^{\top} \mathbf{Type}}{\Omega \vdash \lambda^{\ell_0} x.b :^{\ell} \Pi x :^{\ell_0} A.B} \qquad \text{SDT-App}\quad \frac{\Omega \vdash b :^{\ell} \Pi x :^{\ell_0} A.B \quad \Omega \vdash a :^{\ell_0 \vee \ell} A}{\Omega \vdash b\ a^{\ell_0} :^{\ell} B\{a/x\}} \qquad \text{SDT-Conv}\quad \frac{\Omega \vdash a :^{\ell} A \quad |\Omega| \vdash A \equiv_{\top} B \quad \Omega \vdash B :^{\top} \mathbf{Type}}{\Omega \vdash a :^{\ell} B}$$

Fig. 5. DDC$^{\top}$ type system (core rules)

Both DDC$^{\top}$ and DDC share the same syntax, shown in Figure 4, combining terms and types into the same grammar. For simplicity, we work in a setting with the **Type** : **Type** axiom. This should not be seen as a limitation. Universe levels do not interact with the graded structures that we present here; so a Martin-Löf Type Theory version of our system is entirely possible. However, by working in the Type-in-Type calculus, we show that our results do not require strong normalization.

Many expression forms in the syntax are annotated with labels. In a source language, type inference might be used to add these labels automatically. Here, we use these labels for syntactic reasoning about dependence.

Dependent function types, written $\Pi x :^{\ell} A.B$ include a label for the argument of the function. Similarly, weak dependent pair types, written $\Sigma x{:}^{\ell} A.B$ and eliminated via pattern matching, include a label for the first component of the pair.[9] We can interpret these types as a fusion of the usual, unlabelled dependent types and the graded modality $T^{\ell}\ A$ of the simply-typed language. In other words, $\Pi x :^{\ell} A.B$ acts like the type $\Pi y{:}(T^{\ell}\ A).\mathbf{bind}\, x = y\, \mathbf{in}\, B$. Because of this fusion, we do not need to add the graded modality type as a separate form—we can always recover it through pairing with unit: $T^{\ell}\ A = \Sigma x{:}^{\ell} A.\mathbf{Unit}$. In DDC$^{\top}$, this fusion is convenient but not necessary. However, in full DDC (Section 5), this design is necessary to allow compile-time irrelevant variables to be relevant in the codomain of dependent function types.

4.3 DDC$^{\top}$: Tracking Run-time Irrelevance

We now demonstrate how dependency tracking can identify erasable code in a dependently-typed language. As discussed earlier, we need a lattice with at least two elements to track run-time irrelevance. The core typing rules for DDC$^{\top}$ appear in Figure 5. As in the simple type system, all variables in the context are labelled and the judgement itself includes a label ℓ, which is the security level of the expression. Rule SDT-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 SDT-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 that does not interact with the rule: 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 rules SDT-Abs and SDT-App, we augment the usual rules for dependent functions with graded modality. What this means is that in rule SDT-Abs, the parameter of the function is introduced

[9]In examples, we use standard abbreviations when x is not free in B: we write ${}^{\ell}A \to B$ for $\Pi x :^{\ell} A.B$ and ${}^{\ell}A \times B$ for $\Sigma x{:}^{\ell} A.B$.

into the context at level $\ell_0 \vee \ell$ (as in **bind**) and that in rule SDT-App the argument to a function application is checked at level $\ell_0 \vee \ell$ (as in η).

Note that the Π-type is checked at $\top$ in rule SDT-Abs. In DDC$^\top$, level $\top$ corresponds to an observer at 'compile time'. As such, the type system must verify that types are well-formed at $\top$, not at the current level of the judgement.

Rule SDT-Conv converts the type of an expression to an equivalent type. The judgement $|\Omega| \vdash A \equiv_\top B$, is a label-indexed definitional equality. This relation contains the small-step evaluation relation and is described in more detail in Section 5.1. The evaluation relation is standard call-by-name reduction. However, for DDC$^\top$, because the index is $\top$, this relation degenerates to untyped β-equivalence. Since the relation is untyped, the third premise is needed to guarantee that the new type is well-formed.

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

$$\textsc{SDT-WSigma}\quad \frac{\Omega \vdash A :^{\ell} \textbf{Type} \qquad \Omega, x{:}^{\ell} A \vdash B :^{\ell} \textbf{Type}}{\Omega \vdash \Sigma x{:}^{\ell_0} A.B :^{\ell} \textbf{Type}}$$

$$\textsc{SDT-WPair}\quad \frac{\Omega \vdash a :^{\ell_0 \vee \ell} A \qquad \Omega \vdash b :^{\ell} B\{a/x\} \qquad \Omega \vdash \Sigma x{:}^{\ell_0} A.B :^{\top} \textbf{Type}}{\Omega \vdash (a^{\ell_0}, b) :^{\ell} \Sigma x{:}^{\ell_0} A.B}$$

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

Similar to Π types, Σ types incorporate the modal type $T^{\ell_0}\ A$ in the their first component. Therefore, we annotate the first component of pairs with a level that tracks its dependency. When we form a pair, in rule SDT-WPair, we check the first component a of the pair at a level that may be raised by ℓ_0, the level annotating the type, as in rule SDC-Return. The second component b is checked at the current level and its type is allowed to mention a. Finally, the rule also ensures that the type itself is well-formed, at level $\top$.

The rule SDT-LetPair eliminates Σ types using dependently-typed pattern matching. The scrutinee a must have a dependent pair type at the current level ℓ. Then, x and y are introduced into the context while checking the body c, also at the current level ℓ. As in **bind**, the level of the first component of the pair 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 dependently-typed projection operations using pattern matching. In particular, the first projection, $\pi_1^{\ell_0}\ a$ can be defined as **let** $(x^{\ell_0}, y)\ =\ a$ **in** x while the second projection, $\pi_2^{\ell_0} a$ as **let** $(x^{\ell_0}, y)\ =\ a$ **in** y. These projections can be type checked according to the following derived rules:

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

In the derived rule, we find that rule SDT-Proj1 limits access to the first component through the premise $\ell_0 \leq \ell$, as in rule Sealing-Unseal. This condition makes sense because it aligns the erasability of the first component of the pair with the label on the Σ type. If this component is marked with $\top$, then it can only be projected in the parts of the program that are marked as compile-time.

Note that in DDC$^\top$, the conversion rule SDT-Conv uses equalities derived at $\top$. The definitional equality may be thought of as the guarded equivalence relation, as defined in Section 3.2, but one that is closed under the step relation. Recall that the guarded equality relation at ℓ ignores all sub-terms marked with ℓ_0 where $\neg(\ell_0\ \leq\ \ell)$. Now, the guarded equality at $\top$ cannot ignore any sub-terms in DDC$^\top$ because for any ℓ_0, we know $\ell_0\ \leq\ \top$. But, we want to ignore compile-time irrelevant terms while checking for type equality, as we discussed in the `phantom` example. To address this, we need the type system to use equality at a grade less than $\top$.

5 DDC: RUN-TIME AND COMPILE-TIME IRRELEVANCE

We now extend our dependently typed language to also support compile-time irrelevance: we want to ignore compile-time irrelevant terms when checking for type equality in the language. Coarsening the equality used in the conversion rule can have significant practical benefits during type checking, as we saw in `phantom` example.

Of course, we should not overcoarsen our definition of equivalence. Going to an extreme we could equate all types, but such a type system would not be sound. Therefore, we need to make sure that the type system correctly identifies which parts of an expression are *safe* to ignore during type checking.

As discussed earlier, we need to work with a lattice with at least three elements: $\bot < ... < C < \top$. Here, the label $\bot$ (and any other label less than C) marks run-time terms. C indicates that a sub-term is erasable at run time, but relevant at compile time, and $\top$ is reserved for the parts of a program that are irrelevant in both contexts.

To see why these are different, consider both `x` and `y` below, which have types that are equal to **`Int`**. In the first example, we apply the polymorphic identity function to **`Int`**; its first argument (**`Type`**) is not relevant at compile time. So it is safe to mark this argument with $\top$ (and allow the compiler to ignore it when reducing this expression to **Int**). In the second example, we also compute **`Int`** by projecting from a dependent pair. Note that the first component of this pair must be marked as C, not $\top$, because it is used to compute the type of `y`.

```
x : id Type^⊤ Int
x = 5

y : π₁ (Int^C, unit)
y = 4
```

With this intuition, there are two problems that we need to solve in tracking and using compile-time irrelevance. First, we need a version of definitional equality for the conversion rule that ignores subterms marked as $\top$. Second, we must safely employ this definition of equality in the type system.

5.1 Graded definitional equality

Creating a level-aware version of definitional equality is not so difficult. Our definition of Guarded Equality, from Section 3.2, is a good inspiration.

Graded definitional equality, written $\Phi \vdash a \equiv_\ell b$, is an untyped equivalence relation that contains β–reduction and is a congruence relation for the parts of terms that are observable by ℓ. Otherwise, for the parts of the terms that are not less than or equal to ℓ, this relation is the total relation, meaning the relation does not distinguish what it cannot observe. When ℓ is $\top$, this relation can observe everything and as such, degenerates to β-equivalence.

However, when $\ell < \top$, this relation equates more terms than just β-equivalence. For example, the two congruence rules for applications (rules Eq-AppRel and AppIrrel) differ based on the label annotated on the argument. If the argument is visible, i.e. if its label is less than or equal to that of the observer, then the two arguments must be equal. Otherwise, if the argument is secret, then there is no such restriction. Any pair of arguments will produce equal results.

Eq-AppRel

$$\frac{\Phi \vdash b_1 \equiv_\ell b_2 \qquad \Phi \vdash a_1 \equiv_\ell a_2 \qquad \ell_0 \leq \ell}{\Phi \vdash b_1\ a_1{}^{\ell_0} \equiv_\ell b_2\ a_2{}^{\ell_0}}$$

Eq-AppIrrel

$$\frac{\Phi \vdash b_1 \equiv_\ell b_2 \qquad \neg(\ell_0 \leq \ell)}{\Phi \vdash b_1\ a_1{}^{\ell_0} \equiv_\ell b_2\ a_2{}^{\ell_0}}$$

For example, we can use rule Eq-AppIrrel to show that `phantom a`$^\top \equiv_C$ `phantom b`$^\top$, as long as we know that the argument to `phantom` is irrelevant at compile time.

These rules are similar in spirit to the extended equality relation (Section 3.2), used to sometimes ignore the contents of $\eta^{\ell_0}\ a$. Furthermore, there is an analogue to guarded equality definable for DDC and that relation is contained in definitional equality. (The key difference between the two is that definitional equality also contains β-reductions, so equates many more terms.)

Lemma 5.1 (Contains guarded equivalence[10]). *If $\Phi \vdash a \sim_\ell b$ then $\Phi \vdash a \equiv_\ell b$*

Like guarded equality, definitional equality includes a label-context Φ that specifies the labels of all variables that occur in the visible parts of the terms. This label context tells us what sort of substitutions are valid for this equality. If the variable is visible, then it must be substituted by related terms at the same level of equivalence. Otherwise, if the variable is not visible, then any pair of terms can be used for substitution.

Lemma 5.2 (Relevant substitution[11]). *If $\Phi, x\!: k \vdash b_1 \equiv_\ell b_2$ and $k \leq \ell$ and $\Phi \vdash a_1 \equiv_\ell a_2$, then $\Phi \vdash b_1\{a_1/x\} \equiv_\ell b_2\{a_2/x\}$.*

Lemma 5.3 (Irrelevant substitution[12]). *If $\Phi, x : k \vdash b_1 \equiv_\ell b_2$ and $\neg(k \leq \ell)$, then $\Phi \vdash b_1\{a_1/x\} \equiv_\ell b_2\{a_2/x\}$.*

5.2 When compile-time isn't ⊤

Guarded equality in hand, we now must allow the language to make use of it. This is more challenging than it first seems. We cannot just replace the definitional equality in rule SDT-Conv with $|\Omega| \vdash A \equiv_C B$ without making other changes to the type system.

For example, when types are compared for equality, the type system must ensure that there are no dependencies on ⊤-marked subterms . Can this condition fail to hold? In DDC$^\top$, types are checked at ⊤. So the first projection from a pair of type $\Sigma x{:}^\top A.B$ can be used in types. However, in this language, that would be unsound, as it would allow the following program to type check.

```
y : π₁ (Int ⊤, unit) -> Int
y = \x. x + 1
x : π₁ (Bool ⊤, unit)
x = true

boom = y x
```

This example would unfortunately type check because we have an equality between the two pairs—both the first components are marked as compile-time irrelevant, so definitional equality at level C is free to ignore them.

$\vdash \pi_1$ `(Int`$^\top$`, unit)` $\equiv_C \pi_1$ `(Bool`$^\top$`, unit)`

So we cannot check types at ⊤ and then check type equality at C. If we want to use C for checking type equality, we must check all types at level C as well. But this modification poses its own difficulties—sometimes variables marked as ⊤ are valid in type expressions, and we don't want

[10] defeq.v:CEq_DefEq [11] defeq.v:DefEq_equality_substitution [12] defeq.v:DefEq_substitution_irrel2

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

$$\textsc{T-Var}\quad \frac{\begin{array}{c}\ell_0 \leq \ell \\ x :^{\ell_0} A \in \Omega \qquad \ell \leq C\end{array}}{\Omega \vdash x :^{\ell} A} \qquad \textsc{T-Type}\quad \frac{\ell \leq C}{\Omega \vdash \mathbf{Type} :^{\ell} \mathbf{Type}} \qquad \textsc{T-Pi}\quad \frac{\begin{array}{c}\Omega \vdash A :^{\ell} \mathbf{Type} \\ \Omega, x :^{\ell} A \vdash B :^{\ell} \mathbf{Type}\end{array}}{\Omega \vdash \Pi x :^{\ell_0} A.B :^{\ell} \mathbf{Type}}$$

$$\textsc{T-AbsC}\quad \frac{\begin{array}{c}\Omega, x :^{\ell_0 \vee \ell} A \vdash b :^{\ell} B \\ \Omega \Vdash (\Pi x :^{\ell_0} A.B) :^{\top} \mathbf{Type}\end{array}}{\Omega \vdash \lambda^{\ell_0} x.b :^{\ell} \Pi x :^{\ell_0} A.B} \qquad \textsc{T-AppC}\quad \frac{\begin{array}{c}\Omega \vdash b :^{\ell} \Pi x :^{\ell_0} A.B \\ \Omega \Vdash a :^{\ell_0 \vee \ell} A\end{array}}{\Omega \vdash b\ a^{\ell_0} :^{\ell} B\{a/x\}} \qquad \textsc{T-ConvC}\quad \frac{\begin{array}{c}\Omega \vdash a :^{\ell} A \\ \lfloor C \wedge \Omega \rfloor \vdash A \equiv_C B \\ \Omega \Vdash B :^{\top} \mathbf{Type}\end{array}}{\Omega \vdash a :^{\ell} B}$$

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

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

Fig. 6. Dependent type system with compiletime irrelevance (core rules)

to rule them out. For example, when type checking the polymorphic identity function, the type system must show that the inner function is well-formed, i.e. $x :^{\top} \mathbf{Type} \vdash \lambda y.y :^{\bot} x \to x$. But then, we cannot show $x :^{\top} \mathbf{Type} \vdash x \to x :^{C} \mathbf{Type}$. A problem!

To move out of this impasse, we look to EPTS [Mishra-Linger and Sheard 2008; Mishra-Linger 2008]. We cannot use EPTS directly, as it is hard-wired to only two levels and we want to support at least three. But ETPS does provide a clue about how to safely work with multiple levels of dependency.

The key idea, adapted from Mishra-Linger and Sheard [2008], is to use a judgement of the form $C \wedge \Omega \vdash a :^{C} A$ instead of a judgement of the form $\Omega \vdash a :^{\top} A$, whenever the type system needs to check types at 'compile time'. 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 and making it available for use in a C-expression. We call this operation *truncation* and it is related to the Reset rule from EPTS. Due to the use of truncation, even though the judgement is checked at C, all $\top$ marked variables are still available.

DDC is a generalization of EPTS. Under the assumption that $C = \bot$, i.e. where the lattice has only two levels, it degenerates to a form equivalent to EPTS. However, in that setting, the type system cannot distinguish between run-time and compile-time irrelevance.

5.3 Π types

The core typing rules of DDC are shown in Figure 6. 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 by deriving $C \wedge \Omega \vdash a :^{C} A$ in place of $\Omega \vdash a :^{\top} A$.

So wherever DDC$^{\top}$ uses $\top$ as the observer level on a typing judgement, 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$^{\top}$ judgement $\Omega \vdash a :^{\ell} A$ is replaced with a judgement form, $\Omega \Vdash a :^{\ell} A$

which can be read as: if $\ell = \top$, use the truncated version $C \wedge \Omega \vdash a :^{C} A$; otherwise use the normal version $\Omega \vdash a :^{\ell} A$. In Figure 6, uses of this new judgement have been highlighted in gray to emphasize the modification.

Finally, observe that rule T-ConvC uses the definitional equality at C instead of $\top$. To enable this demotion, the context is also truncated in this premise.

Example: identity function. To see how full irrelevance works in this system, let's consider the definition and use of the polymorphic identity function. We want the type parameter of this function to be both eraseable and compile-time irrelevant, so we use the same definitions as discussed earlier.

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

Before, the type system checked the type `Π x :⊤Type. x -> x` at level $\top$. However, in this version, it must be checked at level C. Rule T-Pi is unchanged: to check the body of the Pi, we need to make sure that `x:`^{C}`Type ⊢ x -> x :`^{C} `Type`. Note that if we used $\ell_0 \vee \ell$ as the level of `x` in rule T-Pi, we would have been in trouble now. It is safe to consider `x` to be compile-time relevant while checking the type, even though `x` must be irrelevant in the term.[13] While checking the term, we do keep `x` at $\top$ in the context, making it fully unavailable.

5.4 Σ types

The updates to Figure 6 also apply to the DDC rules for Σ types. In particular, when we create a pair, we check the first component and the type itself using the conditional typing judgement. Likewise, we use this judgement when checking the result type during pattern matching.

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

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

However, observe what happens to our encoding of the projections.[14] The derived form for the first projection is unchanged; we cannot project that component unless we are already at that level.

$$\textsc{T-Proj1}\quad \frac{\Omega \vdash a :^{\ell} (x{:}^{\ell_0} A)\ \&\ B \qquad \ell_0 \leq \ell}{\Omega \vdash \pi_1^{\ell_0}\ a :^{\ell} A}
\qquad
\textsc{T-Proj2C}\quad \frac{\Omega \vdash a :^{\ell} \Sigma x{:}^{\ell_0} A.B \qquad \ell_0 \leq C}{\Omega \vdash \pi_2^{\ell_0} a :^{\ell} B\{\pi_1^{\ell_0}\ a/x\}}$$

However, the second projection rule gains a new constraint—we can only project from tuples where the first component is C or less. This constraint shows up because the first projection appears in the type of the second projection. This type is checked at level C, so the level of the projected first component can be at most C.

What this means is that the first component of a Σ-type may be erasable, but never compile-time irrelevant if we want to eliminate it via projection. If we have a type of the form $\Sigma x{:}^{\top} A.B$, the only way to eliminate it is via pattern matching. As a result, the pattern matching form of the elimination rule is strictly more expressive than the (derived) projections.

[13] This is why we fuse the graded modality and the dependent types. If they were separated, and we had to unbind here, it would be a problem since a dependent function and its type have different restrictions vis-à-vis the bound variable.
[14] strong_exists.v:T_wproj1,T_wproj2

5.5 Soundness theorem

DDC is type sound. We have proved the soundness results using the Coq proof assistant. These proofs are available to reviewers as supplementary material. 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$^\top$. In fact, the typing rules for DDC$^\top$ are the same as the ones for DDC when we assume that C equals $\top$.

First, we list the properties related to grading that hold for all judgements: guarded equality, definitional equality, and typing. (We only state the lemmas for typing, the others are analogous.) These lemmas are similar to those in Section 3.1.

Lemma 5.4 (Narrowing[15]). *If $\Omega \vdash a :^\ell A$ and $\Omega' \leq \Omega$, then $\Omega' \vdash a :^\ell A$*

Lemma 5.5 (Weakening[16]). *If $\Omega_1, \Omega_2 \vdash a :^\ell A$ then $\Omega_1, \Omega, \Omega_2 \vdash a :^\ell A$.*

Lemma 5.6 (Pumping[17]). *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 judgement. For any typing judgement in DDC, the observer grade ℓ is at most C. Further, the observer grade of any judgement can be raised up to C.

Lemma 5.7 (Bounded by C[18]). *If $\Omega \vdash a :^\ell A$ then $\ell \leq C$.*

Lemma 5.8 (Subsumption[19]). *If $\Omega \vdash a :^\ell A$ and $\ell \leq k$ and $k \leq C$ then $\Omega \vdash a :^k A$*

As before, the proof of the subsumption lemma requires pumping in order to raise the level of variables up to the level of the term.

Through narrowing and subsumption, we can transform a well-typed expression at any grade to be suitable to be used in at ‘compile time’.

Corollary 5.9 (Lift[20]). *If $\Omega \vdash a :^\ell A$ then $\Omega \Vdash a :^\top A$.*

Note that we don’t require contexts to be well-formed in the typing judgement; we add context well-formedness constraints, as required, to our lemmas. The following lemmas are only valid with well-formed contexts. The context well-formedness judgement $\vdash \Omega$ says that for any assumption $x :^\ell A$ in Ω, we have $\Omega' \Vdash A :^\top$ **Type**, where Ω' is the prefix of the context before the assumption.

Lemma 5.10 (Substitution[21]). *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 5.11 (Regularity[22]). *If $\Omega \vdash a :^\ell A$ and $\vdash \Omega$ then $\Omega \Vdash A :^\top$ **Type**.*

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

Lemma 5.12 (Preservation[23]). *If $\Omega \vdash a :^\ell A$ and $\vdash \Omega$ and $a \leadsto a'$, then $\Omega \vdash a' :^\ell A$.*

Lemma 5.13 (Progress[24]). *If $\varnothing \vdash a :^\ell A$ then either a is a value or there exists some a' such that $a \leadsto a'$.*

[15]narrowing.v:Typing_narrowing [16]weakening.v:Typing_weakening [17]pumping.v:Typing_pumping
[18]pumping.v:Typing_leq_C [19]typing.v:Typing_subsumption [20]typing.v:Typing_lift
[21]typing.v:Typing_substitution_CTyping [22]typing.v:Typing_regularity [23]typing.v:Typing_preservation
[24]progress.v:Typing_progress

The proof of the progress lemma requires showing that definitional equality is consistent. Consistency of definitional equality means that there is no derivation that equates two types having different head forms. We prove this property by constructing a parallel reduction relation and showing that this relation is confluent. We show that if two terms are definitionally equal, then they reduce, through parallel reduction, to two terms that are related by the guarded equality relation. The proof of consistency holds regardless of the level used for definitional equality.

As mentioned earlier, the proof of non-interference carries over smoothly from the simple to the dependent setting. The corresponding proof scripts for the dependent version are already referenced in Section 3.2. Hence, DDC is type sound and programs derived in the calculus are secure by design. As a corollary, we know that our system tracks irrelevance correctly.

5.6 Extensions and Variations

The current section presents DDC and its application to tracking run-time and compile-time irrelevance. However, there are several variations and extensions that we would like to consider in future work.

Decidable type checking. We have presented DDC in a quasi-Curry-style manner, where the syntax of the language lacks type annotations. Furthermore, because the language is nonterminating, the definitional equality judgement used in conversion is undecidable. As a result, type checking for DDC is undecidable.

However, it would not be difficult to modify this type system to support decidable type checking. The missing ingredients are annotations on λ-bound variables and explicit coercions at uses of definitional equality, similar to the design of FC [Sulzmann et al. 2007] and DC [Weirich et al. 2017] languages. Additionally, the mechanisms presented here can be used to ensure that these new components remain erasable and irrelevant.

Type and dependency inference. We make no attempt in this work to *infer* dependencies; the syntax of this language requires label annotations throughout. As a result, this type system is appropriate for verifying the output of such an inference, perhaps similar to those proposed by Brady et al. [2003]; Tejiščák [2020], once it has been embedded into the syntax of the programming language.

Propositional guarded equality. Because guarded equality is already a component of the design of DDC, it may be possible to internalize the judgement $\Phi \vdash a \equiv_\ell b$ as a proposition/type $a =_\ell b$. This form of propositional equality would allow reasoning about non-interference directly inside the language, permitting internal verification of security properties. We hope to explore this idea in future work.

Roles. In Haskell, users sometimes distinguish between nominal types and their underlying representations (using type-indexed types) while at other times, treat them as the same type (using zero-cost coercions). The type system feature of *roles* [Breitner et al. 2016; Weirich et al. 2019] determines whether representational coercions are safe by tracking the dependencies in parameterized types. In future work, we hope to use or adapt the general mechanisms for dependency tracking, presented here, for implementing roles.

6 GRADED TYPES

In this section, we look at our work from the perspective of graded type theory. The graded type systems, whether effectful or coeffectful, can be divided into two sets. The first set contains all of the type systems based on a graded (co)monad such as the systems from Gaboardi et al. [2016]; Ghica and Smith [2014]; Orchard et al. [2019]; Petricek et al. [2014]. These systems only label variables that were introduced through the (co)monad with a grade. The second set contains all the type systems that may not necessarily have a graded (co)monad, but every type is graded such as the systems from Atkey [2018]; Choudhury et al. [2021]; McBride [2016]; Moon et al. [2021]. We might call the type systems in the second set *fully graded type systems.*

How are these two sets related? We conjecture that fully graded type systems are related to (co)monadic graded type systems through the adjoint relationship between (co)monad and their categories of (co)algebras. We are not currently able to answer this conjecture in full generality, but instead we show that the categories of algebras for graded (co)monads can be abstracted into a new category that captures the semantic notion of being fully graded. This result also shows that SDC should have an equivalent (up to a translation) formalization in terms of a graded monadic effect system. Furthermore, this implies that there should be an embedding of SDC into existing graded effect type systems such as Granule.

In this section, we show that SDC (Fig. 2) is a fully graded effect type system by showing that its categorical model has the structure of a category of graded algebras. We do this by first abstracting the category of graded algebras into a new category called a *grade-indexed multicategory*, that captures the basic structure of fully graded type systems. Then we show that SDC is, indeed, a grade indexed multicategory.

Definition 6.1. Suppose $(\mathcal{R}, \leq)$ is a preorder and $\mathcal{M}$ is a class of objects. Then a *grade-indexed multicategory* $\mathrm{Gr}(\mathcal{R}, \mathcal{M})$ consists of objects, pairs (X, r) where $X \in \mathcal{M}$ and $r \in \mathcal{R}$, and morphisms of the form $f : \langle (X^1, r_1), \ldots, (X^n, r_n) \rangle \longrightarrow (Y, r)$ where $\langle (X^1, r_1), \ldots, (X^n, r_n) \rangle$ is a vector of objects.

Furthermore, a notion of approximation must exist. That is, for any $s \leq s'$, $r_1' \leq r_1, \ldots, r_n' \leq r_n$, and morphism $f : \langle (X^1, r_1), \ldots, (X^n, r_n) \rangle \longrightarrow (Z, s)$ there must be an approximated morphism $\mathrm{approx}(f) : \langle (X^1, r_1'), \ldots, (X^n, r_n') \rangle \longrightarrow (Z, s')$.

The category of graded algebras can be generalized into a category of lax monoidal functors of the shape $X : \mathcal{R} \longrightarrow \mathcal{M}$ and grades $r \in \mathcal{R}$. This generalization is a grade-indexed multicategory where the multimorphisms are induced by the monoidal structure of $\mathcal{M}$. An example of such a category can be found in the model of classified sets.

Definition 6.2. Suppose $\mathcal{L}$ is a set of labels. The category of classified sets $\mathrm{CSet}(\mathcal{L})$ has the following data. Objects are classified sets S which are functors $S : \mathcal{L} \longrightarrow \mathrm{RRel}$ from the discrete category $\mathcal{L}$ to the category of reflexive relations. A morphism between classified sets S and S' is a natural transformation $h : S \longrightarrow S'$.

Let's unpack the previous definition. Here we define RRel to be the category of binary relations that respect reflexivity. That is, given a relation $R_X \subseteq X \times X$ over some set X then $x \, R \, x$ holds for all $x \in X$. Now, a morphism from relation R_X to relation R_Y in RRel is a function $X \xrightarrow{f} Y$ such that $f(x_1) \, R_Y \, f(x_2)$ holds when $x_1 \, R_X \, x_2$ holds for any $x_1, x_2 \in X$. This implies that the components of a natural transformation $S \xrightarrow{h} S'$ between classified sets are morphisms $S(l) \xrightarrow{h_l} S'(l)$ in RRel, and thus, have the previous property.

The definition of classified sets given above is different from the standard definition [Kavvos 2019], but the definition given here makes it quite easy to show that classified sets define a grade-indexed multicategory.

Lemma 6.3 (Graded-Indexed Multicategory of Relations). *Suppose* $(L, \leq, \top, \bot, \sqcup, \sqcap)$ *is a lattice. Then there is a grade-indexed multicategory* $\mathrm{Gr}(L, \mathrm{CSet}(L))$.

The most interesting aspect of the proof of previous fact is the definition of multimorphisms. A multimoirphism $f : \langle (S_1, l_1), \ldots, (S_n, l_n)\rangle \longrightarrow (S, l)$ is the natural transformation:

$$(S_1(l_1) \times \cdots \times S_n(l_n)) \xrightarrow{S_1(a_1)\times\cdots\times S_n(a_n)} (S_1 \times \cdots \times S_n)(l_1 \sqcup \cdots \sqcup l_n) \xrightarrow{f} S(l)$$

where $f : (S_1 \times \cdots \times S_n)(l_1 \sqcup \cdots \sqcup l_n) \longrightarrow S(l)$ is a morphism in RRel that is natural in each label, $a_i : l_i \leq (l_1 \sqcup \cdots \sqcup l_i \sqcup \cdots \sqcup l_n)$ for $1 \leq i \leq n$, and $S_1 \times S_2 = \lambda l'.S_1(l') \times S_2(l')$.

The graded-indexed multicategory $\mathsf{Gr}(L, \mathsf{CSet}(L))$ has lots of structure. We can define the graded cartesian product and internal hom:

$$(S_1, l) \times (S_2, l) = (S_1 \times S_2, l) \qquad (S_1, l) \Rightarrow (S_2, l) = (S_1 \Rightarrow S_2, l)$$

where $S_1 \Rightarrow S_2 = \lambda l.\lambda f.\lambda g.a\, S_1(l)\, b \implies f(a)\, S_2(l)\, g(b)$. The former implies the following:

$$\frac{\langle (S_1, l_1), \ldots, (S_n, l_n)\rangle \xrightarrow{f} (S, l) \times (S', l)}{\langle (S_1, l_1), \ldots, (S_n, l_n)\rangle \xrightarrow{\pi_1(f)} (S, l)} \qquad \frac{\langle (S_1, l_1), \ldots, (S_n, l_n)\rangle \xrightarrow{f} (S, l) \times (S', l)}{\langle (S_1, l_1), \ldots, (S_n, l_n)\rangle \xrightarrow{\pi_2(f)} (S', l)}$$

where $\pi_1(f) = f;\pi_1$ in RRel. In addition, we have the following:

$$\frac{\langle (S_1, l_1), \ldots, (S_n, l_n)\rangle \xrightarrow{f_1} (S, l) \qquad \langle (S'_1, l'_1), \ldots, (S'_n, l'_n)\rangle \xrightarrow{f_2} (S', l)}{\langle (S_1, l_1), \ldots, (S_n, l_n), (S'_1, l'_1), \ldots, (S'_n, l'_n)\rangle \xrightarrow{\mathsf{prodr}(f_1,f_2)} (S, l) \times (S', l)}$$

where $\mathsf{prodr}(f_1, f_2) = f_1 \times f_2$ in RRel. Then we have the standard bijection:

$$\frac{\langle (S_1, l_1), \ldots, (S_{n-1}, l_{n-1}), (S_n, l)\rangle \xrightarrow{f} (S, l)}{\langle (S_1, l_1), \ldots, (S_{n-1}, l_{n-1})\rangle \xrightarrow{\mathsf{curry}(f)} (S_n, l) \Rightarrow (S, l)}$$

This structure gives us everything we need to soundly interpret SDC in classified sets.

Lemma 6.4 (Sound Typing). *Suppose $(L, \leq, \top, \bot, \sqcup, \sqcap)$ is the lattice parameterizing SDC. If $\Omega \vdash a :^{\ell} A$, then there is a multimorphism $[\![a]\!] : \langle [\![\Omega]\!]\rangle \longrightarrow [\![A]\!]_{\ell}$ in $\mathsf{Gr}(L, \mathsf{CSet}(L))$. Furthermore, if $a \rightsquigarrow b$, then $[\![a]\!] = [\![b]\!]$.*

The latter holds by the fact that $\mathsf{Gr}(L, \mathsf{CSet}(L))$ is cartesian closed and the proof of the former is a fairly straightforward proof by induction on the assumed typing derivation.

What we have shown here is that SDC has a model in classified sets and this model can be phrased as a grade-indexed multicategory. This result reveals that the structure of the model is the same kind of structure we find in the categories of algebras of graded monads. Thus, SDC is an example of a fully graded type system.

7 DISCUSSIONS AND RELATED WORK

7.1 Irrelevance and Erasure in Dependent Type Theories

Overall, compile-time and run-time irrelevance has been a well-studied topic in the design of dependent type systems. In some settings, the focus is only on support for run-time irrelevance, also called erasability [Atkey 2018; Brady 2021; McBride 2016; Mishra-Linger and Sheard 2008; Tejiščák 2020]. In other work, only compile-time irrelevance is considered [Abel and Scherer 2012; Nuyts and Devriese 2018; Pfenning 2001]. Some systems support both, but require them to overlap [Barras and Bernardo 2008; Miquel 2001; Mishra-Linger 2008; Weirich et al. 2017]. Our $\text{DDC}^{\top}$, which supports run-time irrelevance only, is most similar to the core language of Tejiščák [2020]. The Agda programming language supports both run-time and compile-time irrelevance, but uses completely separate mechanisms, based on Atkey [2018] and Abel and Scherer [2012] respectively. Nuyts and Devriese [2018] include multiple modalities annotating Π and Σ types, one of which corresponds to compile-time irrelevance.

DDC is the only system that we are aware of that can express both run-time and compile-time irrelevance as separate dependencies in the same framework.

Compile-time irrelevance and Σ-types. A subtle issue is the treatment of strong Σ-types with erasable first components, especially in a setting that includes compile-time irrelevance. Abel and Scherer [2012] point out that strong irrelevant Σ-types make their theory inconsistent. Similarly, EPTS [Mishra-Linger 2008] cannot define the projection from an existential type (i.e. a Σ type with an irrelevant first component) because the first component is required to be irrelevant in all contexts.

In a language with just two levels for tracking irrelevant and relevant terms, adding irrelevant projections from strong Σ-types is problematic. This is because the first projection of a pair having this type is irrelevant but not erasable at compile-time: it is required to type check the second component of the same pair.

This is one of the reasons our lattice for tracking irrelevance requires at least three levels: one for run-time relevant terms, another for run-time irrelevant but compile-time relevant terms and a third for compile-time irrelevant terms.

Type-directed equality. Guarded definitional equality in DDC is a type-oblivious equivalence relation, similar to prior treatments of compile-time irrelevance [Barras and Bernardo 2008; Miquel 2001; Mishra-Linger 2008; Tejiščák 2020; Weirich et al. 2017]. Other prior work [Abel and Scherer 2012; Nuyts and Devriese 2018; Pfenning 2001] consider compile-time irrelevance in the context of typed-directed equality.

These two versions of equality are not equivalent, even for terms that have the same type. Consider the examples below, all of which depend on `Proxy` declared as follows:

```
type Proxy : Π x:⊤Type. x -> Type
```

Now, which of the following types should be equal?

```
t1 : Type
t1 = Proxy (Unit -> Unit)⊤ (λ x:Unit.x)

t2 : Type
t2 = Proxy (Bool -> Bool)⊤ (λ x:Bool.x)

t3 : Type
t3 = Proxy (Unit -> Unit)⊤ (λ x:Unit.unit)
```

Under an untyped definitional equivalence, types `t1` and `t2` can be equated as long as the first argument to `Proxy` is irrelevant. However, an untyped equivalence does not include η-equiavalence for the **`Unit`** type, so the third type `t3` is not the same.

In contrast, type-directed equivalence can equate `t1` and `t3`. However, because type information is used as part of this equivalence, systems such as Abel and Scherer [2012] do not allow `x`, the first argument of `Proxy`, to be irrelevant at compile time. As a result, typed equivalence must distinguish between `t1` and `t2`. More generally, in Agda and other related systems, a compile-time irrelevant variable must appear irrelevantly both in the term and the type. To work around this limitation, the Agda language also includes *shape irrelevance* [Abel et al. 2017]. While we have developed the system here using untyped equivalence, the general ideas could be explored in the context of a typed definitional equivalence.

7.2 Quantitative Type Theories

Our work is closely related to quantitative type theories [Abel and Bernardy 2020; Atkey 2018; Brunel et al. 2014; Choudhury et al. 2021; Ghica and Smith 2014; McBride 2016; Orchard et al. 2019; Petricek et al. 2014]. Such theories track quantities in a type system. Quantities can range from simple natural numbers tracking variable usage to complex multisets of affine transformations used in timing analysis. The essence of quantities is captured by a partially ordered semiring structure Q (or other similar structures). The type system is parametrized by the semiring Q, similar to the lattice structure that parametrizes this work.

A quantitative typing judgement has a semi-graded nature: meaning, the contexts are graded by quantities $q \in Q$, but the term is always derived at a fixed grade, 1. This restriction is necessary: a quantitative system that allows an arbitrary grade on the RHS of a typing judgement is not closed under substitution [Atkey 2018; McBride 2016]. Our type system is not quantitative in nature: as such, we are not bound by this restriction.

One way to read a quantitative typing judgement $x_1 :^{q_1} A_1, x_2 :^{q_2} A_2, \ldots, x_n :^{q_n} A_n \vdash a : A$ is that, to produce one copy of resource a, we need q_i copies of resource x_i, where $1 \leq i \leq n$. We can compare this to our typing judgement: $x_1 :^{\ell_1} A_1, x_2 :^{\ell_2} A_2, \ldots, x_n :^{\ell_n} A_n \vdash a :^{\ell} A$ which can be read as, observer ℓ can see a, assuming x_i is ℓ_i-secure for $1 \leq i \leq n$. Dependency tracking is more *qualitative* in nature than quantitative: it cannot count, but it can compare. Quantitative type systems are good at counting: for example, tracking linear variables or run-time irrelevant variables. But they are not that good at modelling sharing. For example, say, we want to take the projections of a term having a strong Σ-type; how do we split the context between the two projections? We may say that we don't care about the quantities when dealing with terms having strong Σ-types; in which case, we also don't care about the quantities for the projections. But this is not very satisfactory. On the other hand, in our system, we don't need to split. We can get the first projection at some level and the second one at some other level. We have this flexibility since we are not restricted to a fixed grade on the RHS of the typing judgement.

Many dependency analyses, including information flow control, require a system that can compare levels to one another and decide on the flow between the levels. The grade on the RHS comes in handy during such comparison. Quantitative type theories can also control information flow by instantiating the partially-ordered semiring $(+, \times, 0, 1, \leq)$ to a lattice $(\wedge, \vee, \top, \bot, \leq)$; but all the analysis has to be carried out relative to level 1. This makes qualitative type theories like ours a better mechanism for tracking information flow and analysing dependency. On the flip side, we can not track linearity and other properties related to counting.

However, on a two-point lattice, correspondingly a two element semiring, qualitative and quantitative type theories come quite close. Let S be a partially-ordered semiring with two elements $\{0, 1\}$ such that $1 + 1 = 1$ and $1 \leq 0$. Quantitative type systems over such a semiring can track whether a variable is definitely irrelevant (0) or potentially irrelevant (1). The semiring S can also be seen as a two-point lattice with $\bot = 1$ and $\top = 0$ and $\vee = \times$ and $\wedge = +$.

Using this structure with just two grades, we compare our DDC with two quantitative dependent type theories: GRAD of Choudhury et al. [2021] and QTT of [Atkey 2018].

With this semiring, a GRAD typing judgement $\Gamma \vdash a : A$ can be derived in DDC as $\Gamma \vdash a :^{\bot} A$. The GRAD type system axiomatizes an untyped definition of equivalence and can be instantiated with one that includes compile-time irrelevance. However, with only two points, GRAD cannot distinguish between run-time irrelevance and compile-time irrelevance, so cannot support an erasable Σ type eliminated via projections.

Now we compare DDC with QTT. In QTT, there are two fragments, one where resources are tracked (the 1 fragment), and the other where resources are ignored (the 0 fragment). For both resource-aware and resource-agnostic derivations $\Gamma \vdash a :^1 A$ and $\Gamma \vdash a :^0 A$ respectively in QTT, we have corresponding derivations $\Gamma \vdash a :^\perp A$ and $\Gamma \vdash a :^\top A$ in $\text{DDC}^\top$. Going the other way is problematic here too, since QTT allows projections from Σ-types only in the resource-agnostic fragment, whereas DDC allows projections both at C and $\perp$.

Over just two grades, these three type systems are quite close to each other. But, in general, DDC and quantitative type theories are unique in their own ways. The former provides a more precise account of irrelevance and information flow control, while the latter enables counting and tracking of resource usage.

7.3 Dependency Analysis and Dependent Type Theory

Dependency analysis and dependent type theories have come together in some existing work; for example, Algehed and Bernardy [2019]; Bernardy and Guilhem [2013]; Lourenço and Caires [2015].

Algehed and Bernardy [2019] gives an embedding of the Dependency Core Calculus in the Calculus of Constructions (CoC) and derives the non-interference theorem as a corollary of the general parametricity theorem of CoC. This paper follows the arc of a long line of research initiated by Tse and Zdancewic [2004] and carried forward by Bowman and Ahmed [2015]; Shikuma and Igarashi [2006]. The goal of this direction of research has been to derive non-interference from parametricity. Our paper takes a different route altogether: we use dependency to track irrelevance in dependent type theory.

Bernardy and Guilhem [2013] is more related to our work. Erasure is one of the focuses of both the papers. But Bernardy and Guilhem [2013] use internalized parametricity to reason about erasure; so it is important that their type theory is logically consistent. Our work does not rely on the normalizing nature of the theory; we take a direct route to analysing erasure. Further, Bernardy and Guilhem [2013] is more like Abel and Scherer [2012] rather than Mishra-Linger and Sheard [2008] in the treatment of Π-types.

Like our work, Lourenço and Caires [2015] track information flow in a dependent type system. But Lourenço and Caires [2015] focus on more imperative features, like modelling of state while we focus on irrelevance. We hope to explore ideas from Lourenço and Caires [2015] to see whether states can be added to our calculus.

7.4 Graded Type systems and effects

Graded types have roots as far back as nearly thirty years ago with Girard et al. [1992]'s introduction of bounded linear logic. This system introduced the modality $!_p A$ which can be seen as an annotated version of the of-course modality $!A$ with a polynomial p that captures the usage of programs with type A. They use this bounded exponential in complexity analysis. Fast forward six years when Wadler [1998] introduces what one could consider to be the first graded effect type system which has a new notion of labelled monad $T^\sigma A$ annotated with an effect delimiter σ that describes the kinds of effects the monad T has access to. Moving forward, Ghica and Smith [2014] show how to generalize bounded linear logic to be parameterized by a general semiring; the bounded exponential $!_r A$ is now labelled by an element of the semiring. Similar to the previous system, this system is the first graded coeffect system to be introduced. In the same year, Brunel et al. [2014] propose a graded coeffect type system that is perhaps the closest in presentation to modern day graded type systems, and is highly related to Ghica and Smith [2014]'s system; but they provide several interesting meta-theoretical results for their system including a sound operational model.

At this point, the study of graded type systems begins to pick up with new results in the area continuing every year [Atkey 2018; Choudhury et al. 2021; Eades et al. 2019; Fujii et al. 2016;

Gaboardi et al. 2016; Katsumata 2014; McBride 2016; Moon et al. 2021; Petricek et al. 2014]. The calculi presented in this paper can be seen as fully graded effect systems.

Perhaps the system that is most closely related to our system SDC is SDCC of Algehed [2018]. They simplify DCC into a system that exposes an equivalent formalization of DCC as essentially a graded monadic system. However, our system is a fully graded type system. In addition, we are not aware of any dependent graded effect systems other than the Granule language of Orchard et al. [2019], but Granule has a restricted notion of type dependency called index types where DDC has full dependent types.

8 CONCLUSION

Graded type theories have been used for many purposes: from tracking variable usage to enforcing security constraints. We can divide such theories into two broad categories: quantitative or coeffect-based and qualitative or effect-based. The former are good at counting while the latter are good at dependency tracking. Our Dependent Dependency Calculus (DDC) is a member of the latter category and is a general-purpose system for analysing dependency in dependent type theories.

In this paper, we use DDC for one specific application: tracking compile-time and run-time irrelevance. The generality of the language gives us a novel way of handling irrelevance. A benefit of using this approach is that DDC can analyse strong irrelevant Σ-types, a feature that often causes difficulty in the setting of compile-time irrelevance. The modular structure of DDC also gives us the flexibility to choose between several points in the design space (DDC$^{\top}$, EPTS-like, or full DDC) trading type-system complexity for expressiveness.

Another novelty of our calculus lies in its graded design, which enables syntactic reasoning about security properties. We provide a straightforward syntactic proof of non-interference. The syntactic methods are valuable because sometimes semantic models are hard to construct. For example, it is difficult to give a denotational model for a Curry-style dependently typed language tracking irrelevance, or a type-in-type type system tracking irrelevance.

There are many topics worth exploring in future, some of which have already been mentioned: combining irrelevance and information flow in dependent systems, combining quantitative type systems with our calculus, finding the relation between our calculus and graded effect systems in dependent type theories, etc. Because our work is based on a general approach to dependency tracking, and is connected to existing work such as DCC, we believe that it is an appropriate framework for future exploration of dependencies in dependent type systems.

REFERENCES

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 147–160. https://doi.org/10.1145/292540.292555

Andreas Abel and Jean-Philippe Bernardy. 2020. A Unified View of Modalities in Type Systems. *Proc. ACM Program. Lang.* 4, ICFP, Article 90 (Aug. 2020), 28 pages. https://doi.org/10.1145/3408972

Andreas Abel and Gabriel Scherer. 2012. 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

Andreas Abel, Andrea Vezzosi, and Theo Winterhalter. 2017. Normalization by Evaluation for Sized Dependent Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 33 (Aug. 2017), 30 pages. https://doi.org/10.1145/3110277

Maximilian Algehed. 2018. A Perspective on the Dependency Core Calculus. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security* (Toronto, Canada) *(PLAS '18)*. Association for Computing Machinery, New York, NY, USA, 24–28. https://doi.org/10.1145/3264820.3264823

Maximilian Algehed and Jean-Philippe Bernardy. 2019. Simple Noninterference from Parametricity. *Proc. ACM Program. Lang.* 3, ICFP, Article 89 (July 2019), 22 pages. https://doi.org/10.1145/3341693

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 56–65. https://doi.org/10.1145/3209108.3209189

Bruno Barras and Bruno Bernardo. 2008. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. In *Foundations of Software Science and Computational Structures (FOSSACS 2008)*, Roberto Amadio (Ed.). Springer Berlin Heidelberg, Budapest, Hungary, 365–379.

Jean-Philippe Bernardy and Moulin Guilhem. 2013. Type-Theory in Color. *SIGPLAN Not.* 48, 9 (Sept. 2013), 61–72. https://doi.org/10.1145/2544174.2500577

William J. Bowman and Amal Ahmed. 2015. Noninterference for Free. *SIGPLAN Not.* 50, 9 (Aug. 2015), 101–113. https://doi.org/10.1145/2858949.2784733

Edwin Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming (ECOOP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:26. https://doi.org/10.4230/LIPIcs.ECOOP.2021.9

Edwin Brady, Conor McBride, and James McKinna. 2003. Inductive Families Need Not Store Their Indices. In *Types for Proofs and Programs. TYPES 2003 (Lecture Notes in Computer Science, Vol. 3085)*, Damiani F. Berardi S., Coppo M. (Ed.). Springer, Berlin, Heidelberg, 115–129. https://doi.org/10.1007/978-3-540-24849-1_8

Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2016. Safe zero-cost coercions for Haskell. *Journal of Functional Programming* 26 (2016). https://doi.org/10.1017/S0956796816000150

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 351–370.

Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. 2021. A Graded Dependent Type System with a Usage-Aware Semantics. *Proc. ACM Program. Lang.* 5, POPL, Article 50 (Jan. 2021), 32 pages. https://doi.org/10.1145/3434331

Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (May 1976), 236–243. https://doi.org/10.1145/360051.360056

Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (July 1977), 504–513. https://doi.org/10.1145/359636.359712

Harley Eades, III, Domnic Orchard, and Vilem-Benjamin Liepelt. 2019. Linear and graded modal types for fine-grained program reasoning. (2019). Tutorial at the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2019).

Richard A. Eisenberg, Guillaume Duboc, Stephanie Weirich, and Daniel Lee. 2021. 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

Soichiro Fujii, Shin-ya Katsumata, and Paul-André Melliès. 2016. Towards a Formal Theory of Graded Monads. In *Foundations of Software Science and Computation Structures*, Bart Jacobs and Christof Löding (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 513–530.

Marco Gaboardi, Shin-ya Katsumata, Dominic A Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining effects and coeffects via grading. In *ICFP*. 476–489.

Dan R Ghica and Alex I Smith. 2014. Bounded linear types in a resource semiring. In *European Symposium on Programming Languages and Systems*. Springer, 331–350.

Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. 1992. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science* 97, 1 (1992), 1–66.

Nevin Heintze and Jon G. Riecke. 1998. The SLam Calculus: Programming with Secrecy and Integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '98)*. Association for Computing Machinery, New York, NY, USA, 365–377. https://doi.org/10.1145/268946.268976

Shin-ya Katsumata. 2014. Parametric Effect Monads and Semantics of Effect Systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 633–645.

G. A. Kavvos. 2019. Modalities, Cohesion, and Information Flow. *Proc. ACM Program. Lang.* 3, POPL, Article 20 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290333

Peng Li and Steve Zdancewic. 2010. Arrows for secure information flow. *Theoretical Computer Science* 411, 19 (2010), 1974–1994. https://doi.org/10.1016/j.tcs.2010.01.025 Mathematical Foundations of Programming Semantics (MFPS 2006).

Luísa Lourenço and Luís Caires. 2015. Dependent Information Flow Types. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 317–328. https://doi.org/10.1145/2676726.2676994

Conor McBride. 2016. *I Got Plenty o' Nuttin'*. Springer International Publishing, Cham, 207–233.

Alexandre Miquel. 2001. The Implicit Calculus of Constructions Extending Pure Type Systems with an Intersection Type Binder and Subtyping. In *Typed Lambda Calculi and Applications*, Samson Abramsky (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 344–359.

Nathan Mishra-Linger and Tim Sheard. 2008. 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* (Budapest, Hungary) *(FOSSACS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 350–364.

Richard Nathan Mishra-Linger. 2008. *Irrelevance, Polymorphism, and Erasure in Type Theory.* Ph.D. Dissertation. Portland State University, Department of Computer Science. https://doi.org/10.15760/etd.2669

Benjamin Moon, Harley Eades III, and Dominic Orchard. 2021. Graded Modal Dependent Type Theory. In *Programming Languages and Systems*, Nobuko Yoshida (Ed.). Springer International Publishing, Cham, 462–490.

Andreas Nuyts and Dominique Devriese. 2018. Degrees of Relatedness: A Unified Framework for Parametricity, Irrelevance, Ad Hoc Polymorphism, Intersections, Unions and Algebra in Dependent Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 779–788. https://doi.org/10.1145/3209108.3209119

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative Program Reasoning with Graded Modal Types. *Proc. ACM Program. Lang.* 3, ICFP, Article 110 (July 2019), 30 pages. https://doi.org/10.1145/3341714

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A calculus of context-dependent computation. In *Proceedings of International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP 2014)*.

Frank Pfenning. 2001. Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*. IEEE Computer Society, Washington, DC, USA, 221–. http://dl.acm.org/citation.cfm?id=871816.871845

Alejandro Russo, Koen Claessen, and John Hughes. 2008. A Library for Light-Weight Information-Flow Security in Haskell. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (Victoria, BC, Canada) *(Haskell '08)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/1411286.1411289

Naokata Shikuma and Atsushi Igarashi. 2006. 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* (Tokyo, Japan) *(ASIAN'06)*. Springer-Verlag, Berlin, Heidelberg, 301–315.

Geoffrey Smith and Dennis Volpano. 1998. Secure Information Flow in a Multi-Threaded Imperative Language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '98)*. Association for Computing Machinery, New York, NY, USA, 355–364. https://doi.org/10.1145/268946.268975

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with Type Equality Coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (Nice, Nice, France) *(TLDI '07)*. Association for Computing Machinery, New York, NY, USA, 53–66. https://doi.org/10.1145/1190315.1190324

Matúš Tejiščák. 2020. A Dependently Typed Calculus with Pattern Matching and Erasure Inference. *Proc. ACM Program. Lang.* 4, ICFP, Article 91 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408973

The Agda Team. 2021. https://agda.github.io/agda-stdlib/Data.Fin.Base.html

Peter Thiemann. 1997. A Unified Framework for Binding-Time Analysis. In *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT '97)*. Springer-Verlag, Berlin, Heidelberg, 742–756.

Stephen Tse and Steve Zdancewic. 2004. Translating Dependency into Parametricity. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming* (Snow Bird, UT, USA) *(ICFP '04)*. Association for Computing Machinery, New York, NY, USA, 115–125. https://doi.org/10.1145/1016850.1016868

Philip Wadler. 1998. The Marriage of Effects and Monads. *SIGPLAN Not.* 34, 1 (Sept. 1998), 63–74. https://doi.org/10.1145/291251.289429

Stephanie Weirich, Pritam Choudhury, Antoine Voizard, and Richard Eisenberg. 2019. A Role for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 3, ICFP (2019). https://doi.org/10.1145/3341705

Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A. Eisenberg. 2017. A Specification for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 1, ICFP, Article 31 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110275

Steve Zdancewic and Andrew C. Myers. 2001. Robust declassification. In *14th IEEE Computer Security Foundations Workshop (CSFW)*. 15–23. http://www.cs.cornell.edu/andru/papers/csfw01.pdf