A Dependent Dependency Calculus

Anonymous authors

No Institute Given

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

Keywords: Dependent Types · Information Flow · Irrelevance

1 Dependency Analysis

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

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

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

Dependency analysis, as we see above, is an *expressive* addition to programming languages. Such analyses allow languages to protect sensitive information (Smith and Volpano, 1998; Heintze and Riecke, 1998), support run-time code generation (Thiemann, 1997), slice programs while preserving behavior (Tip, 1995), etc. Several existing dependency analyses were unified by Abadi et al. (1999) in their Dependency Core Calculus (DCC). This calculus has served as a foundation for static analysis of dependencies in programming languages.

What makes DCC powerful is the parameterization of the type system by a *generic* lattice of dependency levels. Dependency analysis, in essence, is about ensuring secure information flow—that information never flows from more secure to less secure levels. Denning (1976) showed that a lattice model, where increasing order corresponds to higher security, can be used to enforce secure flow of information. DCC integrates this lattice model with the computational λ-calculus (Moggi, 1991) by grading the monad operator of the latter with elements of the former. This integration enables DCC to analyze dependencies in its type system.

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

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

More specifically, our work makes the following contributions:

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

2 Irrelevance and Dependent Types

Run-time irrelevance (sometimes called *erasure*) and *compile-time irrelevance* are two forms of *dependency* analyses that arise in dependent type theories. Tracking these dependencies helps compilers produce faster executables and makes type checking more flexible. (Pfenning, 2001; Miquel, 2001; Barras and Bernardo, 2008; Mishra-Linger and Sheard, 2008; Abel and Scherer, 2012; McBride, 2016; Atkey, 2018; Nuyts and Devriese, 2018; Tejiščák, 2020; Moon et al., 2021).

2.1 Run-time irrelevance

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

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

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

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

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

Consider the `Vec` datatype for length-indexed vectors, as it might look in a core language inspired by GHC (Sulzmann et al., 2007; Weirich et al., 2017). The `Vec` datatype has two parameters, `n` and `a`, that also appear in the types of the data constructors `Nil` and `Cons`. These parameters are relevant to `Vec`, but irrelevant to the data constructors. (In the types of the constructors, the equality constraints `(n ~ Zero)` and `(n ~ Succ m)` force `n` to be equal to the length of the vector.)

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

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

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

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

2.2 Compile-time Irrelevance

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

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

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

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

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

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

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

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

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

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

2.3 Strong Irrelevant Σ-types

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

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

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

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

Eisenberg et al. (2021) observe that, in Haskell, it is important to use projection functions to access the components of the pair that results from the recursive call (as in π_1 `ys` and π_2 `ys`) to ensure that `filter` is not excessively strict. If `filter` instead used pattern matching to eliminate the pair returned by the recursive call, it would have needed to filter the entire vector before returning the first successful value. This `filter` function demonstrates the practical utility of strong irrelevant Σ-types because it supports the same run-time behavior of the usual list `filter` function but with a more richly-typed data structure.

3 A Simple Dependency Analyzing Calculus

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

We call the calculus of this section SDC, for Simple Dependency Calculus. This calculus is parameterized by a lattice of *labels* or *grades*, which can also be thought of as security *levels*.[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*, which means that in a typing judgment, the derived term and every variable in the context carries a label or grade.

(Grammar)

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

Fig. 1. Simple Dependency Calculus

3.1 Type System

The typing judgment has the form $\Omega \vdash\ a\ :^\ell\ A$ which means that "ℓ is allowed to observe a" or that "a is visible at ℓ". An excerpt of the typing rules for SDC appear in Figure 2. (The rules for unit, products and sums appear in the the appendix.) Most rules are straightforward and propagate the level of the sub-terms to the expression.

The rule SDC-Var requires that the grade of the variable in the context must be less than or equal to the grade of the observer. In other words, an observer at level ℓ is allowed to use a variable from level k if and only if $k \leq \ell$. If the variable's level is too high, then this rule does not apply, ensuring that information can always flow to more secure levels but never to less secure ones.

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

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

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

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

Fig. 2. Typing rules for SDC (Excerpt)

Abstraction rule SDC-Abs uses the current level of the expression for the newly introduced variable in the context. This makes sense because the argument to the function is checked at the same level in rule SDC-App.

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

3.2 Meta-theoretic Properties

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 further, 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 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 them to the level of the judgment.

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

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

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

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

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

These lemmas allow us to prove, for a standard call-by-name small-step semantics, a standard progress and preservation based type soundness result, which we omit here.

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

3.3 A Syntactic Proof of Non-interference

When users with low-security clearance are oblivious to high-security data, we say that the system enjoys *non-interference*. Non-interference results from level-specific views of the world. The values η^H **True** and η^H **False** appear the same to an L-user while an H-user can differentiate between them. To capture this notion of a level-specific view, we design an indexed equivalence relation on open terms, $\sim_\ell$, called *indexed indistinguishability*, and shown in Figure 3. To define this relation, we need the labels of the variables in the context but not their types. So, we use grade-only contexts Φ, defined as $\Phi ::= \varnothing \mid \Phi, x : \ell$.

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

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

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

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

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

Fig. 3. Indexed indistiguishability for SDC (Excerpt)

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

Note the rule SGEq-Return in Figure 3. 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 indistinguishable at ℓ; otherwise, they may be arbitrary terms.

Now, we explore some properties of the indistinguishability relation.[2]
If we remove the second component from an indistinguishability relation, $\Phi \vdash a \sim_\ell b$, we get a new judgment, $\Phi \vdash a : \ell$, called grading judgment. Now, corresponding to every indistinguishability rule, we define a grading rule where the indistinguishability judgments have been replaced with their grading counterparts. Terms derived using these grading rules are called well-graded. We can show that well-typed terms are well-graded.

Lemma 5 (Typing implies grading[3]). *If $\Omega \vdash a :^\ell A$ then $|\Omega| \vdash a : \ell$.*

Lemma 6 (Equivalence[4]). *Indexed indistinguishability at ℓ is an equivalence relation on well-graded terms at ℓ.*

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

Lemma 7 (Indistinguishability under substitution[5]). *If $\Phi, x : \ell \vdash b_1 \sim_k b_2$ and $\Phi \vdash^{\ell}_{k} a_1 \sim a_2$ then $\Phi \vdash b_1\{a_1/x\} \sim_k b_2\{a_2/x\}$.*

With regard to the above lemma, consider the situation when $\neg(\ell \leq k)$, for example, when $\ell = H$ and $k = L$. In such a situation, for any two terms a_1 and a_2, if $\Phi, x : \ell \vdash b_1 \sim_k b_2$, then $\Phi \vdash b_1\{a_1/x\} \sim_k b_2\{a_2/x\}$. Let us work out a concrete example. For a typing derivation $x :^H A \vdash b :^L$ **Bool**, we have, by lemmas 5 and 6, $x : H \vdash b \sim_L b$. Then, $\varnothing \vdash b\{a_1/x\} \sim_L b\{a_2/x\}$. This is almost non-interference in action. What's left to show is that the indistinguishability relation respects the small step semantics, written $a_1 \rightsquigarrow a_2$. The small-step relation is standard call-by-name reduction.

[2] Because this relation is untyped, its analogue for DDC is similar. For each lemma below, we include a reference to the location in the Coq development where it may be found for the dependent system.
[3] `typing.v:Typing_Grade` [4] `geq.v:GEq_refl,GEq_symmetry,GEq_trans`
[5] `subst.v:CEq_GEq_equality_substitution`

Theorem 1 (Non-interference[6]). *If* $\Phi \vdash a_1 \sim_k a'_1$ *and* $a_1 \leadsto a_2$ *then there exists some* a'_2 *such that* $a'_1 \leadsto a'_2$ *and* $\Phi \vdash a_2 \sim_k a'_2$.

Since the step relation is deterministic, in the above lemma, there is exactly one such a'_2 that a'_1 steps to. Now, going back to our last example, we see that $b\{a_1/x\}$ and $b\{a_2/x\}$ take steps in tandem and they are L-indistinguishable after each and every step. Since the language itself is terminating, both the terms reduce to boolean values, values that are themselves L-indistinguishable as well. But the indistinguishability for boolean values is just the identity relation. This means that $b\{a_1/x\}$ and $b\{a_2/x\}$ reduce to the same value.

The indistinguishability relation gives us a syntactic method of proving non-interference for programs derived in SDC. Essentially, we show that a user with low-security clearance cannot distinguish between high security values just by observing program behavior.
Next, we show that SDC is no less expressive than the terminating fragment of DCC.

3.4 Relation with Sealing Calculus and Dependency Core Calculus

SDC is extremely similar to the sealing calculus $\lambda^{[]}$ of Shikuma and Igarashi (2006). Like SDC, $\lambda^{[]}$ has a label on the typing judgment.[7] But unlike SDC, $\lambda^{[]}$ uses standard ungraded typing contexts Γ. Both the calculi have the same types. As far as terms are concerned, there is only one difference. The sealing calculus has an **unseal** term whereas SDC uses **bind**. We present the rules for sealing and unsealing terms in $\lambda^{[]}$ below.[8]

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

Shikuma and Igarashi (2006) have shown that $\lambda^{[]}$ is equivalent to DCC_{pc}, an extension of the terminating fragment of DCC. Therefore, we compare SDC to DCC by simulating $\lambda^{[]}$ in SDC. For this, we define a translation $\overline{\cdot}$, from $\lambda^{[]}$ to SDC. Most of the cases are handled inductively in a straightforward manner. For **unseal**, we have, $\overline{\mathbf{unseal}^{\ell} a} := \mathbf{bind}^{\ell}\ x = \overline{a}\ \mathbf{in}\ x$.

With this translation, we can give a forward and a backward simulation connecting the two languages. The reduction relation $\leadsto$ below is full reduction for both the languages, the reduction strategy 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.

[6] `geq.v:CEq_GEq_respects_Step`
[7] Note that our labels correspond to observer levels of Shikuma and Igarashi (2006), which can be viewed as a lattice.
[8] We take the liberty of making small cosmetic changes in the presentation.

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

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

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

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

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

4 A Dependent Dependency Analyzing Calculus

$$
\begin{array}{rcll}
a, A, b, B & ::= & s \mid \mathbf{unit} \mid \mathbf{Unit} & \textit{sorts and unit} \\
& \mid & \Pi x\!:^{\ell} A.B \mid x \mid \lambda x\!:^{\ell} A.a \mid a\ b^{\ell} & \textit{dependent functions} \\
& \mid & \Sigma x\!:^{\ell} A.B \mid (a^{\ell}, b) \mid \mathbf{let}\ (x^{\ell}, y)\ =\ a\ \mathbf{in}\ b & \textit{dependent pairs} \\
& \mid & A + B \mid \mathbf{inj_1}\ a \mid \mathbf{inj_2}\ a \mid \mathbf{case}\ a\ \mathbf{of}\ b_1 ; b_2 & \textit{disjoint unions}
\end{array}
$$

Fig. 4. Dependent Dependency Calculus Grammar (Types and Terms)

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

Both $\mathrm{DDC}^{\top}$ and DDC are pure type systems (Barendregt, 1993). They share the same syntax, shown in Figure 4, combining terms and types into the same

[9] This assumption does not lead to any loss in generality because given a pure type system (S', A', R') that does not meet the above condition, we can provide an-

grammar. They are parameterized by a set of sorts s, a set of axioms $\mathcal{A}(s_1, s_2)$ which is a binary relation on sorts, and a set of rules $\mathcal{R}(s_1, s_2, s_3)$ which is a ternary relation on sorts. For simplicity, we assume, without loss of generality, that for every sort s_1, there is some sort s_2, such that $\mathcal{A}(s_1, s_2)$.[9]

We annotate several syntactic forms with grades for dependency analysis. The dependent function type, written $\Pi x :^{\ell} A.B$, includes the grade of the argument to a function having this type. Similarly, the dependent pair type, written $\Sigma x:^{\ell} A.B$, includes the grade of the first component of a pair having this type. [10] We can interpret these types as a fusion of the usual, ungraded dependent types and the graded modality $T^{\ell}\ A$ we saw earlier. In other words, $\Pi x :^{\ell} A.B$ acts like the type $\Pi y : (T^{\ell}\ A).\mathbf{bind}\, x = y\, \mathbf{in}\, B$ and $\Sigma x :^{\ell} A.B$ acts like the type $\Sigma y : (T^{\ell}\ A).\mathbf{bind}\, x = y\, \mathbf{in}\, B$. Because of this fusion, we do not need to add the graded modality type as a separate form—we can define $T^{\ell}\ A$ as $\Sigma x:^{\ell} A.\mathbf{Unit}$. Using $\Pi x :^{\ell} A.B$ instead of $\Pi y : (T^{\ell}\ A).\mathbf{bind}\, x = y\, \mathbf{in}\, B$ has an advantage: the former allows x to be held at differing grades while type checking B and the body of a function having this Π-type while the latter requires x to be held at the same grade in both the cases. We utilize this flexibility in Section 5.

4.1 DDC$^{\top}$: *Π*-types

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

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

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

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

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

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

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

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

The core typing rules for DDC$^{\top}$ appear in Figure 5. As in the simple type system, the variables in the context are labelled and the judgement itself includes

other pure type system (S'', A'', R''), where $S'' = S' \cup \{\text{☖}\}$ (given $\text{☖} \notin S'$) and $A'' = A' \cup \{(s, \text{☖}) | s \in S''\}$ and $R'' = R'$, such that there exists a straightforward bisimulation between the two systems.

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

a label ℓ. Rule DCT-Var is similar to its counterpart in the simply-typed language: the variable being observed must be graded less than or equal to the level of the observer. Rule DCT-Pi propagates the level of the expression to the subterms of the Π-type. Note that this type is annotated with an arbitrary label ℓ_0: the purpose of this label ℓ_0 is to denote the level at which the argument to a function having this type may be used.

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

Rule DCT-Conv converts the type of an expression to an equivalent type. The judgment $|\Omega| \vdash A \equiv_\top B$ is a label-indexed definitional equality relation instantiated to $\top$. This relation is the closure of the indexed indistinguishability relation (Section 3.3) under small-step call-by-name evaluation. When instantiated to $\top$, the relation degenerates to β-equivalence. The relation itself is untyped. As such, we need the third premise to guarantee that the new type is well-formed.

4.2 DDC$^\top$: Σ-types

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

DCT-WSigma

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

DCT-WPair

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

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

DCT-LetPair

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

The rule DCT-LetPair eliminates pairs using dependently-typed pattern matching. The pattern variables x and y are introduced into the context while checking the body c. Akin to rule SDC-Bind, the level of the first pattern

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

variable, x, is raised by ℓ_0. The result type C is refined by the pattern match, informing the type system that the pattern (x^{ℓ_0}, y) is equal to the scrutinee a.

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

$$\text{DCT-PROJ1}\ \frac{\Omega \vdash a :^{\ell} \Sigma x{:}^{\ell_0} A.B \qquad \ell_0 \leq \ell}{\Omega \vdash \pi_1^{\ell_0}\ a :^{\ell} A} \qquad\qquad \text{DCT-PROJ2}\ \frac{\Omega \vdash a :^{\ell} \Sigma x{:}^{\ell_0} A.B}{\Omega \vdash \pi_2^{\ell_0} a :^{\ell} B\{\pi_1^{\ell_0}\ a/x\}}$$

Note that the derived rule DCT-PROJ1 limits access to the first component through the premise $\ell_0 \leq \ell$, akin to rule SEALING-UNSEAL. This condition makes sense because it aligns the observability of the first component of the pair with the label on the Σ-type.

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

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

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

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

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

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

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

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

4.4 Run-time Irrelevance

Next, we show how to track run-time irrelevance using DDC$^{\top}$. We use the two element lattice $\{\perp, \top\}$ with $\perp < \top$ such that $\perp$ and $\top$ correspond to run-time relevant and run-time irrelevant terms respectively. So, we need to erase terms marked with $\top$. However, we first define a general indexed erasure function, $\lfloor \cdot \rfloor_{\ell}$,

on DDC$^\top$ terms, that erases everything an ℓ-user should not be able to see. The function is defined by straightforward recursion in most cases. For example, $\lfloor x \rfloor_\ell := x$ and $\lfloor \Pi x\!:^{\ell_0} A.B \rfloor_\ell := \Pi x\!:^{\ell_0} \lfloor A \rfloor_\ell.\lfloor B \rfloor_\ell$ and $\lfloor \lambda^{\ell_0} x.b \rfloor_\ell := \lambda^{\ell_0} x.\lfloor b \rfloor_\ell$. The interesting cases are:
$\lfloor b\ a^{\ell_0} \rfloor_\ell := (\lfloor b \rfloor_\ell\ \lfloor a \rfloor_\ell^{\ell_0})$ if $\ell_0 \leq \ell$ and $(\lfloor b \rfloor_\ell\ \mathbf{unit}^{\ell_0})$ otherwise,
$\lfloor (a^{\ell_0}, b) \rfloor_\ell := (\lfloor a \rfloor_\ell^{\ell_0}, \lfloor b \rfloor_\ell)$ if $\ell_0 \leq \ell$ and $(\mathbf{unit}^{\ell_0}, \lfloor b \rfloor_\ell)$ otherwise.
They are so defined because if $\neg(\ell_0 \leq \ell)$, an ℓ-user should not be able to see a, so we replace it with **unit**.

This erasure function is closely related to the indistinguishability relation, we saw in Section 3.3, extended to a dependent setting. (Since the extension is straightforward, we present it in Appendix C.4.) The erasure function maps the equivalence classes formed by the indistinguishability relation to their respective canonical elements.

Lemma 8 (Canonical Element[11]). *If $\Phi \vdash a_1 \sim_\ell a_2$, then $\lfloor a_1 \rfloor_\ell = \lfloor a_2 \rfloor_\ell$.*

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

Lemma 9 (Erasure Indistinguishability[12]). *If $\Phi \vdash a : \ell$, then $\Phi \vdash a \sim_\ell \lfloor a \rfloor_\ell$.*

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

Lemma 10 (Erasure Simulation[13]). *If $\Phi \vdash a : \ell$ and $a \rightsquigarrow b$, then $\lfloor a \rfloor_\ell \rightsquigarrow \lfloor b \rfloor_\ell$. Otherwise, if a is a value, then so is $\lfloor a \rfloor_\ell$.*

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

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

5 DDC: Run-time and Compile-time Irrelevance

5.1 Towards Compile-time Irrelevance

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

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

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

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

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

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

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

To move out of this impasse, we take inspiration from EPTS (Mishra-Linger and Sheard, 2008; Mishra-Linger, 2008). The key idea, adapted from Mishra-Linger and Sheard (2008), is to use a judgment of the form $C \wedge \Omega \vdash a :^{C} A$ instead of a judgment of the form $\Omega \vdash a :^{\top} A$. The operation $C \wedge \Omega$ takes the point-wise meet of the labels in the context Ω with C, essentially reducing any label marked as $\top$ to C, making it available for use in a C-expression. This operation, called *truncation*, makes $\top$ marked variables available at C.

5.2 DDC: Basics

Next, we design a general dependency analyzing calculus, DDC, that takes advantage of compile-time irrelevance in its type system. DDC is a generalization of DDC$^\top$ and EPTS$^\bullet$ (Mishra-Linger and Sheard, 2008). When C equals $\top$, DDC degenerates to DDC$^\top$, that does not use compile-time irrelevance. When C equals $\bot$, DDC degenerates to EPTS$^\bullet$, that identifies compile-time and run-time irrelevance. A crucial distinction between EPTS$^\bullet$ and DDC is that while the former is tied to a two element lattice, the latter can use any lattice. Thus, not only can DDC distinguish between run-time and compile-time irrelevance, but also it can simultaneously track other dependencies.

The core typing rules of DDC appear 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

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

$$\frac{x :^{\ell_0} A \in \Omega \quad \ell_0 \leq \ell \quad \ell \leq C}{\Omega \vdash x :^{\ell} A}\ \textsc{T-Var}$$

$$\frac{\ell \leq C \quad \mathcal{A}(s_1, s_2)}{\Omega \vdash s_1 :^{\ell} s_2}\ \textsc{T-Type}$$

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

$$\frac{\Omega, x :^{\ell_0 \vee \ell} A \vdash b :^{\ell} B \quad \Omega \Vdash (\Pi x :^{\ell_0} A.B) :^{\top} s}{\Omega \vdash \lambda x :^{\ell_0} A.b :^{\ell} \Pi x :^{\ell_0} A.B}\ \textsc{T-AbsC}$$

$$\frac{\Omega \vdash b :^{\ell} \Pi x :^{\ell_0} A.B \quad \Omega \Vdash a :^{\ell_0 \vee \ell} A}{\Omega \vdash b\ a^{\ell_0} :^{\ell} B\{a/x\}}\ \textsc{T-AppC}$$

$$\frac{\Omega \vdash a :^{\ell} A \quad |C \wedge \Omega| \vdash A \equiv_C B \quad \Omega \Vdash B :^{\top} s}{\Omega \vdash a :^{\ell} B}\ \textsc{T-ConvC}$$

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

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

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

Fig. 6. Dependent type system with compile-time irrelevance (core rules)

at $\top$ in DDC. We can get around this restriction by deriving $C \wedge \Omega \vdash a :^{C} A$ in place of $\Omega \vdash a :^{\top} A$.

Wherever DDC$^{\top}$ uses $\top$ as the observer level on a typing judgment, DDC uses truncation and level C instead. If DDC$^{\top}$ uses some grade other than $\top$ as the observer level, DDC leaves the derivation as such. So a DDC$^{\top}$ judgment $\Omega \vdash a :^{\ell} A$ is replaced with a *truncated-at-top judgment*, $\Omega \Vdash a :^{\ell} A$ which can be read as: if $\ell = \top$, use the truncated version $C \wedge \Omega \vdash a :^{C} A$; otherwise use the normal version $\Omega \vdash a :^{\ell} A$, as we see in Figure 6. In the typing rules, uses of this new judgment have been highlighted in gray to emphasize the modification with respect to DDC$^{\top}$.

5.3 Π-types

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

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

```
id : Π x:⊤Type. x -> x
```

```
id = λ⊤x. λ y. y
```

In DDC$^\top$, the type Π `x:`$^\top$`Type. x -> x` is checked at $\top$. However, here it must be checked at level C, which requires the premise `x:`^{C}`Type` $\vdash$ `x -> x :`^{C} `Type`. Note that if we used the same grade for the bound variable x in rule T-Pi and rule T-AbsC, we would have been in trouble because variable `x` is compile-time relevant while we check the type, even though it is irrelevant in the term.[14]

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

5.4 Σ-types

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

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

We also need to modify the typing rules for Σ types accordingly. In particular, when we create a pair, we check the first component using the truncated-at-top judgment. This is akin to how we check the argument in rule T-AppC. Note that if $\ell_0 = \top$, the first component a is compile-time irrelevant. In such a situation, we cannot type-check the second projection since it requires the first projection, as we see in the derived[15] projection rules below. So pairs having type $\Sigma x{:}^{\top} A.B$ can only be eliminated via pattern matching if B mentions x. However, pairs having type $\Sigma x{:}^{C} A.B$ can be eliminated via projections.

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

$$\textsc{T-Proj1}\quad \frac{\Omega \vdash a :^{\ell} (x{:}^{\ell_0} A) \,\&\, B \qquad \ell_0 \leq \ell}{\Omega \vdash \pi_1^{\ell_0}\ a :^{\ell} A}$$

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

5.5 Consistency of Equality

The equality relation of DDC incorporates compile-time irrelevance. To show that the type system is sound, we need to show that the equality relation is consistent. Consistency of definitional equality means that there is no derivation

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

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

that equates two types having different head forms. For example, it should not equate **Nat** with **Unit**.

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

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

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

Theorem 8 (Consistency[16]). *If* $\Phi \vdash a \equiv_{\ell} b$*, and* a *and* b *both are head forms, then they have the same head form.*

5.6 Soundness theorem

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

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

Lemma 11 (Narrowing[17]). *If* $\Omega \vdash a :^{\ell} A$ *and* $\Omega' \leq \Omega$*, then* $\Omega' \vdash a :^{\ell} A$

Lemma 12 (Weakening[18]). *If* $\Omega_1, \Omega_2 \vdash a :^{\ell} A$ *then* $\Omega_1, \Omega, \Omega_2 \vdash a :^{\ell} A$.

Lemma 13 (Restricted upgrading[19]). *If* $\Omega_1, x :^{\ell_0} A, \Omega_2 \vdash b :^{\ell} B$ *and* $\ell_1 \leq \ell$ *then* $\Omega_1, x :^{\ell_0 \vee \ell_1} A, \Omega_2 \vdash b :^{\ell} B$.

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

[16] `consist.v:DefEq_Consistent` [17] `narrowing.v:Typing_narrowing`
[18] `weakening.v:Typing_weakening` [19] `pumping.v:Typing_pumping`

Lemma 14 (Bounded by C[20]). *If $\Omega \vdash a :^{\ell} A$ then $\ell \leq C$.*

Lemma 15 (Subsumption[21]). *If $\Omega \vdash a :^{\ell} A$ and $\ell \leq k$ and $k \leq C$ then $\Omega \vdash a :^{k} A$*

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

Lemma 16 (Substitution[22]). *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 17 (Regularity[23]). *If $\Omega \vdash a :^{\ell} A$ and $\vdash \Omega$ then $\Omega \Vdash A :^{\top} s$.*

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

Lemma 18 (Preservation[24]). *If $\Omega \vdash a :^{\ell} A$ and $\vdash \Omega$ and $a \rightsquigarrow a'$, then $\Omega \vdash a' :^{\ell} A$.*

Lemma 19 (Progress[25]). *If $\varnothing \vdash a :^{\ell} A$ then either a is a value or there exists some a' such that $a \rightsquigarrow a'$.*

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

Note that DDC is parameterized by a generic pure type system and a generic lattice. When the parameterizing pure type system is strongly normalizing, such as the Calculus of Constructions, DDC can support decidable type checking either via bidirectionality or through additional type annotations. Such revisions to DDC would not be difficult and would have little interaction with dependency analysis. For parameterizing pure type systems where type equality is undecidable, such as the Type-in-Type language, we would first need to annotate the equality relation with coercions to make it decidable, for example, as in Weirich et al. (2017). Thereafter, we can follow the same method to support decidable type-checking.

6 Graded Types

A type system can be explicitly monadic by having an explicit type for the monad. In such a system a typing judgment of a type with side effects has the form $\Gamma \vdash t : \mathsf{m}A$. We can program in the world with all the side effects given to us by the monad by applying the monad to all of our types; e.g., judgments of

[20] `pumping.v:Typing_leq_C` [21] `typing.v:Typing_subsumption`
[22] `typing.v:Typing_substitution_CTyping` [23] `typing.v:Typing_regularity`
[24] `typing.v:Typing_preservation` [25] `progress.v:Typing_progress`

the form $x_1 : \mathsf{m}A_1, \ldots, x_i : \mathsf{m}A_i \vdash t : \mathsf{m}A$. In fact, this offers up an alternative design where we leave off the explicit application of the monad, but with every type being endowed with the side effects of the monad.

Moving over to categorical models of type systems a system with an explicit monad can be modeled by a category and a monad over the category; e.g., a tuple $(\mathcal{C}, \mathsf{m}, \eta, \mu)$ where the $\mathsf{m} : \mathcal{C} \longrightarrow \mathcal{C}$ is a functor, $\delta : A \longrightarrow \mathsf{m}A$ is the return of the monad, $\mu : \mathsf{m}^2 A \longrightarrow \mathsf{m}A$ is the join of the monad, and this structure is subject to several coherence laws called the monad laws[26]. Every monad is related to its category of algebras (often called the Eilenberg-Moore Category) $\mathcal{C}^{\mathsf{m}}$ which consists of objects pairs (A, h) called an algebra where A is an object of $\mathcal{C}$ and $h : \mathsf{m}A \longrightarrow A$ is an action on A. The model of implicitly monadic systems is the category of algebras for the monad. Finally, these two systems are related through an adjunction $\mathcal{C}^{\mathsf{m}} : \mathsf{U} \dashv \mathsf{F} : \mathcal{C}$ where the functor $\mathsf{U} : \mathcal{C}^{\mathsf{m}} \longrightarrow \mathcal{C}$ forgets the action of an algebra and the functor $\mathsf{F} : \mathcal{C} \longrightarrow \mathcal{C}^{\mathsf{m}}$ sends an object A to the free algebra $(\mathsf{m}A, \mu_A)$. This implies that each design–explicitly or implicitly monadic–result in expressively equivalent systems, but with alternative perspectives.

In this section, we look at our work from this perspective and begin to understand how the systems we present here are related to graded types. We can already divide the work on graded type systems up into explicitly graded and implicitly graded systems. The first contains all of the type systems with an explicit graded (co)monad such as the systems from Orchard et al. (2019); Gaboardi et al. (2016); Petricek et al. (2014); Ghica and Smith (2014). These systems only label variables that were introduced through the (co)monad with a grade. Implicitly graded type systems label every type with a grade such as the systems from McBride (2016); Atkey (2018); Choudhury et al. (2021); Moon et al. (2021). We might call these implicitly graded type systems *fully graded type systems.*

Now are these two perspectives related in the similar way that they are for monads? We conjecture that the answer is positive, but we are not currently able to answer this conjecture in full generality. Instead we show that the categories of algebras for graded 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.

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.

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

[26] We omit the various coherence laws from our discussion.

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 $\mathsf{approx}(f) : \langle (X^1, r'_1), \ldots, (X^n, r'_n) \rangle \longrightarrow (Z, s')$[27].

The category of graded algebras due to Fujii et al. (2016) 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 indeed 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 2. *Suppose $\mathcal{L}$ is a set of labels. The category of classified sets $\mathsf{CSet}(\mathcal{L})$ has the following data. Objects are classified sets S (functors $S : \mathcal{L} \longrightarrow \mathsf{RRel}$ 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 20 (Graded-Indexed Multicategory of Relations). *Suppose $(L, \leq, \top, \bot, \sqcup, \sqcap)$ is a lattice. Then there is a grade-indexed multicategory $\mathsf{Gr}(L, \mathsf{CSet}(L))$.*

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. In fact, we get all of the necessary structure to soundly interpret SDC in classified sets.

Lemma 21 (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 $[\![a]\!] = [\![b]\!]$.*

[27] There are a number of coherence axioms necessary that we omit here, but can be

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.

7 Discussions and Related Work

7.1 Irrelevance in Dependent Type Theories

Overall, compile-time and run-time irrelevance is a well-studied topic in the design of dependent type systems. In some systems, the focus is only on support for run-time irrelevance: see McBride (2016); Atkey (2018); Brady (2021); Miquel (2001); Mishra-Linger and Sheard (2008); Tejiščák (2020). In other systems, the focus is on compile-time irrelevance: see Pfenning (2001); Abel and Scherer (2012). Some systems support both, but require them to overlap, such as Barras and Bernardo (2008); Mishra-Linger (2008); Weirich et al. (2017); Nuyts and Devriese (2018). The system of Moon et al. (2021) does not require them to overlap but their type system does not make use of compile-time irrelevance in the conversion rule.

To compare, system $\mathrm{DDC}^{\top}$, presented here, can support run-time irrelevance only and is similar to the core language of Tejiščák (2020). However, note that $\mathrm{DDC}^{\top}$ can track dependencies in general while the system in Tejiščák (2020) tracks run-time irrelevance alone. DDC, on the other hand, is the only system that we are aware of that tracks run-time and compile-time irrelevance separately and makes use of the latter in the conversion rule. Further, DDC tracks these irrelevances in the presence of strong Σ-types with erasable first components, something which, to the best of our knowledge, no prior work has been able to.

Prior work has identified the difficulty in handling strong Σ-types with erasable first components in a setting that tracks compile-time irrelevance. Abel and Scherer (2012) point out that strong irrelevant Σ-types make their theory inconsistent. Similarly, $\mathrm{EPTS}^{\bullet}$ (Mishra-Linger, 2008) cannot define the projections for pairs having such Σ-types. The reason behind this is that $\mathrm{EPTS}^{\bullet}$ is hard-wired to work with a two-element lattice which identifies compile-time and run-time irrelevance. As such, projections from such pairs lead to type unsoundness. For example, considering the first components to be run-time irrelevant, the pairs $(\mathbf{Int}, \mathbf{unit})$ and $(\mathbf{Bool}, \mathbf{unit})$ are run-time equivalent. Since $\mathrm{EPTS}^{\bullet}$ identifies run-time and compile-time irrelevance, these pairs are also compile-time equivalent. Then, taking the first projections of these pairs, one ends up with **Int** and **Bool** being compile-time equivalent. We resolve this problem by distinguishing between run-time and compile-time irrelevance, thus requiring a lattice with three elements.

found in Appendix A.

Next, we compare our work with existing literature with respect to the equality relation. We analyze compile-time irrelevance to enable the equality relation to ignore unnecessary sub-terms. However, since our equality relation is untyped, we cannot include type-dependent rules in our system, such as η-equivalence for the `Unit` type. Several prior works on irrelevance (Miquel, 2001; Barras and Bernardo, 2008; Mishra-Linger, 2008; Tejiščák, 2020) use an untyped equality relation. However, some prior work, such as Pfenning (2001); Abel and Scherer (2012), do consider compile-time irrelevance in the context of typed-directed equality. But such systems require irrelevant arguments to functions appear only irrelevantly in the codomain type of the function, thus ruling out several examples including the polymorphic identity function.

7.2 Quantitative Type Systems

Our work is closely related to quantitative type systems (Petricek et al., 2014; Ghica and Smith, 2014; Brunel et al., 2014; McBride, 2016; Atkey, 2018; Orchard et al., 2019; Abel and Bernardy, 2020; Choudhury et al., 2021; Moon et al., 2021). Such systems provide a fine-grained accounting of coeffects, viewed as resources, for example, variable usage, linearity, liveness, etc. A typical judgment from a quantitative type system (Choudhury et al., 2021) may look like:

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

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

This form of judgment is very similar to our typing judgments with quantities appearing in place of levels. However, there is a crucial difference: to the right of the turnstile, while any level may appear in our judgments, only the quantity 1 can appear in typing judgments of quantitative systems. A quantitative system that allows an arbitrary quantity to the right of the turnstile is not closed under substitution (McBride, 2016; Atkey, 2018). As such, quantitative systems are tied to a fixed reference while our systems can view programs from different reference levels. This difference in form results from the difference in the purposes the two kinds of systems serve: quantitative systems count while our systems compare. Counting requires a fixed standard or reference whereas comparison does not. Applications that require counting, like linearity tracking, are handled well by quantitative systems while applications that require comparison, like ensuring secure information flow, are handled well by systems of our kind.

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

7.3 Dependency Analysis and Dependent Type Theory

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

Like our system, Prost (2000) extends the λ-cube so that it may track dependencies. However, unlike our system, this work uses sorts to track dependencies. It is inspired by the distinction between sorts in the Calculus of Constructions where computationally relevant and irrelevant terms live in sorts `Set` and `Prop` respectively. As Mishra-Linger (2008) points out, such an approach ties up two distinct language features, sorts and dependency analysis, which can be treated in a more orthogonal manner.

Bernardy and Guilhem (2013) is very related to our work. They use colors to erase terms while we use grades. Colors and grades both form a lattice structure and their usage in the respective type systems are quite similar. However, 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 analyzing erasure.

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

8 Conclusion

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

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

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

Bibliography

Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.: A core calculus of dependency. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, p. 147–160, POPL '99, Association for Computing Machinery, New York, NY, USA (1999), ISBN 1581130953, , URL `https://doi.org/10.1145/292540.292555`

Abel, A., Bernardy, J.P.: A unified view of modalities in type systems. Proc. ACM Program. Lang. **4**(ICFP) (Aug 2020), , URL `https://doi.org/10.1145/3408972`

Abel, A., Scherer, G.: On irrelevance and algorithmic equality in predicative type theory. Logical Methods in Computer Science **8**(1) (mar 2012), , URL `https://doi.org/10.2168%2Flmcs-8%281%3A29%292012`

Atkey, R.: Syntax and semantics of quantitative type theory. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, p. 56–65, LICS '18, Association for Computing Machinery, New York, NY, USA (2018), ISBN 9781450355834, , URL `https://doi.org/10.1145/3209108.3209189`

Barendregt, H.P.: Lambda Calculi with Types, p. 117–309. Oxford University Press, Inc., USA (1993), ISBN 0198537611

Barras, B., Bernardo, B.: The implicit calculus of constructions as a programming language with dependent types. In: Amadio, R. (ed.) Foundations of Software Science and Computational Structures, pp. 365–379, FOSSACS 2008, Springer Berlin Heidelberg, Budapest, Hungary (2008)

Bernardy, J.P., Guilhem, M.: Type-theory in color. SIGPLAN Not. **48**(9), 61–72 (Sep 2013), ISSN 0362-1340, , URL `https://doi.org/10.1145/2544174.2500577`

Brady, E.: Idris 2: Quantitative Type Theory in Practice. In: Møller, A., Sridharan, M. (eds.) 35th European Conference on Object-Oriented Programming (ECOOP 2021), Leibniz International Proceedings in Informatics (LIPIcs), vol. 194, pp. 9:1–9:26, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021), ISBN 978-3-95977-190-0, ISSN 1868-8969, , URL `https://drops.dagstuhl.de/opus/volltexte/2021/14052`

Brunel, A., Gaboardi, M., Mazza, D., Zdancewic, S.: A core quantitative coeffect calculus. In: Shao, Z. (ed.) Programming Languages and Systems, pp. 351–370, Springer Berlin Heidelberg, Berlin, Heidelberg (2014)

Choudhury, P., Eades III, H., Eisenberg, R.A., Weirich, S.: A graded dependent type system with a usage-aware semantics. Proc. ACM Program. Lang. **5**(POPL) (Jan 2021), , URL `https://doi.org/10.1145/3434331`

Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (May 1976), ISSN 0001-0782, , URL `https://doi.org/10.1145/360051.360056`

Eisenberg, R.A., Duboc, G., Weirich, S., Lee, D.: An existential crisis resolved: Type inference for first-class existential types. Proc. ACM Program. Lang.

5(ICFP) (Aug 2021), URL `https://richarde.dev/papers/2021/exists/exists.pdf`
Fujii, S., Katsumata, S.y., Melliès, P.A.: Towards a formal theory of graded monads. In: Jacobs, B., Löding, C. (eds.) Foundations of Software Science and Computation Structures, pp. 513–530, Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
Gaboardi, M., Katsumata, S.y., Orchard, D.A., Breuvart, F., Uustalu, T.: Combining effects and coeffects via grading. In: ICFP, pp. 476–489 (2016)
Ghica, D.R., Smith, A.I.: Bounded linear types in a resource semiring. In: European Symposium on Programming Languages and Systems, pp. 331–350, Springer (2014)
Heintze, N., Riecke, J.G.: The slam calculus: Programming with secrecy and integrity. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, p. 365–377, POPL '98, Association for Computing Machinery, New York, NY, USA (1998), ISBN 0897919793, , URL `https://doi.org/10.1145/268946.268976`
Kavvos, G.A.: Modalities, cohesion, and information flow. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019), , URL `https://doi.org/10.1145/3290333`
Lourenço, L., Caires, L.: Dependent information flow types. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, p. 317–328, POPL '15, Association for Computing Machinery, New York, NY, USA (2015), ISBN 9781450333009, , URL `https://doi.org/10.1145/2676726.2676994`
McBride, C.: I Got Plenty o' Nuttin', pp. 207–233. Springer International Publishing, Cham (2016)
Miquel, A.: The implicit calculus of constructions extending pure type systems with an intersection type binder and subtyping. In: Abramsky, S. (ed.) Typed Lambda Calculi and Applications, pp. 344–359, Springer Berlin Heidelberg, Berlin, Heidelberg (2001), ISBN 978-3-540-45413-7
Mishra-Linger, N., Sheard, T.: Erasure and polymorphism in pure type systems. In: Proceedings of the Theory and Practice of Software, 11th International Conference on Foundations of Software Science and Computational Structures, p. 350–364, FOSSACS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008), ISBN 3540784977
Mishra-Linger, R.N.: Irrelevance, Polymorphism, and Erasure in Type Theory. Ph.D. thesis, Portland State University, Department of Computer Science (2008),
Moggi, E.: Notions of computation and monads. Information and Computation **93**(1), 55–92 (1991), ISSN 0890-5401, URL `https://www.sciencedirect.com/science/article/pii/0890540191900524`, selections from 1989 IEEE Symposium on Logic in Computer Science
Moon, B., Eades III, H., Orchard, D.: Graded modal dependent type theory. In: Yoshida, N. (ed.) Programming Languages and Systems, pp. 462–490, Springer International Publishing, Cham (2021), ISBN 978-3-030-72019-3
Nuyts, A., Devriese, D.: Degrees of relatedness: A unified framework for parametricity, irrelevance, ad hoc polymorphism, intersections, unions and algebra in dependent type theory. In: Dawar, A., Grädel, E. (eds.) Proceedings

of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018, pp. 779–788, ACM (2018), , URL `https://doi.org/10.1145/3209108`

Orchard, D., Liepelt, V.B., Eades III, H.: Quantitative program reasoning with graded modal types. Proc. ACM Program. Lang. **3**(ICFP) (Jul 2019), , URL `https://doi.org/10.1145/3341714`

Petricek, T., Orchard, D., Mycroft, A.: Coeffects: A calculus of context-dependent computation. In: Proceedings of International Conference on Functional Programming, ICFP 2014 (2014)

Pfenning, F.: Intensionality, extensionality, and proof irrelevance in modal type theory. In: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science, pp. 221–, LICS '01, IEEE Computer Society, Washington, DC, USA (2001), URL `http://dl.acm.org/citation.cfm?id=871816.871845`

Prost, F.: A static calculus of dependencies for the λ-cube. In: Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332), pp. 267–276 (2000),

Shikuma, N., Igarashi, A.: Proving noninterference by a fully complete translation to the simply typed λ-calculus. In: Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues, p. 301–315, ASIAN'06, Springer-Verlag, Berlin, Heidelberg (2006), ISBN 3540775048

Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, p. 355–364, POPL '98, Association for Computing Machinery, New York, NY, USA (1998), ISBN 0897919793, , URL `https://doi.org/10.1145/268946.268975`

Sulzmann, M., Chakravarty, M.M.T., Jones, S.P., Donnelly, K.: System f with type equality coercions. In: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, p. 53–66, TLDI '07, Association for Computing Machinery, New York, NY, USA (2007), ISBN 159593393X, , URL `https://doi.org/10.1145/1190315.1190324`

Tejiščák, M.: A dependently typed calculus with pattern matching and erasure inference. Proc. ACM Program. Lang. **4**(ICFP) (Aug 2020), , URL `https://doi.org/10.1145/3408973`

Thiemann, P.: A unified framework for binding-time analysis. In: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development, p. 742–756, TAPSOFT '97, Springer-Verlag, Berlin, Heidelberg (1997), ISBN 3540627812

Tip, F.: A survey of program slicing techniques. Journal of Programming Languages **3** (1995)

Weirich, S., Voizard, A., de Amorim, P.H.A., Eisenberg, R.A.: A specification for dependent types in Haskell. Proc. ACM Program. Lang. **1**(ICFP), 31:1–31:29 (Aug 2017), ISSN 2475-1421, , URL `http://doi.acm.org/10.1145/3110275`

A Relating Graded Types with Qualities

At the base of the categorical models for graded types either coeffectful or effectful is the notion of a *grade-indexed multicategory*.

Definition 3. *Suppose $(\mathcal{R}, \leq)$ is a preorder and $\mathcal{M}$ is a class of objects. Then a* grade-indexed multicategory $\mathsf{Gr}(\mathcal{R}, \mathcal{M})$ *consists of the following data:*

1. *(Objects) pairs X_r where $X \in \mathcal{M}$ and $r \in \mathcal{R}$.*
2. *(Morphisms) Morphisms are of the form $\langle X^1_{r_1}, \ldots, X^n_{r_n}\rangle \xrightarrow{f} Y_r$ where $\langle X^1_{r_1}, \ldots, X^n_{r_n}\rangle$ is a vector of objects.*
   - *(a) Identity morphisms $\langle X_r\rangle \xrightarrow{\mathsf{id}} X_r$.*
   - *(b) The composition of morphisms:*

$$\langle X^{11}_{r_{11}}, \ldots, X^{1n_1}_{r_{1n_1}}\rangle \xrightarrow{f_1} Y^1_{s_1}, \ldots, \langle X^{m1}_{r_{m1}}, \ldots, X^{mn_m}_{r_{mn_m}}\rangle \xrightarrow{f_m} Y^m_{s_m}$$
$$\langle Y^1_{s_1}, \ldots, Y^m_{s_m}\rangle \xrightarrow{g} Z_r$$

   is a morphism:

$$\langle X^{11}_{r_{11}}, \ldots, X^{1n_1}_{r_{1n_1}}, X^{m1}_{r_{m1}}, \ldots, X^{mn_m}_{r_{mn_m}}\rangle \xrightarrow{g(f_1,\ldots,f_m)} Z_r$$

   - *(c) (Approximation) There is a natural transformation:*

$$\frac{s \leq s', r'_1 \leq r_1, \ldots, r'_n \leq r_n \qquad \langle X^1_{r_1}, \ldots, X^n_{r_n}\rangle \xrightarrow{f} Z_s}{\langle X^1_{r'_1}, \ldots, X^n_{r'_n}\rangle \xrightarrow{\mathsf{approx}(r'_1,\ldots,r'_n,f)} Z_{s'}}$$

Definition 4. *Suppose $\mathcal{L}$ is a set of labels. The category of classified sets* $\mathsf{CSet}(L)$ *has the following data. Objects are classified sets S which are functors $S_X : \mathcal{L} \longrightarrow \mathsf{RRel}$ from the discrete category 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 $S(l) \xrightarrow{h_1} S'(l)$ in RRel, and thus, have the previous property.

Lemma 22 (Graded-Indexed Multicategory of Relations). *Suppose $(L, \leq, \top, \bot, \sqcup, \sqcap)$ is a lattice. Then there is a grade-indexed multicategory $\mathsf{Gr}(L, \mathsf{CSet}(L))$.*

Proof. Define an object to be a pair (S,l) for every object S in $\mathsf{CSet}(L)$ and label $l \in L$. Given two relations R_X and R_Y we can define the relation $R_X \times R_Y$ as

$$(x,y)(R_X \times R_Y)(x',y') = x\, R_X\, x' \wedge y\, R_Y, y'$$

and there are morphisms:

$$\begin{array}{l} \pi_1 : R_X \times R_Y \longrightarrow R_X \\ \pi_2 : R_X \times R_Y \longrightarrow R_Y \\ (f,g) : R_Z \longrightarrow R_X \times R_Y \end{array}$$

making the standard diagram commute. The previous morphisms are inherited from the structure of Set which is the bases of the morphisms in RRel. We now use this cartesian product on relations to define multimorphisms. A multimorphism $\langle (S_1,l_1),\ldots,(S_n,l_n)\rangle \xrightarrow{f} (S,l)$ is the natural transformation:

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

where $(S_1\times\cdots\times S_n)\xrightarrow{f} S$ is a morphism in $\mathsf{CSet}(L)$, $a : (l_1\sqcup\cdots\sqcup l_n) \leq l$, $a_i : l_i \leq (l_1 \sqcup\cdots\sqcup l_i \sqcup\cdots\sqcup l_n)$ for $1 \leq i \leq n$, and $S_1 \times S_2 = \lambda l'.S_1(l') \times S_2(l')$. Notice that we do not have a morphism $\mathsf{id}_{S_1(l_1)\times S_2(l_2)} : \langle (S_1,l_1),(S_2,l_2)\rangle \longrightarrow S_1(l_1) \times S_2(l_2)$ making this a non-representable multicategory. Approximation comes from the fact that we have a morphism $S(l_1)\xrightarrow{S(a)} S(l_2)$ when $a : l_1 \leq l_2$ due to the fact that $S : L \longrightarrow \mathsf{RRel}$ is a functor.

Lets take a look at the structure of $\mathsf{Gr}(L,\mathsf{CSet}(L))$. First, we can define the graded cartesian product:

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

This definition implies the following:

$$\frac{\langle (S_1,l_1),\ldots,(S_i,l_i),(S_{i+1},l_i),\ldots,(S_n,l_n)\rangle \xrightarrow{f} (S,l)}{\langle (S_1,l_1),\ldots,(S_i,l_i)\times(S_{i+1},l_i),\ldots,(S_n,l_n)\rangle \xrightarrow{\mathsf{prodl}(f)} (S,l)}$$

where $\mathsf{prod}(f) = f$ because $\mathsf{Gr}(L,\mathsf{CSet}(L))$. 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 $\mathsf{prodr}(f_1,f_2) = f_1 \times f_2$.

The internal-hom is defined as follows:

$$S_1 \Rightarrow S_2 := \lambda l.\lambda f.\lambda g.a\, S_1(l)\, b \implies f(a)\, S_2(l)\, g(b)$$

This implies the following bijection:

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

Since $\mathsf{Gr}(L, \mathsf{CSet}(L))$ is indeed cartesian closed we have the following structural rules:

$$\langle (S, l) \rangle \xrightarrow{\mathsf{contract}} (S, l) \times (S, l)$$
$$\langle (S, l) \rangle \xrightarrow{\mathsf{weak}} \top$$

where $\top := \lambda l.\mathsf{true}$ is the singleton classified set, and hence, is terminal.

The final element of the model is a grade actor which we define to be the join of the lattice. Using this we define an action:

$$(S, l') \odot l = (S, l' \sqcup l)$$

Furthermore, we can define an adjunction. First, we have the following forgetful functor:

$$\begin{array}{l} \mathsf{Forget} : \mathsf{Gr}(L, \mathsf{CSet}(L)) \longrightarrow \mathsf{CSet}(L) \\ \mathsf{Forget}(S, l) = S(- \sqcup l) \\ \mathsf{Forget}(f : \langle (S_1, l_1), \ldots, (S_n, l_n) \rangle \longrightarrow (S, l)) = f : S_1(- \sqcup l_1) \times \ldots \times S_n(- \sqcup l_n) \longrightarrow S(- \sqcup l) \end{array}$$

and we have the following free functor:

$$\begin{array}{l} \mathsf{Free} : \mathsf{CSet}(L) \longrightarrow \mathsf{Gr}(L, \mathsf{CSet}(L)) \\ \mathsf{Free}(S) = (S, \bot) \\ \mathsf{Free}(f : S_1 \longrightarrow S_2) = f : \langle (S_1, \bot) \rangle \longrightarrow (S_2, \bot) \end{array}$$

We can now prove that there is an adjunction $\mathsf{CSet}(L) : \mathsf{Free} \dashv \mathsf{Forget} : \mathsf{Gr}(L, \mathsf{CSet}(L))$. We have:

$$\begin{aligned} \mathsf{Forget}(\mathsf{Free}(S)) &= \mathsf{Forget}(S, \bot) && (1) \\ &= S(- \sqcup \bot) && (2) \end{aligned}$$

Then we have:

$$\begin{aligned} \mathsf{Free}(\mathsf{Forget}(S, l)) &= \mathsf{Free}(S(- \sqcup l)) && (3) \\ &= (S(- \sqcup l), \bot) && (4) \end{aligned}$$

Theorem 9 (Free Forgetful Adjunction). *There is an adjunction:*

$$\mathsf{CSet}(L) : \mathsf{Free} \dashv \mathsf{Forget} : \mathsf{Gr}(L, \mathsf{CSet}(L))$$

Proof. To have a morphism $\eta_S : S \longrightarrow \mathsf{Forget}(\mathsf{Free}(S))$ is to have a natural transformation $\eta_S : S \longrightarrow S(- \sqcup \bot)$ which we can define as the coordinate $\hat{\eta}_{S(l)} = \mathsf{id}_{S(l)} : S(l) \longrightarrow S(l)$ because we know $l = l \sqcup \bot$ for any l.

To have a morphism $\varepsilon_{(S,l)} : \langle \mathsf{Free}(\mathsf{Forget}(S,l)) \rangle \longrightarrow (S,l)$ means we must have a morphism $\varepsilon_{(S,l)} : \langle (S(- \sqcup l), \perp) \rangle \longrightarrow (S,l)$, but this is equivalent to $\hat{\varepsilon}_{(S,l)} : S(\perp \sqcup l) \longrightarrow S(l)$. And, again $\perp \sqcup l = l$, and thus, set $\hat{\varepsilon}_{(S,l)} = \mathsf{id}_{S(l)}$.

Finally, we have:

$$\mathsf{Free}(S) \xrightarrow{\mathsf{Free}(\eta_S)} \mathsf{Free}(\mathsf{Forget}(\mathsf{Free}(S))) \tag{5}$$

$$= \mathsf{Free}(S) \xrightarrow{\mathsf{id}_{\mathsf{Free}(S)}} \mathsf{Free}(S) \tag{6}$$

$$= \langle \mathsf{Free}(\mathsf{Forget}(\mathsf{Free}(S))) \rangle \xrightarrow{\varepsilon_{\mathsf{Free}(S)}} \mathsf{Free}(S) \tag{7}$$

and

$$\mathsf{Forget}(S,l) \xrightarrow{\eta_{\mathsf{Forget}(S,l)}} \mathsf{Forget}(\mathsf{Free}(\mathsf{Forget}(S,l))) \tag{8}$$

$$= \mathsf{Forget}(S,l) \xrightarrow{\mathsf{id}_{\mathsf{Forget}(S,l)}} \mathsf{Forget}(S,l) \tag{9}$$

$$= \langle \mathsf{Forget}(\mathsf{Free}(\mathsf{Forget}(S,l))) \rangle \xrightarrow{\mathsf{Forget}(\varepsilon_{(S,l)})} \mathsf{Forget}(S,l) \tag{10}$$

Since both η and ε are both identities it is clear they are natural transformations. Therefore, $\mathsf{CSet}(L) : \mathsf{Free} \dashv \mathsf{Forget} : \mathsf{Gr}(L, \mathsf{CSet}(L))$ is indeed an adjunction.

Now the previous adjunction along with the previously defined action implies there is a graded monad:

$$\begin{aligned} T : L \times \mathsf{CSet}(L) &\longrightarrow \mathsf{CSet}(L) && (11)\\ T_l(S) &= \mathsf{Forget}(\mathsf{Free}(S) \odot l) && (12)\\ &= \mathsf{Forget}((S, \perp) \odot l) && (13)\\ &= \mathsf{Forget}(S, \perp \sqcup l) && (14)\\ &= \mathsf{Forget}(S, l) && (15)\\ &= S(- \sqcup l) && (16) \end{aligned}$$

This comes with the following data:

$$\begin{aligned} &\eta = \mathsf{id}_S : S \longrightarrow T_\perp S \\ &\mu = \mathsf{id}_{S(-\sqcup l_1 \sqcup l_2)} : T_{l_1} T_{l_2} S \longrightarrow T_{l_1 \sqcup l_2} S \end{aligned}$$

Clearly, the standard diagrams will hold. In fact, this graded monad is indeed just a renaming of the action $T_l = - \odot l : \mathsf{CSet}(L) \longrightarrow \mathsf{CSet}(L)$, and the adjunction is an equivalence of categories because the unit and counit are identities. Thus, the latter has all the necessary structure to model the system presented here, and makes the interpretation of the system quite simple.

The interpretation terms and types is as follows:

Lemma 23 (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* $\mathsf{Gr}(L, \mathsf{CSet}(L))$.

Proof. This is a proof by induction on the form of the assumed typing derivation.

First, consider the variable rule:

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

We know that $x :^{\ell_0} A \in \Omega$, and hence, $[\![\Omega]\!] = (\Omega_1, x :^{\ell_0} A, \Omega_2)$ for some Ω_1 and Ω_2. Hence, it suffices to find a multimorphism:

$$[\![x]\!] : \langle [\![\Omega_1]\!], [\![A]\!]_{\ell_0}, [\![\Omega_2]\!] \rangle \longrightarrow [\![A]\!]_{\ell}$$

We know by definition that $[\![A]\!]_{\ell_0} = ([\![A]\!], \ell_0)$, and $[\![A]\!]_{\ell} = ([\![A]\!], \ell)$. By assumption we know $\ell_0 \leq \ell$ which implies that there is a unique multimorphism $f : \ell_0 \leq \ell$. Thus, we have the following multimorphism:

$$\langle [\![\Omega_1]\!], ([\![A]\!], \ell_0), [\![\Omega_2]\!] \rangle \xrightarrow{\pi} ([\![A]\!], \ell_0) \xrightarrow{[\![A]\!](f)} ([\![A]\!], \ell)$$

Furthermore, this is natural in $[\![\Omega_1]\!]$, $[\![\Omega_2]\!]$, A, ℓ_0, and ℓ.

Next consider the abstraction rule:

$$\frac{\Omega, x :^{\ell} A \vdash b :^{\ell} B}{\Omega \vdash \lambda x : A.b :^{\ell} A \to B}\ \text{SDC-Abs}$$

By the induction hypothesis we have a multimorphism:

$$\langle [\![\Omega]\!], ([\![A]\!], \ell) \rangle \xrightarrow{[\![b]\!]} ([\![B]\!], \ell)$$

Finally, define $[\![\lambda x : A.b]\!] = \mathsf{curry}([\![b]\!])$.

Now consider the application rule:

$$\frac{\Omega \vdash b :^{\ell} A \to B \qquad \Omega \vdash a :^{\ell} A}{\Omega \vdash b\ a :^{\ell} B}\ \text{SDC-App}$$

By the induction hypothesis we have the following multimorphisms:

$$\langle [\![\Omega]\!] \rangle \xrightarrow{[\![b]\!]} ([\![A]\!], \ell) \Rightarrow ([\![B]\!], \ell) \tag{17}$$

$$\langle [\![\Omega]\!] \rangle \xrightarrow{[\![a]\!]} ([\![A]\!], \ell) \tag{18}$$

and then our final interpretation is as follows:

$$\langle [\![\Omega]\!] \rangle \xrightarrow{\mathsf{prod}([\![b]\!],[\![a]\!])} (([\![A]\!],\ell) \Rightarrow ([\![B]\!],\ell)) \times ([\![A]\!],\ell) \xrightarrow{\mathsf{eval}} ([\![B]\!],\ell)$$

Consider the pair rule:

$$\textsc{SDC-Pair}\quad \frac{\begin{array}{c}\Omega \vdash\ a_1\ :^{\ell}\ A_1\\ \Omega \vdash\ a_2\ :^{\ell}\ A_2\end{array}}{\Omega \vdash\ (a_1, a_2)\ :^{\ell}\ A_1 \times A_2}$$

By the induction hypothesis we have:

$$\langle [\![\Omega]\!] \rangle \xrightarrow{[\![a_1]\!]} ([\![A_1]\!],\ell) \tag{19}$$

$$\langle [\![\Omega]\!] \rangle \xrightarrow{[\![a_2]\!]} ([\![A_2]\!],\ell) \tag{20}$$

$$\tag{21}$$

Then out final interpretation is $[\![(a_1, a_2)]\!] = \mathsf{prod}(a_1, a_2)$.

Consider the first projection rule:

$$\textsc{SDC-Proj1}\quad \frac{\Omega \vdash\ a\ :^{\ell}\ A_1 \times A_2}{\Omega \vdash\ \pi_1\ a\ :^{\ell}\ A_1}$$

By the induction hypothesis we have:

$$\langle [\![\Omega]\!] \rangle \xrightarrow{[\![a]\!]} ([\![A_1]\!],\ell) \times ([\![A_2]\!],\ell) \tag{22}$$

$$\tag{23}$$

Then our final interpretation is $[\![\pi_1\ a]\!] = \pi_1([\![a]\!])$. The case for the second projection is similar.

Consider the first injection rule:

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

By the induction hypothesis we have:

$$\langle [\![\Omega]\!] \rangle \xrightarrow{[\![a_1]\!]} ([\![A_1]\!],\ell) \tag{24}$$

Then our final interpretation is $[\![\mathbf{inj}_1\, a_1]\!] = \iota_1([\![a_1]\!])$. The second injection is similar.

Consider the case rule:

$$\frac{\begin{array}{c}\text{SDC-CASE}\\ \Omega \vdash\ a\ :^{\ell}\ A_1 + A_2 \\ \Omega, x :^{\ell} A_1 \vdash\ b_1\ :^{\ell}\ B \\ \Omega, y :^{\ell} A_2 \vdash\ b_2\ :^{\ell}\ B\end{array}}{\Omega \vdash\ \mathbf{case}\ a\ \mathbf{of}\ x \hookrightarrow b_1 \mid y \hookrightarrow b_2\ :^{\ell}\ B}$$

By the induction hypothesis we have:

$$\langle [\![\Omega]\!] \rangle \xrightarrow{[\![a]\!]} ([\![A_1]\!], \ell) + ([\![A_2]\!], \ell) \tag{25}$$

$$\langle [\![\Omega]\!], ([\![A_1]\!], \ell) \rangle \xrightarrow{[\![b_1]\!]} ([\![B]\!], \ell) \tag{26}$$

$$\langle [\![\Omega]\!], ([\![A_2]\!], \ell) \rangle \xrightarrow{[\![b_2]\!]} ([\![B]\!], \ell) \tag{27}$$

$$\tag{28}$$

Then our final interpretation is:

$$[\![\mathbf{case}\ a\ \mathbf{of}\ x \hookrightarrow b_1 \mid y \hookrightarrow b_2]\!] = \mathsf{contract}_{[\![\Omega]\!]}(\mathsf{coprod}([\![b_1]\!], [\![b_2]\!])(\mathsf{id}_{[\![\Omega]\!]}, [\![a]\!]))$$

Consider the case rule:

$$\frac{\begin{array}{c}\text{SDC-RETURN}\\ \Omega \vdash\ a\ :^{\ell \vee \ell_0}\ A\end{array}}{\Omega \vdash\ \eta^{\ell_0}\ a\ :^{\ell}\ T^{\ell_0}\ A}$$

By the induction hypothesis we have:

$$\langle [\![\Omega]\!] \rangle \xrightarrow{[\![a]\!]} ([\![A]\!], \ell \vee \ell_0) \tag{29}$$

Furthermore, define $[\![T^{\ell}\ A]\!] = [\![A]\!](- \vee \ell)$. Then we have the following final interpretation $[\![\eta^{\ell_0}\ a]\!] = [\![a]\!]$. This is because for a morphism $([\![A]\!], \ell \vee \ell_0) \longrightarrow ([\![A]\!](\ell_0 \vee -), \ell)$ to exist there must be a morphism $[\![A]\!](\ell \vee \ell_0) \xrightarrow{f} [\![A]\!](\ell \vee \ell_0)$, but this implies that we can define $f = \mathsf{id}$. Thus, we can freely internalize labels in the model using least-upper bound.

Consider the case rule:

$$\frac{\begin{array}{c}\text{SDC-BIND}\\ \Omega \vdash\ a\ :^{\ell}\ T^{\ell_0}\ A \\ \Omega, x :^{\ell \vee \ell_0} A \vdash\ b\ :^{\ell}\ B\end{array}}{\Omega \vdash\ \mathbf{bind}^{\ell_0}\ x = a\ \mathbf{in}\ b\ :^{\ell}\ B}$$

By the induction hypothesis we have:

$$\langle [\![\Omega]\!] \rangle \xrightarrow{[\![a]\!]} ([\![A]\!](- \vee \ell_0), \ell) \tag{30}$$

$$\langle [\![\Omega]\!], ([\![A]\!], \ell \vee \ell_0) \rangle \xrightarrow{[\![b]\!]} ([\![B]\!], \ell) \tag{31}$$

$$\tag{32}$$

Then we have the following final interpretation $[\![\mathbf{bind}^{\ell_0}\ x = a\ \mathbf{in}\ b]\!] = [\![b]\!](\mathsf{id}_\Omega, [\![a]\!])$. That is, bind is simply composition in the model.

B System Specification for SDC

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

B.1 Typing and Operational Semantics

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

$$\textsc{SDC-Var}\ \frac{\ell_0 \leq \ell \qquad x :^{\ell_0} A \in \Omega}{\Omega \vdash x :^\ell A} \qquad \textsc{SDC-Unit}\ \frac{}{\Omega \vdash \mathbf{unit} :^\ell \mathbf{Unit}} \qquad \textsc{SDC-Abs}\ \frac{\Omega, x :^\ell A \vdash b :^\ell B}{\Omega \vdash \lambda x\!:\!A.b :^\ell A \to B}$$

$$\textsc{SDC-App}\ \frac{\Omega \vdash b :^\ell A \to B \qquad \Omega \vdash a :^\ell A}{\Omega \vdash b\ a :^\ell B} \qquad \textsc{SDC-Pair}\ \frac{\Omega \vdash a_1 :^\ell A_1 \qquad \Omega \vdash a_2 :^\ell A_2}{\Omega \vdash (a_1, a_2) :^\ell A_1 \times A_2} \qquad \textsc{SDC-Proj1}\ \frac{\Omega \vdash a :^\ell A_1 \times A_2}{\Omega \vdash \pi_1\ a :^\ell A_1}$$

$$\textsc{SDC-Proj2}\ \frac{\Omega \vdash a :^\ell A_1 \times A_2}{\Omega \vdash \pi_2\ a :^\ell A_2} \qquad \textsc{SDC-Inj1}\ \frac{\Omega \vdash a_1 :^\ell A_1}{\Omega \vdash \mathbf{inj}_1\ a_1 :^\ell A_1 + A_2} \qquad \textsc{SDC-Inj2}\ \frac{\Omega \vdash a_2 :^\ell A_2}{\Omega \vdash \mathbf{inj}_2\ a_2 :^\ell A_1 + A_2}$$

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

$$\textsc{SDC-Return}\quad \frac{\Omega \vdash a :^{\ell \vee \ell_0} A}{\Omega \vdash \eta^{\ell_0}\ a :^{\ell} T^{\ell_0}\ A}$$

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

$\boxed{a \leadsto a'}$ *(CBN small step operational semantics for SDC)*

$$\textsc{SDCStep-AppCong}\quad \frac{a \leadsto a'}{a\ b \leadsto a'\ b}$$

$$\textsc{SDCStep-Beta}\quad \frac{}{(\lambda x : A.a)\ b \leadsto a\{b/x\}}$$

$$\textsc{SDCStep-Proj1Cong}\quad \frac{a \leadsto a'}{\pi_1\ a \leadsto \pi_1\ a'}$$

$$\textsc{SDCStep-Proj2Cong}\quad \frac{a \leadsto a'}{\pi_2\ a \leadsto \pi_2\ a'}$$

$$\textsc{SDCStep-Proj1Beta}\quad \frac{}{\pi_1\ (a_1, a_2) \leadsto a_1}$$

$$\textsc{SDCStep-Proj2Beta}\quad \frac{}{\pi_2\ (a_1, a_2) \leadsto a_2}$$

$$\textsc{SDCStep-CaseCong}\quad \frac{a \leadsto a'}{\mathbf{case}\, a\, \mathbf{of}\, x \hookrightarrow b_1 \mid y \hookrightarrow b_2 \leadsto \mathbf{case}\, a'\, \mathbf{of}\, b_1 ; b_2}$$

$$\textsc{SDCStep-Case1Beta}\quad \frac{}{\mathbf{case}\, (\mathbf{inj}_1\ a)\, \mathbf{of}\, x \hookrightarrow b_1 \mid y \hookrightarrow b_2 \leadsto b_1\{a/x\}}$$

$$\textsc{SDCStep-Case2Beta}\quad \frac{}{\mathbf{case}\, (\mathbf{inj}_2\ a)\, \mathbf{of}\, x \hookrightarrow b_1 \mid y \hookrightarrow b_2 \leadsto b_2\{a/y\}}$$

$$\textsc{SDCStep-BindCong}\quad \frac{a \leadsto a'}{\mathbf{bind}^{\ell}\, x = a\, \mathbf{in}\, b \leadsto \mathbf{bind}^{\ell}\, x = a'\, \mathbf{in}\, b}$$

$$\textsc{SDCStep-BindBeta}\quad \frac{}{\mathbf{bind}^{\ell}\, x = \eta^{\ell}\ a\, \mathbf{in}\, b \leadsto b\{a/x\}}$$

B.2 Indexed Indistinguishability

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

$$\textsc{SGEq-Var}\quad \frac{x : \ell_0\ \mathbf{in}\ \Phi \qquad \ell_0 \leq \ell}{\Phi \vdash x \sim_\ell x}$$

$$\textsc{SGEq-Unit}\quad \frac{}{\Phi \vdash \mathbf{unit} \sim_\ell \mathbf{unit}}$$

$$\textsc{SGEq-Abs}\quad \frac{\Phi, x : \ell \vdash b_1 \sim_\ell b_2}{\Phi \vdash \lambda x : A.b_1 \sim_\ell \lambda x : A.b_2}$$

$$\textsc{SGEq-App}\quad \frac{\Phi \vdash b_1 \sim_\ell b_2 \qquad \Phi \vdash a_1 \sim_\ell a_2}{\Phi \vdash b_1\ a_1 \sim_\ell b_2\ a_2}$$

$$\textsc{SGEq-Return}\quad \frac{\Phi \vdash^{\ell_0}_{\ell} a_1 \sim a_2}{\Phi \vdash \eta^{\ell_0}\ a_1 \sim_\ell \eta^{\ell_0}\ a_2}$$

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

$$\textsc{SGEq-Pair}\quad \frac{\Phi \vdash a_1 \sim_\ell a_2 \qquad \Phi \vdash b_1 \sim_\ell b_2}{\Phi \vdash (a_1, b_1) \sim_\ell (a_2, b_2)}$$

$$\textsc{SGEq-Proj1}\quad \frac{\Phi \vdash a_1 \sim_\ell a_2}{\Phi \vdash \pi_1\ a_1 \sim_\ell \pi_1\ a_2}$$

$$\textsc{SGEq-Proj2}\quad \frac{\Phi \vdash a_1 \sim_\ell a_2}{\Phi \vdash \pi_2\ a_1 \sim_\ell \pi_2\ a_2}$$

$$\textsc{SGEq-Inj1}\quad \frac{\Phi \vdash a_1 \sim_\ell a_1'}{\Phi \vdash \mathbf{inj}_1\ a_1 \sim_\ell \mathbf{inj}_1\ a_1'}$$

$$\textsc{SGEq-Inj2}\quad \frac{\Phi \vdash a_2 \sim_\ell a_2'}{\Phi \vdash \mathbf{inj}_2\ a_2 \sim_\ell \mathbf{inj}_2\ a_2'}$$

$$\textsc{SGEq-Case}\quad \frac{\Phi \vdash a \sim_\ell a' \qquad \Phi, x : \ell \vdash b_1 \sim_\ell b_1' \qquad \Phi, y : \ell \vdash b_2 \sim_\ell b_2'}{\Phi \vdash \mathbf{case}\ a\ \mathbf{of}\ x \hookrightarrow b_1 \mid y \hookrightarrow b_2 \sim_\ell \mathbf{case}\ a'\ \mathbf{of}\ x \hookrightarrow b_1' \mid y \hookrightarrow b_2'}$$

$\boxed{\Phi \vdash^{\ell_0}_{\ell} a \sim b}$ *(Conditional Syntactic Equality)*

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

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

C System Specification for DDC

This is the complete system that we have formalized using the Coq proof assistant. The type setting of all of these rules is generated from the same Ott source that also generates the Coq definitions. Some of the following rules may appear different from their presentation in the paper. The reason behind this is that Ott is not very good at handling multiple variable bindings. So, when necessary, we replace expressions involving multiple bound variables with equivalent expressions that have only single bound variable.

C.1 Operational semantics

$\square$ *(Values that are types)*

$$\frac{}{\mathsf{ValueType}\ s}\ \textsc{ValueType-Type} \qquad \frac{}{\mathsf{ValueType}\ \Pi x :^{\ell} A.B}\ \textsc{ValueType-Pi} \qquad \frac{}{\mathsf{ValueType}\ \Sigma x{:}^{\ell} A.B}\ \textsc{ValueType-WSigma}$$

$$\frac{}{\mathsf{ValueType}\ A + B}\ \textsc{ValueType-Sum} \qquad \frac{}{\mathsf{ValueType}\ \mathbf{Unit}}\ \textsc{ValueType-Unit}$$

$\square$ *(Values)*

$$\frac{\mathsf{ValueType}\ a}{\mathsf{ValueType}\ a}\ \textsc{V-ValueType} \qquad \frac{}{\mathsf{ValueType}\ \mathbf{unit}}\ \textsc{V-TmUnit} \qquad \frac{}{\mathsf{ValueType}\ (a^{\ell}, b)}\ \textsc{V-WPair}$$

$$\frac{}{\mathsf{ValueType}\ \mathbf{inj}_1\ a}\ \textsc{V-Inj1} \qquad \frac{}{\mathsf{ValueType}\ \mathbf{inj}_2\ a}\ \textsc{V-Inj2}$$

$\boxed{a \leadsto a'}$ *(CBN small-step operational semantics)*

$$\frac{a \leadsto a'}{a\ b^{\ell} \leadsto a'\ b^{\ell}}\ \textsc{S-AppCong} \qquad \frac{}{(\lambda x :^{\ell} A.a)\ b^{\ell} \leadsto a\{b/x\}}\ \textsc{S-Beta}$$

$$\frac{a \leadsto a'}{\mathbf{case}_{\ell}\ a\ \mathbf{of}\ b_1; b_2 \leadsto \mathbf{case}_{\ell}\ a'\ \mathbf{of}\ b_1; b_2}\ \textsc{S-CaseCong} \qquad \frac{}{\mathbf{case}_{\ell}\ (\mathbf{inj}_1\ a)\ \mathbf{of}\ b_1; b_2 \leadsto b_1\ a^{\ell}}\ \textsc{S-Case1Beta}$$

$$\frac{}{\mathbf{case}_{\ell}\ (\mathbf{inj}_2\ a)\ \mathbf{of}\ b_1; b_2 \leadsto b_2\ a^{\ell}}\ \textsc{S-Case2Beta} \qquad \frac{a \leadsto a'}{\mathbf{let}\ (x^{\ell},) = a\ \mathbf{in}\ b \leadsto \mathbf{let}\ (x^{\ell},) = a'\ \mathbf{in}\ b}\ \textsc{S-LetPairCong}$$

$$\frac{}{\mathbf{let}\ (x^{\ell},) = ({a_1}^{\ell}, a_2)\ \mathbf{in}\ b \leadsto b\{a_1/x\}\ {a_2}^{\perp}}\ \textsc{S-LetPairBeta}$$

C.2 Definitional equality

$\boxed{\Phi \vdash a \equiv_\ell b}$ *(Definitional Equality)*

$$\textsc{Eq-Refl}\quad \frac{\Phi \vdash a : \ell}{\Phi \vdash a \equiv_\ell a}$$

$$\textsc{Eq-Sym}\quad \frac{\Phi \vdash a \equiv_\ell b}{\Phi \vdash b \equiv_\ell a}$$

$$\textsc{Eq-Trans}\quad \frac{\Phi \vdash a \equiv_\ell b \qquad \Phi \vdash b \equiv_\ell c}{\Phi \vdash a \equiv_\ell c}$$

$$\textsc{Eq-SubstIrrel}\quad \frac{\Phi, x : \ell \vdash b_1 \equiv_k b_2 \qquad \neg(\ell \leq k)}{\Phi \vdash b_1\{a_1/x\} \equiv_k b_2\{a_2/x\}}$$

$$\textsc{Eq-Beta}\quad \frac{\Phi \vdash a : \ell \qquad a \leadsto b \qquad \Phi \vdash b : \ell}{\Phi \vdash a \equiv_\ell b}$$

$$\textsc{Eq-Pi}\quad \frac{\Phi \vdash A_1 \equiv_\ell A_2 \qquad \Phi, x : \ell \vdash B_1 \equiv_\ell B_2}{\Phi \vdash \Pi x :^{\ell_0} A_1 . B_1 \equiv_\ell \Pi x :^{\ell_0} A_2 . B_2}$$

$$\textsc{Eq-Abs}\quad \frac{\Phi \vdash^{\top} A_1 \equiv_\ell A_2 \qquad \Phi, x : \ell_0 \vdash b_1 \equiv_\ell b_2}{\Phi \vdash \lambda x :^{\ell_0} A_1 . b_1 \equiv_\ell \lambda x :^{\ell_0} A_2 . b_2}$$

$$\textsc{Eq-App}\quad \frac{\Phi \vdash b_1 \equiv_\ell b_2 \qquad \Phi \vdash^{\ell_0} a_1 \equiv_\ell a_2}{\Phi \vdash b_1\ {a_1}^{\ell_0} \equiv_\ell b_2\ {a_2}^{\ell_0}}$$

$$\textsc{Eq-PiFst}\quad \frac{\Phi \vdash \Pi x :^{\ell_0} A_1 . B_1 \equiv_\ell \Pi x :^{\ell_0} A_2 . B_2}{\Phi \vdash A_1 \equiv_\ell A_2}$$

$$\textsc{Eq-PiSnd}\quad \frac{\Phi \vdash \Pi x :^{\ell_0} A_1 . B_1 \equiv_\ell \Pi x :^{\ell_0} A_2 . B_2 \qquad \Phi \vdash a_1 \equiv_\ell a_2}{\Phi \vdash B_1\{a_1/x\} \equiv_\ell B_2\{a_2/x\}}$$

$$\textsc{Eq-WSigma}\quad \frac{\Phi \vdash A_1 \equiv_\ell A_2 \qquad \Phi, x : \ell \vdash B_1 \equiv_\ell B_2}{\Phi \vdash \Sigma x{:}^{\ell_0} A_1 . B_1 \equiv_\ell \Sigma x{:}^{\ell_0} A_2 . B_2}$$

$$\textsc{Eq-WSigmaFst}\quad \frac{\Phi \vdash \Sigma x{:}^{\ell_0} A_1 . B_1 \equiv_\ell \Sigma x{:}^{\ell_0} A_2 . B_2}{\Phi \vdash A_1 \equiv_\ell A_2}$$

$$\textsc{Eq-WSigmaSnd}\quad \frac{\Phi \vdash \Sigma x{:}^{\ell_0} A_1 . B_1 \equiv_\ell \Sigma x{:}^{\ell_0} A_2 . B_2 \qquad \Phi \vdash a : \ell}{\Phi \vdash B_1\{a/x\} \equiv_\ell B_2\{a/x\}}$$

$$\textsc{Eq-WPair}\quad \frac{\Phi \vdash^{\ell_0} a_1 \equiv_\ell a_2 \qquad \Phi \vdash b_1 \equiv_\ell b_2}{\Phi \vdash ({a_1}^{\ell_0}, b_1) \equiv_\ell ({a_2}^{\ell_0}, b_2)}$$

$$\textsc{Eq-LetPair}\quad \frac{\Phi \vdash a_1 \equiv_\ell a_2 \qquad \Phi, x : \ell_0 \vdash b_1 \equiv_\ell b_2}{\Phi \vdash \mathbf{let}\,(x^{\ell_0},) = a_1\ \mathbf{in}\ b_1 \equiv_\ell \mathbf{let}\,(x^{\ell_0},) = a_2\ \mathbf{in}\ b_2}$$

$$\textsc{Eq-Sum}\quad \frac{\Phi \vdash A_1 \equiv_\ell A_1' \qquad \Phi \vdash A_2 \equiv_\ell A_2'}{\Phi \vdash A_1 + A_2 \equiv_\ell A_1' + A_2'}$$

$$\textsc{Eq-SumFst}\quad \frac{\Phi \vdash A_1 + A_2 \equiv_\ell A_1' + A_2'}{\Phi \vdash A_1 \equiv_\ell A_1'}$$

$$\textsc{Eq-SumSnd}\quad \frac{\Phi \vdash A_1 + A_2 \equiv_\ell A_1' + A_2'}{\Phi \vdash A_2 \equiv_\ell A_2'}$$

$$\textsc{Eq-Inj1}\quad \frac{\Phi \vdash a_1 \equiv_\ell a_1'}{\Phi \vdash \mathbf{inj}_1\ a_1 \equiv_\ell \mathbf{inj}_1\ a_1'} \qquad \textsc{Eq-Inj2}\quad \frac{\Phi \vdash a_2 \equiv_\ell a_2'}{\Phi \vdash \mathbf{inj}_2\ a_2 \equiv_\ell \mathbf{inj}_2\ a_2'}$$

$$\textsc{Eq-Case}\quad \frac{\begin{array}{c}\Phi \vdash a \equiv_\ell a' \\ \Phi \vdash b_1 \equiv_\ell b_1' \\ \Phi \vdash b_2 \equiv_\ell b_2' \\ \ell_0 \leq \ell\end{array}}{\Phi \vdash \mathbf{case}_{\ell_0}\ a\ \mathbf{of}\ b_1; b_2 \equiv_\ell \mathbf{case}_{\ell_0}\ a'\ \mathbf{of}\ b_1'; b_2'} \qquad \textsc{Eq-TyUnit}\quad \frac{}{\Phi \vdash \mathbf{Unit} \equiv_\ell \mathbf{Unit}}$$

$$\textsc{Eq-TmUnit}\quad \frac{}{\Phi \vdash \mathbf{unit} \equiv_\ell \mathbf{unit}}$$

$\boxed{\Phi \vdash^k a \equiv_\ell b}$ *(Conditional Definitional Equality)*

$$\textsc{CEq-Leq}\quad \frac{\begin{array}{c}\ell_0 \leq \ell \\ \Phi \vdash a_1 \sim_\ell a_2\end{array}}{\Phi \vdash_\ell^{\ell_0} a_1 \sim a_2} \qquad \textsc{CEq-Nleq}\quad \frac{\neg(\ell_0 \leq \ell)}{\Phi \vdash_\ell^{\ell_0} a_1 \sim a_2}$$

$\boxed{\Phi \vdash a : k}$ *(Grading)*

$$\textsc{G-Type}\quad \frac{}{\Phi \vdash s : \ell} \qquad \textsc{G-Var}\quad \frac{\ell_0 \leq \ell \qquad x : \ell_0\ \mathbf{in}\ \Phi}{\Phi \vdash x : \ell} \qquad \textsc{G-Pi}\quad \frac{\begin{array}{c}\Phi \vdash A : \ell \\ \Phi, x : \ell \vdash B : \ell\end{array}}{\Phi \vdash \Pi x :^{\ell_0} A.B : \ell} \qquad \textsc{G-Abs}\quad \frac{\begin{array}{c}\Phi, x : \ell_0 \vdash b : \ell \\ \Phi \vdash_\ell^\top A\end{array}}{\Phi \vdash \lambda x :^{\ell_0} A.b : \ell}$$

$$\textsc{G-App}\quad \frac{\Phi \vdash b : \ell \qquad \Phi \vdash_\ell^{\ell_0} a}{\Phi \vdash b\ a^{\ell_0} : \ell} \qquad \textsc{G-WSigma}\quad \frac{\begin{array}{c}\Phi \vdash A : \ell \\ \Phi, x : \ell \vdash B : \ell\end{array}}{\Phi \vdash \Sigma x{:}^{\ell_0} A.B : \ell} \qquad \textsc{G-WPair}\quad \frac{\Phi \vdash_\ell^{\ell_0} a \qquad \Phi \vdash b : \ell}{\Phi \vdash (a^{\ell_0}, b) : \ell}$$

$$\textsc{G-LetPair}\quad \frac{\begin{array}{c}\Phi \vdash a : \ell \\ (\Phi, x : \ell_0) \vdash c : \ell\end{array}}{\Phi \vdash \mathbf{let}\ (x^{\ell_0},) = a\ \mathbf{in}\ c : \ell} \qquad \textsc{G-Sum}\quad \frac{\Phi \vdash A : \ell \qquad \Phi \vdash B : \ell}{\Phi \vdash A + B : \ell} \qquad \textsc{G-Inj1}\quad \frac{\Phi \vdash a_1 : \ell}{\Phi \vdash \mathbf{inj}_1\ a_1 : \ell}$$

$$\textsc{G-Inj2}\quad \frac{\Phi \vdash a_2 : \ell}{\Phi \vdash \mathbf{inj}_2\ a_2 : \ell} \qquad \textsc{G-Case}\quad \frac{\begin{array}{c}\Phi \vdash a : \ell \qquad \Phi \vdash b_1 : \ell \\ \Phi \vdash b_2 : \ell \qquad \ell_0 \leq \ell\end{array}}{\Phi \vdash \mathbf{case}_{\ell_0}\ a\ \mathbf{of}\ b_1; b_2 : \ell} \qquad \textsc{G-TyUnit}\quad \frac{}{\Phi \vdash \mathbf{Unit} : \ell} \qquad \textsc{G-TmUnit}\quad \frac{}{\Phi \vdash \mathbf{unit} : \ell}$$

$\boxed{\Phi \vdash^{\ell}_{k} a}$ *(Conditional Grading)*

$$\textsc{CG-Leq}\quad \frac{k_0 \leq k \qquad \Phi \vdash a : k}{\Phi \vdash^{k_0}_{k} a} \qquad\qquad \textsc{CG-Nleq}\quad \frac{\neg(k_0 \leq k)}{\Phi \vdash^{k_0}_{k} a}$$

C.3 Type System

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

$$\textsc{T-Type}\quad \frac{\ell \leq C \qquad \mathcal{A}(s_1, s_2)}{\Omega \vdash s_1 :^{\ell} s_2}$$

$$\textsc{T-Conv}\quad \frac{\begin{array}{c}\Omega \vdash a :^{\ell} A \\ |C \wedge \Omega| \vdash A \equiv_C B \\ C \wedge \Omega \vdash B :^{C} s\end{array}}{\Omega \vdash a :^{\ell} B}$$

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

$$\textsc{T-Pi}\quad \frac{\begin{array}{c}\Omega \vdash A :^{\ell} s_1 \\ \Omega, x :^{\ell} A \vdash B :^{\ell} s_2 \\ \mathcal{R}(s_1, s_2, s_3)\end{array}}{\Omega \vdash \Pi x :^{\ell_0} A.B :^{\ell} s_3}$$

$$\textsc{T-Abs}\quad \frac{\begin{array}{c}\Omega, x :^{\ell_0 \vee \ell} A \vdash b :^{\ell} B \\ C \wedge \Omega \vdash (\Pi x :^{\ell_0} A.B) :^{C} s\end{array}}{\Omega \vdash \lambda x :^{\ell_0} A.b :^{\ell} \Pi x :^{\ell_0} A.B}$$

$$\textsc{T-App}\quad \frac{\begin{array}{c}\Omega \vdash b :^{\ell} \Pi x :^{\ell_0} A.B \\ \Omega \vdash a :^{\ell_0 \vee \ell} A \\ \ell_0 \leq C\end{array}}{\Omega \vdash b\ a^{\ell_0} :^{\ell} B\{a/x\}}$$

$$\textsc{T-AppIrrel}\quad \frac{\begin{array}{c}\Omega \vdash b :^{\ell} \Pi x :^{\ell_0} A.B \\ C \wedge \Omega \vdash a :^{C} A \\ C < \ell_0\end{array}}{\Omega \vdash b\ a^{\ell_0} :^{\ell} B\{a/x\}}$$

$$\textsc{T-WSigma}\quad \frac{\begin{array}{c}\Omega \vdash A :^{\ell} s_1 \\ \Omega, x :^{\ell} A \vdash B :^{\ell} s_2 \\ \mathcal{R}(s_1, s_2, s_3)\end{array}}{\Omega \vdash \Sigma x{:}^{\ell_0} A.B :^{\ell} s_3}$$

$$\textsc{T-WPair}\quad \frac{\begin{array}{c}C \wedge \Omega \vdash (\Sigma x{:}^{\ell_0} A.B) :^{C} s \\ \Omega \vdash a :^{\ell_0 \vee \ell} A \\ \Omega \vdash b :^{\ell} B\{a/x\} \\ \ell_0 \leq C\end{array}}{\Omega \vdash (a^{\ell_0}, b) :^{\ell} \Sigma x{:}^{\ell_0} A.B}$$

$$\textsc{T-WPairIrrel}\quad \frac{\begin{array}{c}C \wedge \Omega \vdash (\Sigma x{:}^{\ell_0} A.B) :^{C} s \\ C \wedge \Omega \vdash a :^{C} A \\ C < \ell_0 \\ \Omega \vdash b :^{\ell} B\{a/x\}\end{array}}{\Omega \vdash (a^{\ell_0}, b) :^{\ell} \Sigma x{:}^{\ell_0} A.B}$$

$$\textsc{T-LetPairCA}\quad \frac{\begin{array}{c}\Omega \vdash a :^{\ell} \Sigma x{:}^{\ell_0} A.B \\ \Omega, x :^{\ell_0 \vee \ell} A \vdash c :^{\ell} \Pi y :^{k} B.C\{(x^{\ell_0}, y)/z\} \\ C \wedge (\Omega, z :^{\top} (\Sigma x{:}^{\ell_0} A.B)) \vdash C :^{C} s\end{array}}{\Omega \vdash \mathbf{let}\,(x^{\ell_0},) = a\,\mathbf{in}\,c :^{\ell} C\{a/z\}}$$

T-Sum
$$\frac{\Omega \vdash A :^{\ell} s \qquad \Omega \vdash B :^{\ell} s}{\Omega \vdash A + B :^{\ell} s}$$

T-Inj1
$$\frac{\Omega \vdash a_1 :^{\ell} A_1 \qquad C \wedge \Omega \vdash A_1 + A_2 :^{C} s}{\Omega \vdash \mathbf{inj}_1\, a_1 :^{\ell} A_1 + A_2}$$

T-Inj2
$$\frac{\Omega \vdash a_2 :^{\ell} A_2 \qquad C \wedge \Omega \vdash A_1 + A_2 :^{C} s}{\Omega \vdash \mathbf{inj}_2\, a_2 :^{\ell} A_1 + A_2}$$

T-CaseC
$$\frac{\begin{array}{c}\Omega \vdash a :^{\ell} A_1 + A_2 \\ \Omega \vdash b_1 :^{\ell} \Pi x :^{\ell_0} A_1 . B\{\mathbf{inj}_1\, x/z\} \\ \Omega \vdash b_2 :^{\ell} \Pi y :^{\ell_0} A_2 . B\{\mathbf{inj}_2\, y/z\} \\ \ell_0 \leq \ell \\ ((C \wedge \Omega), z :^{C} A_1 + A_2) \vdash B :^{C} s\end{array}}{\Omega \vdash \mathbf{case}_{\ell_0}\, a\, \mathbf{of}\, b_1 ; b_2 :^{\ell} B\{a/z\}}$$

T-TmUnit
$$\frac{\ell \leq C}{\Omega \vdash \mathbf{Unit} :^{\ell} s}$$

T-TyUnit
$$\frac{\ell \leq C}{\Omega \vdash \mathbf{unit} :^{\ell} \mathbf{Unit}}$$

C.4 Indexed Indistinguishability

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

GEq-Type
$$\frac{}{\Phi \vdash s \sim_{\ell} s}$$

GEq-Var
$$\frac{x : \ell_0\, \mathbf{in}\, \Phi \qquad \ell_0 \leq \ell}{\Phi \vdash x \sim_{\ell} x}$$

GEq-Pi
$$\frac{\Phi \vdash A_1 \sim_{\ell} A_2 \qquad \Phi, x : \ell \vdash B_1 \sim_{\ell} B_2}{\Phi \vdash \Pi x :^{\ell_0} A_1 . B_1 \sim_{\ell} \Pi x :^{\ell_0} A_2 . B_2}$$

GEq-Abs
$$\frac{\Phi, x : \ell_0 \vdash b_1 \sim_{\ell} b_2 \qquad \Phi \vdash^{\top}_{\ell} A_1 \sim A_2}{\Phi \vdash \lambda x :^{\ell_0} A_1 . b_1 \sim_{\ell} \lambda x :^{\ell_0} A_2 . b_2}$$

GEq-App
$$\frac{\Phi \vdash b_1 \sim_{\ell} b_2 \qquad \Phi \vdash^{\ell_0}_{\ell} a_1 \sim a_2}{\Phi \vdash b_1\ {a_1}^{\ell_0} \sim_{\ell} b_2\ {a_2}^{\ell_0}}$$

GEq-WSigma
$$\frac{\Phi \vdash A_1 \sim_{\ell} A_2 \qquad \Phi, x : \ell \vdash B_1 \sim_{\ell} B_2}{\Phi \vdash \Sigma x{:}^{\ell_0} A_1 . B_1 \sim_{\ell} \Sigma x{:}^{\ell_0} A_2 . B_2}$$

GEq-WPair
$$\frac{\Phi \vdash^{\ell_0}_{\ell} a_1 \sim a_2 \qquad \Phi \vdash b_1 \sim_{\ell} b_2}{\Phi \vdash ({a_1}^{\ell_0}, b_1) \sim_{\ell} ({a_2}^{\ell_0}, b_2)}$$

GEq-LetPair
$$\frac{\Phi \vdash a_1 \sim_{\ell} a_2 \qquad \Phi, x : \ell_0 \vdash b_1 \sim_{\ell} b_2}{\Phi \vdash \mathbf{let}\, (x^{\ell_0},) = a_1\, \mathbf{in}\, b_1 \sim_{\ell} \mathbf{let}\, (x^{\ell_0},) = a_2\, \mathbf{in}\, b_2}$$

GEq-Sum
$$\frac{\Phi \vdash A_1 \sim_{\ell} A_1' \qquad \Phi \vdash A_2 \sim_{\ell} A_2'}{\Phi \vdash A_1 + A_2 \sim_{\ell} A_1' + A_2'}$$

GEq-Inj1
$$\frac{\Phi \vdash a_1 \sim_{\ell} a_1'}{\Phi \vdash \mathbf{inj}_1\, a_1 \sim_{\ell} \mathbf{inj}_1\, a_1'}$$

GEq-Inj2
$$\frac{\Phi \vdash a_2 \sim_{\ell} a_2'}{\Phi \vdash \mathbf{inj}_2\, a_2 \sim_{\ell} \mathbf{inj}_2\, a_2'}$$

$$\textsc{GEq-Case}\quad \frac{\Phi \vdash a \sim_\ell a' \quad \Phi \vdash b_1 \sim_\ell b_1' \quad \Phi \vdash b_2 \sim_\ell b_2' \quad \ell_0 \leq \ell}{\Phi \vdash \mathbf{case}_{\ell_0}\ a\ \mathbf{of}\ b_1; b_2 \sim_\ell \mathbf{case}_{\ell_0}\ a'\ \mathbf{of}\ b_1'; b_2'}$$

$$\textsc{GEq-TyUnit}\quad \frac{}{\Phi \vdash \mathbf{Unit} \sim_\ell \mathbf{Unit}}$$

$$\textsc{GEq-TmUnit}\quad \frac{}{\Phi \vdash \mathbf{unit} \sim_\ell \mathbf{unit}}$$

$\boxed{\Phi \vdash^{\ell_0}_{\ell} a \sim b}$ *(Conditional Indistinguishability)*

$$\textsc{CEq-Leq}\quad \frac{\ell_0 \leq \ell \quad \Phi \vdash a_1 \sim_\ell a_2}{\Phi \vdash^{\ell_0}_{\ell} a_1 \sim a_2}$$

$$\textsc{CEq-Nleq}\quad \frac{\neg(\ell_0 \leq \ell)}{\Phi \vdash^{\ell_0}_{\ell} a_1 \sim a_2}$$

C.5 Auxiliary Judgments

$\boxed{\Phi \vdash a \Rightarrow_\ell b}$ *(Parallel reduction)*

$$\textsc{Par-Refl}\quad \frac{\Phi \vdash a : \ell}{\Phi \vdash a \Rightarrow_\ell a}$$

$$\textsc{Par-Pi}\quad \frac{\Phi \vdash A_1 \Rightarrow_\ell A_2 \quad \Phi, x : \ell \vdash B_1 \Rightarrow_\ell B_2}{\Phi \vdash \Pi x :^{\ell_1} A_1 . B_1 \Rightarrow_\ell \Pi x :^{\ell_1} A_2 . B_2}$$

$$\textsc{Par-AppBeta}\quad \frac{\Phi \vdash a \Rightarrow_\ell (\lambda x :^{\ell_0} A . a') \quad \Phi \vdash^{\ell_0}_{\ell} b \Rightarrow b'}{\Phi \vdash a\ b^{\ell_0} \Rightarrow_\ell a'\{b'/x\}}$$

$$\textsc{Par-App}\quad \frac{\Phi \vdash a \Rightarrow_\ell a' \quad \Phi \vdash^{\ell_0}_{\ell} b \Rightarrow b'}{\Phi \vdash a\ b^{\ell_0} \Rightarrow_\ell a'\ b'^{\ell_0}}$$

$$\textsc{Par-Abs}\quad \frac{\Phi, x : \ell_0 \vdash b_1 \Rightarrow_\ell b_2 \quad \Phi \vdash^{\top}_{\ell} A_1 \Rightarrow A_2}{\Phi \vdash \lambda x :^{\ell_0} A_1 . b_1 \Rightarrow_\ell \lambda x :^{\ell_0} A_2 . b_2}$$

$$\textsc{Par-WSigma}\quad \frac{\Phi \vdash A_1 \Rightarrow_\ell A_2 \quad \Phi, x : \ell \vdash B_1 \Rightarrow_\ell B_2}{\Phi \vdash \Sigma x :^{\ell_1} A_1 . B_1 \Rightarrow_\ell \Sigma x :^{\ell_1} A_2 . B_2}$$

$$\textsc{Par-WPair}\quad \frac{\Phi \vdash^{\ell_0}_{\ell} a_1 \Rightarrow a_2 \quad \Phi \vdash b_1 \Rightarrow_\ell b_2}{\Phi \vdash ({a_1}^{\ell_0}, b_1) \Rightarrow_\ell ({a_2}^{\ell_0}, b_2)}$$

$$\textsc{Par-WPairBeta}\quad \frac{\Phi \vdash a_1 \Rightarrow_\ell ({a_1'}^{\ell_0}, a_2') \quad \Phi, x : \ell_0 \vdash b_1 \Rightarrow_\ell b_2\{x/x\}}{\Phi \vdash \mathbf{let}\ (x^{\ell_0},) = a_1\ \mathbf{in}\ b_1 \Rightarrow_\ell b_2\{a_1'/x\}\ {a_2'}^{\bot}}$$

$$\textsc{Par-LetPair}\quad \frac{\Phi \vdash a_1 \Rightarrow_\ell a_2 \qquad \Phi, x : \ell_0 \vdash b_1 \Rightarrow_\ell b_2}{\Phi \vdash \mathbf{let}\,(x^{\ell_0},) = a_1\,\mathbf{in}\,b_1 \Rightarrow_\ell \mathbf{let}\,(x^{\ell_0},) = a_2\,\mathbf{in}\,b_2}$$

$$\textsc{Par-Sum}\quad \frac{\Phi \vdash A_1 \Rightarrow_\ell A_1' \qquad \Phi \vdash A_2 \Rightarrow_\ell A_2'}{\Phi \vdash A_1 + A_2 \Rightarrow_\ell A_1' + A_2'}$$

$$\textsc{Par-Inj1}\quad \frac{\Phi \vdash a_1 \Rightarrow_\ell a_1'}{\Phi \vdash \mathbf{inj}_1\ a_1 \Rightarrow_\ell \mathbf{inj}_1\ a_1'}$$

$$\textsc{Par-Inj2}\quad \frac{\Phi \vdash a_2 \Rightarrow_\ell a_2'}{\Phi \vdash \mathbf{inj}_2\ a_2 \Rightarrow_\ell \mathbf{inj}_2\ a_2'}$$

$$\textsc{Par-CaseBeta1}\quad \frac{\Phi \vdash a \Rightarrow_\ell \mathbf{inj}_1\ a' \qquad \Phi \vdash b_1 \Rightarrow_\ell b_1' \qquad \Phi \vdash b_2 \Rightarrow_\ell b_2' \qquad \ell_0 \leq \ell}{\Phi \vdash \mathbf{case}_{\ell_0}\ a\,\mathbf{of}\ b_1; b_2 \Rightarrow_\ell b_1'\ {a'}^{\ell_0}}$$

$$\textsc{Par-CaseBeta2}\quad \frac{\Phi \vdash a \Rightarrow_\ell \mathbf{inj}_2\ a' \qquad \Phi \vdash b_1 \Rightarrow_\ell b_1' \qquad \Phi \vdash b_2 \Rightarrow_\ell b_2' \qquad \ell_0 \leq \ell}{\Phi \vdash \mathbf{case}_{\ell_0}\ a\,\mathbf{of}\ b_1; b_2 \Rightarrow_\ell b_2'\ {a'}^{\ell_0}}$$

$$\textsc{Par-Case}\quad \frac{\Phi \vdash a \Rightarrow_\ell a' \qquad \Phi \vdash b_1 \Rightarrow_\ell b_1' \qquad \Phi \vdash b_2 \Rightarrow_\ell b_2' \qquad \ell_0 \leq \ell}{\Phi \vdash \mathbf{case}_{\ell_0}\ a\,\mathbf{of}\ b_1; b_2 \Rightarrow_\ell \mathbf{case}_{\ell_0}\ a'\,\mathbf{of}\ b_1'; b_2'}$$

$\boxed{\Phi \vdash^k_\ell a \Rightarrow b}$ *(Conditional Parallel reduction)*

$$\textsc{CPar-Leq}\quad \frac{\ell_0 \leq \ell \qquad \Phi \vdash a_1 \Rightarrow_\ell a_2}{\Phi \vdash^{\ell_0}_\ell a_1 \Rightarrow a_2}$$

$$\textsc{CPar-Nleq}\quad \frac{\neg(\ell_0 \leq \ell)}{\Phi \vdash^{\ell_0}_\ell a_1 \Rightarrow a_2}$$

$\boxed{\Phi \vdash a \Rightarrow^*_\ell b}$ *(Parallel reduction, reflexive transitive closure)*

$$\textsc{MP-Refl}\quad \frac{\Phi \vdash a : \ell}{\Phi \vdash a \Rightarrow^*_\ell a}$$

$$\textsc{MP-Step}\quad \frac{\Phi \vdash a \Rightarrow_\ell b \qquad \Phi \vdash b \Rightarrow^*_\ell a'}{\Phi \vdash a \Rightarrow^*_\ell a'}$$

$\boxed{\Phi \vdash a \Leftrightarrow_\ell b}$ *(Joinability)*

$$\textsc{join}\quad \frac{\Phi \vdash a_1 \Rightarrow^*_\ell b_1 \qquad \Phi \vdash a_2 \Rightarrow^*_\ell b_2 \qquad \Phi \vdash b_1 \sim_\ell b_2}{\Phi \vdash a_1 \Leftrightarrow_\ell a_2}$$

$\boxed{\mathsf{Consistent}\ a\ b}$ *(Consistent Head Forms)*

$$\frac{}{\mathsf{Consistent}\ s\ s}\ \textsc{Consistent-a-Type}$$

$$\frac{}{\mathsf{Consistent}\ \mathbf{Unit}\ \mathbf{Unit}}\ \textsc{Consistent-a-Unit}$$

$$\frac{}{\mathsf{Consistent}\ (\Pi x :^{\ell} A.B)\ (\Pi x :^{\ell} A'.B')}\ \textsc{Consistent-a-Pi}$$

$$\frac{}{\mathsf{Consistent}\ (\Sigma x{:}^{\ell} A.B)\ (\Sigma x{:}^{\ell} A'.B')}\ \textsc{Consistent-a-WSigma}$$

$$\frac{}{\mathsf{Consistent}\ (A + B)\ (A' + B')}\ \textsc{Consistent-a-Sum}$$

$$\frac{\neg\mathsf{ValueType}\ a}{\mathsf{Consistent}\ a\ b}\ \textsc{Consistent-a-Step-R}$$

$$\frac{\neg\mathsf{ValueType}\ b}{\mathsf{Consistent}\ a\ b}\ \textsc{Consistent-a-Step-L}$$