Dependently-Typed Programming with Logical Equality Reflection

YIYUN LIU, University of Pennsylvania, USA
STEPHANIE WEIRICH, University of Pennsylvania, USA

In dependently-typed functional programming languages that allow general recursion, programs used as proofs must be evaluated to retain type soundness. As a result, programmers must make a trade-off between performance and safety. To address this problem, we propose System DE, an explicitly-typed, moded core calculus that supports termination tracking and equality reflection. Programmers can write inductive proofs about potentially diverging programs in a logical sublanguage and reflect those proofs to the type checker, while knowing that such proofs will be erased by the compiler before execution. A key feature of System DE is its use of modes for both termination and relevance tracking, which not only simplifies the design but also leaves it open for future extension. System DE is suitable for use in the Glasgow Haskell Compiler, but could serve as the basis for any general purpose dependently-typed language.

CCS Concepts: • **Software and its engineering** → **Functional languages**; • **Theory of computation** → **Type theory**.

Additional Key Words and Phrases: Modes, Haskell, Dependent Types, Equality Reflection

ACM Reference Format:
Yiyun Liu and Stephanie Weirich. 2023. Dependently-Typed Programming with Logical Equality Reflection. *Proc. ACM Program. Lang.* 7, ICFP, Article 210 (August 2023), 37 pages. https://doi.org/10.1145/3607852

1 INTRODUCTION

Consider the following unsettling Haskell program that applies a function f of type a -> b to a term x of some arbitrary type c.

```
badApp :: forall a b c. (a -> b) -> c -> b
badApp f x = case badEq of
               Refl -> f x
  where badEq :: a :~: c
        badEq = badEq
```

With the function badApp, it is possible to write the expression (badApp not "str") to attempt to perform the bit flip operation on a string. Compared to proof assistants such as Agda [Agda Development Team 2023] and Coq [Coq Development Team 2019], Haskell [GHC Development Team 2023] does not rule out bogus equality “proofs” like badEq. In the body of badApp, by pattern matching on badEq, we are allowed to treat x as if it has type a because the equality evidence convinces the type checker that a and c are convertible. It is worth noting that despite the unsafe nature of the badApp call, programs like badApp do not violate the soundness of Haskell’s type system, as the potentially dangerous operation is protected by a pattern match on badEq that causes the program to loop indefinitely.

Authors’ addresses: Yiyun Liu, University of Pennsylvania, Philadelphia, USA, liuyiyun@seas.upenn.edu; Stephanie Weirich, University of Pennsylvania, Philadelphia, USA, sweirich@seas.upenn.edu.

2475-1421/2023/8-ART210
https://doi.org/10.1145/3607852

Unfortunately, the lack of termination tracking means that none of the explicitly constructed proofs in Haskell can be trusted. All such proofs must be evaluated at run time because the type checker does not have enough information to determine whether they will eventually produce evidence of the equality. The actual evidence produced contains no information itself; only its presence is needed to justify the safety of the computation. But computing this evidence can be expensive. In practice, this means programmers must make a choice between type safety and performance since executing proofs at run time can induce overhead that is not negligible for performance-critical code. Christiansen et al. [2019] observed this limitation in their implementation of Crucible, a framework for writing symbolic executors. To achieve reasonable performance, they created a version of their code that replaced evidence computation with `unsafeCoerce`, avoiding the runtime cost of executing proofs but losing the type soundness guarantees of Haskell's type system. This trade-off is not ideal; we want programmers to write safe code without such performance penalties.

To address this problem, we present System DE, a calculus suitable to be used as the explicitly-typed core language of the Glasgow Haskell compiler [GHC Development Team 2023]. We focus our attention at this level because the core language specifies the semantics of Haskell independent of the complexities of its many source language features and type inference algorithm. The type system of System DE includes both termination and relevance tracking, allowing it to identify evidence computations that can safely be erased during compilation.

System DE is a reformulation of System DC [Weirich et al. 2017], a dependently-typed core language, and a guideline for the extension of Haskell with dependent types. Although the solution that we explore in the context of System DE would also benefit non-dependent versions of Haskell's core language [Breitner et al. 2016; Weirich et al. 2013], and could be integrated into today's Haskell, we choose to work in a dependently-typed framework because of its uniformity, relevance to other languages that might benefit from these ideas, and future relevance to Haskell.

Overview of contributions. In this paper, we first introduce the ideas that underlie System DE through example. Section 2, imagines a future extension of Haskell that takes advantage of the new capabilities that System DE provides. Then in Section 3, we give a formal presentation of System DE and its semantics.

System DE's type system tracks termination and relevance through the use of *modes* (Section 3.1). The mode L carves out a logical sublanguage that is not only weakly-normalizing but also expressive enough to encode inductive proofs. This language includes an equality type, and terms of the equality type in the L fragment can be reflected and made available to the type checker without an explicit pattern match. As a result, logical equality proofs are erasable, and incur no computational cost for their use. Meanwhile, the P fragment does not pose any restrictions on termination or universe consistency, thus allowing backward-compatibility with existing Haskell code.

The relevance tracking mechanism of System DE is not limited to equality proofs. Rather, modes provide a general framework of relevance tracking for arbitrary terms where proof terms of equality can fit in as a special case (Section 3.2). Explicit relevance annotations are important for a dependently language such as System DE since both terms and types may appear as part of a specification that is not needed at runtime. On the other hand, there are use cases such as type providers and generic programming where the specification itself may be computed at runtime. Therefore, System DE provides users with a mechanism to precisely specify the relevance of both terms and types. Furthermore, relevance tracking in System DE affects both erasure at runtime and equational reasoning at compile-time. The compiler can erase irrelevant terms during code generation, but the type checker can also take advantage of irrelevance at compile-time by ignoring the irrelevant arguments when proving equality.

Type checking System DE is decidable and syntax directed. Like other explicitly-typed core languages for GHC [Sulzmann et al. 2007], the syntax of System DE includes explicit-but-irrelevant evidence, called *coercions*, that must be present when type casting. However, in the presence of equality reflection, System DE acts like an extensional type theory (Section 3.3) without sacrificing decidable type checking. The typed operational semantics propagates these coercions through a combination of administrative and computational rules (Section 3.4).

Section 4 presents an overview of our type soundness theorem for System DE, which we have mechanized using the Coq proof assistant [Coq Development Team 2019]. Machine-checked results are marked with their locations in our companion artifact [Liu and Weirich 2023]. The ability to reflect arbitrary equality proofs from the L fragment means that this proof is a bit more difficult than prior work—we now need to define a logical relation to prove the consistency of our equality judgment (Section 4). This difficulty stems from the specification of the semantic rules and the definition of the logical relation itself under the presence of explicit coercions and features such as dependent pattern matching. Our proof artifact is suitable for future extension and exploration: both in pursuit of extensions to the Haskell language and investigations of a syntax directed variant of extensional type theory.

System DE subsumes and reconciles features that have been explored separately in prior work. However, integrating explicit coercion proofs, relevance tracking, termination tracking, and equality reflection into the same language requires more than simple aggregation. But, by leveraging modes, we are able to reconcile these features into Dependent Haskell with minimal additional complexity. Indeed, System DE eliminates a feature found in System DC: coercion abstraction is no longer necessary in the presence of equality reflection. Furthermore, the use of modes also makes System DE extensible to features such as irrelevant strong existentials, call-by-value semantics, and extended L and P interactions (Section 5).

Finally, in Section 6, we compare System DE to its predecessors. In particular, we contrast our designs with prior work that inspired our use of modes and provide a detailed comparison to alternative approaches for implementing relevance tracking and termination tracking.

2 PROGRAMMING WITH REFLECTED EQUALITY PROOFS

The ability to reflect terminating computation as type checker evidence is a powerful language feature, as we demonstrate in this section through examples. The examples below appear in the context of a hypothetical Dependent Haskell, and require extensions to GHC's type inference algorithm for elaboration to our core language System DE.

2.1 Reasoning with Propositional Equalities

One domain that benefits from the use of dependent types is working with tables and data frames [Wright et al. 2022]. Consider the following data type definitions that can be used to represent a typed CSV file. These definitions rely on a length-indexed vector type `Vector` (not shown).

```
newtype CSV (n :: Nat) (fs :: [Type]) = Frame {
  rows :: Vector n (Row fs)
}
data Row (fs :: [Type]) where
  Nil  :: Row []
  (:>) :: forall f fs. f -> Row fs -> Row (f:fs)
```

The `CSV` type is parameterized by the number of rows `n` and a list of types for the columns. Each `Row` is a heterogeneous list or `HList` [Kiselyov et al. 2004] indexed by the types of values in that row. In this definition, the type of the constructor (`:>`) indicates that the arguments `f` and `fs` should be erased by the compiler (specified by the keyword `forall`) and that `f` and `fs` should be inferred by the type checker (specified by the `.`).

For example, here is a table containing the names, ages and favorite colors of several students:

```
table :: CSV 3 [String, Int, Color]
table = Frame {
    rows = ("Bob"   :>  12 :> Blue  :> Nil) ::>
           ("Alice" :>  17 :> Green :> Nil) ::>
           ("Eve"   :>  13 :> Red   :> Nil) ::> VNil)
}
```

and a table containing only empty rows, defined using a function `vreplicate` that creates a `Vector` of the appropriate size.

```
empty :: foreach n -> CSV n []
empty n = Frame (vreplicate n Nil)
```

In this definition, the keyword `foreach` in the type means that `n` is needed at runtime so that `vreplicate` knows how long of a vector to make. The `->` means that `n` must be explicitly provided by the programmer. (If the quantification were instead `foreach n.`, then `n` would be a runtime relevant argument that that the compiler must infer.)

Working with this data structure means that all operations must describe the size and types of the frames that they work with. For example, a horizontal composition function, called `concat`, which could be used to concatenate two `DataFrame`s with the same number of rows, might be given the following type.

```
concat :: forall n fs fs'. CSV n fs -> CSV n fs' -> CSV n (fs ++ fs')
```

The type system ensures that the output of `concat` has the same number of rows as its input lists and that the columns are extended properly.

Note that `empty` is an identity element for horizontal table concatenation. Because this table has no columns, for any table `t` of type `CSV n fs`, the expression `concat t empty` should be equal to `t`. However, if we try to use `concat t empty` in a context that expects an expression of type `CSV n fs`, the type checker will complain since `concat t empty` really has the type `CSV n (fs ++ [])`. Because list concatenation is defined through recursion over the list on the left, the expression `fs ++ []` does not reduce to `fs` when `fs` is an abstract variable. Dependently typed languages require evidence that the type `fs ++ []` is equal to the type `fs`.

In (Dependent) Haskell, we can calculate this evidence with a recursive function that returns a *propositional* equality result. Propositional equality is defined in Haskell with the following data type.

```
data a :~: b where
  Refl :: a :~: a
```

With this type, we can write a short function that calculates this evidence. Below, the list `l` is quantified by `foreach` because it is analyzed via pattern matching.

```
appendNilEq :: foreach (l :: [Type]) -> (l ++ []) :~: l
appendNilEq [] = Refl     -- know that l is [] so must show goal
                          --     ([] ++ []) ~ []
                          -- this is true via reduction
appendNilEq (x : xs) =    -- know that l is (x : xs) so must show goal
                          --     ((x:xs) ++ []) ~ x:xs
                          -- LHS reduces to x:(xs ++ [])
```

```
  case appendNilEq xs of -- calculates evidence that xs ++ [] :~: xs
    Refl -> Refl         -- type inference uses this evidence to
                         -- show goal via congruence
```

We note that the concise way of consuming an equality proof may appear surprising to programmers who are used to a more explicit style of eliminating equalities, but it is in fact grounded in the capability of the inference algorithm of today's Haskell implementation, as evidenced by the reformulated version of the `appendNilEq` function using the `singletons` library.

```
appendNilEqSing :: forall (l :: [Type]). Sing l -> (l ++ '[]) :~: l
appendNilEqSing SNil = Refl
appendNilEqSing (SCons x xs) =
  case appendNilEqSing xs of
    Refl -> Refl
```

Despite the minor syntactic differences, neither example requires the programmer to explicitly provide the motive for substituting in the equality.

With this method of evidence calculation in hand, we can use `appendNilEq` in a simplistic example that calls `concat` with an empty frame. This code must create the evidence before the type checker can use it to coerce the type `CSV n (fs ++ [])` to `CSV n fs`.

```
concatNil :: foreach n fs -> CSV n fs -> CSV n fs
concatNil n fs csv =
  case appendNilEq fs of
    Refl -> concat csv (empty n)
```

However, this example use of propositional equality is problematic in Dependent Haskell. While we can get the code to type check, it must do runtime work to compute the evidence necessary for the type coercion. Every time that `concatNil` is called, the code for `appendNilEq` must produce the evidence that is used to coerce the type. Furthermore, in order to produce this evidence, the list of types `fs` must be provided as a runtime argument to `concatNil` and cannot be erased.

2.2 Our Solution: Reflecting Logical Evidence

To address this issue, our solution has two components. First, we adopt the idea of using labels or *modes* to keep track of the termination of terms, first explored in the Zombie Trellys language [Casinghino et al. 2014]. The type system marks some parts of the program as *logical*, using the mode L, to indicate that it is guaranteed to terminate.

Second, we replace the propositional equality type, `:~:`, with a primitive abstract type `:=:` and introduce two new keywords that work with this type. The introduction form, `reify`, instructs type inference to construct evidence for an equality, much like the `Refl` data constructor above. However, instead of being eliminated through pattern matching, evidence of propositional equality can be reflected directly back to the type system. The expression `reflect e1 in e2` takes a logical expression `e1`, which computes an equality of type `a :=: b`, and provides that information to the type checker when type checking `e2`. Unlike case analysis, reflection has no runtime effect.

We use the label L to annotate terms and parameters that construct evidence for equalities. For example, the inductive proof that `(l ++ []) :=: l` can be written as follows.

```
appendNilEq ::ᴸ foreach (l ::ᴸ [Type]) -> ((l ++ []) :=: l)
appendNilEq [] =
  reify
appendNilEq (x:l) =
  reflect (appendNilEq l) in reify
```

The termination labels tells the type checker to ensure that the definition of `appendNilEq` is well-founded and that it can only be applied to inductive lists (i.e. lists from the L fragment). In the inductive case, we no longer need to pattern match on the result of the recursive call. Because the term `appendNilEq l` is in the L fragment, we can directly reflect it to the coercion proof.

Furthermore, we can define an efficient version of `concatNil` that does not require the evaluation of the proof object and treats `fs` as an erased argument.

```
concatNil :: foreach n -> forall fs^L -> CSV n fs -> CSV n fs
concatNil n fs csv =
   reflect (appendNilEq fs) in
   concat csv (empty n)
```

The reflection of the term `appendNilEq` is safe because it is from the logical fragment. It is impossible to provide bogus evidence. Furthermore, reflected code is erasable. Even though the evidence calculation requires `fs`, the operational semantics of our language never evaluates the equality passed to `reflect`. Therefore, even though `appendNilEq` expects a relevant argument, we are allowed pass in the irrelevant argument `fs` since the reflected equality proof appears in an irrelevant position.

2.3 Relevance Tracking through Modes

Dependent Haskell already has a design[1] for using quantifiers to separate runtime and erased arguments. As we saw above, these quantifiers can specify relevance independently from whether an argument is inferred by the type checker.

Under the hood, relevance tracking in Dependent Haskell is based on prior work [Eisenberg 2016; Weirich et al. 2017], that uses the syntactic occurrence of variables in terms to determine relevance [Barras and Bernardo 2008]. But this distinction can be made instead via modes. The type systems EPTS [Mishra-Linger and Sheard 2008], QTT [Atkey 2018] and DDC [Choudhury et al. 2022] mark variables that are not used inside a function body with a label that indicates irrelevance. Because our language already uses modes to keep track of termination, it is convenient to use a similar mechanism to keep track of relevance. This alternative implementation of relevance tracking does not affect the surface language syntax. We can continue to use `forall` and `foreach` quantifiers to indicate relevance.

In dependently-typed languages, relevance tracking is also important at compile time, especially when working with data structures that contain embedded proofs. We would like the type checker to ignore these proofs during equational reasoning.

For example, suppose we would like to represent CSV headers, the names of the columns in a CSV table. However, we would also like the type system to maintain the invariant that the header names are unique so that that we can non-ambiguously access data stored in the table.

We can express this idea with the following data structure. Adding a new column name to a header via `(:+)` requires evidence that the column name is not already present. This evidence is irrelevant, logical, and should be implicitly provided by the compiler.

```
data Header where
   HNil  :: Header
   (:+)  :: foreach (s :: String) (tl :: Header)
         -> forall (pf ::^L s `elem` tl :=: False). Header
```

By making the equality proof implicit, we can easily construct constant `Header`s by treating the `(:+)` constructor as a binary operator.

[1] https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0378-dependent-type-design.rst

We can then use headers to distinguish CSV files that use similar types of columns.

```
data CSV (n :: Nat) (h :: Header) (fs :: [Type]) = ... -- omitted

table :: CSV 3 ("Name" :+ "Age" :+ "Favorite Color" :+ HNil)
               [String, Int, Color]

table2 :: CSV 3 ("Name" :+ "Score" :+ "Least Favorite Color" :+ HNil)
               [String, Int, Color]
```

For example, if we want to add the rows of two tables together, we can require that this operation only works when the tables have the same headers and column types.

```
addRows :: forall n h fs. CSV n h fs -> CSV m h fs -> CSV (n + m) h fs
```

In a call to this function, the type checker must determine that the headers of the two tables are equal. And during this process, we would not want the type checker to have to reason about the equivalence of the embedded uniqueness proofs in the header type. The `forall` quantifier we attach to the equality proof in `Header` not only ensures that the proof is erasable at runtime, but also informs the type checker to ignore the proof when checking for type equality.

2.4 Precise Termination Tracking

In a functional language where the only side effect is nontermination, we are allowed to use potentially diverging terms in types, as long as we are not overly worried about termination of the type checker itself. Haskell has long allowed the possibility for divergent computation at compile time, through the use of the `UndecidableInstances` flag. However, other than disabling the simplistic analysis used in types, Haskell programmers have little control over termination behavior of their code. The transition to Dependent Haskell must accommodate the fact that, by default, the termination behavior of most Haskell code is unknown.

In contrast, in Coq and Agda, code must pass the termination checker by default, which may be disabled through the use of annotations or flags to the type checker. In particular, Agda programmers may annotate individual definitions using the TERMINATING or the NON_TERMINATING pragma; the latter additionally prevents the type checker from unfolding the definition during type equality. Coq programmers have less control, but can use the flag `-type-in-type` to disable universe consistency checking and can disable guardedness checking.

Disabling the type checker to allow programs such as badApp can be dangerous. Because the default behavior is that everything terminates, and because propositional equality is used pervasively, the OCaml extraction backend for Coq, the MAlonzo compiler backend for Agda, and the Idris compiler all aggressively erase equality proofs, even those that could diverge. As a result, an extracted program, like badApp, could crash with a type error at runtime rather than entering an infinite loop.

Disabling the termination checker on a per-definition basis, as is permitted by Agda, is infective. As a result, any code defined in the scope of such definitions could diverge, even if it passes Agda's termination checker. To guarantee consistency, Agda provides the coinfective `--safe` flag, which ensures that a module and its dependencies do not contain features that may violate consistency. However, the `--safe` flag conservatively rejects a program even if a partial function from the dependencies is never used in the module declared as safe. Idris is more similar to Dependent Haskell. The totality checker is opt-in and can be enabled for an entire module. While the compiler ensures the dependencies of a total function are also total, it will not complain if an imported module contains a partial function as long as that partial function is never used in the body of a total function. However, tracking termination with modes provides even more precise analysis.

Modes			
ρ	$::=$	$R \mid I$	relevance, (R)elevant and (I)rrelevant, $R < I$
θ	$::=$	$L \mid P$	termination, (L)ogic and (P)rogram, $L < P$
δ	$::=$	ρ, θ	combined, pointwise ordering
Contexts			
Γ	$::=$	$\cdot \mid \Gamma, x :^{\delta} A$	variables annotated by mode
Terms			
a, b, A, B	$::=$	$\star \mid x \mid \Pi x :^{\delta} A.B$	sort, variables and function types
	$\mid$	$\lambda^{\delta} x : A.a \mid a\ b^{\delta}$	abstractions and applications
	$\mid$	$a \equiv^{\theta} b \mid \textbf{reify}^{\theta}\ \gamma$	equality type and proof terms
	$\mid$	$a \rhd \gamma$	coerced terms
	$\mid$	$\mathbb{N} \mid \textbf{zero} \mid \textbf{succ}\ a$	natural number type and constructors
	$\mid$	$\textbf{ind}_A\ a\ b_1\ (x.b_2)$	natural number induction, see Section 3.5
Coercions			
γ	$::=$	$\textbf{reflect}\ a$	equality reflection
	$\mid$	$\ldots$	others, see Section 3.3 and Appendix A.1

Fig. 1. Syntax of System DE

Type system			
$\Gamma \vdash a :^{\theta} A$	*Typing*	Figures 3 and 6	Appendix A.2.1
$\Gamma \vdash a :^{\delta} A$	*Typing, relevance-moded*	Figure 3	Appendix A.2.2
$\vdash \Gamma$	*Context formation*	Figure 3	Appendix A.2.3
$\Gamma; \Gamma_0 \vdash^{\theta} \gamma : a \sim b$	*Equality proofs*	Figure 4	Appendix A.2.4
Operational semantics			
$\Gamma \vdash^{\theta} a \rightsquigarrow b$	*Computational reduction*	Figure 5	Appendix A.3.1
$\Gamma \vdash^{\theta} a \rightharpoonup b$	*Administrative reduction*	Figure 5	Appendix A.3.3

Fig. 2. Summary of System DE judgement forms

Because modalities are part of the type system, the compiler can restrict the termination behavior of all arguments that can be supplied to a function, even if those arguments are not present when the function is compiled. Furthermore, our language allows reasoning of diverging programs. Given a diverging computation `loop`, The keyword `reify` can witness the equality `loop :=: loop` while still belonging to the logical fragment. The equality can be safely erased without affecting type soundness because the diverging term only appears inside the equality type. The proof of the equality, on the other hand, is already in normal form.

3 SYSTEM DE

In this section, we describe System DE, a dependently-typed language that supports relevance tracking and sound equational reasoning through a logical sublanguage. These two features are governed by the modes ρ and θ respectively. We use the metavariable δ to refer to the combination of these two modes and annotate variables in the context and bound variables in the term syntax with δ. The syntax of System DE appears in Figure 1.

Our formalization of System DE includes dependently-typed functions, an equality type, and natural numbers. We defer discussion of natural number induction until Section 3.5. Due to the presence of the $\star : \star$ axiom, not all programs terminate.[2] However, type conversions require an explicit coercion proof γ, which ensures that type checking terminates. The type system is syntax directed; given an input context Γ, mode θ and term a, we can write an algorithm that decides whether the term type checks and if so, determines its unique type. For mechanization reasons described in section 4, our formal system also includes a mode annotation in function applications $a\ b^{\delta}$, but because this mode can be uniquely determined by the type of the function, we omit it from examples.

3.1 A Type System with Dependency Tracking

The semantics of System DE are specified by the judgement forms listed in Figure 2. For many of the judgments, the figures in the main text provide an excerpt of the rules, deferring the listing of the full system to the Appendix.

The modes ρ and θ extend the System DE type system with *dependency tracking*. Modes are ordered and programs checked at one mode cannot depend on subexpressions that are marked with a higher mode. Dependency tracking works for both termination and relevance, but in subtly different ways as we explain below. Relevance tracking in System DE is inspired by the DDC type system developed by Choudhury et al. [2022] while our approach to termination checking is a variation of the type system developed by Casinghino et al. [2014].

The typing judgment (Figure 3) is annotated by a termination mode, θ, and has the general form $\Gamma \vdash a :^{\theta} A$. This mode divides the language into two fragments: P, a flexible programming language with unrestricted dependency between terms and types and L, a restricted proof language that can be used to reason about both fragments in a consistent way. In a derivation of $\Gamma \vdash a :^{L} A$, the typing rules ensure that the term a belongs to the logical fragment. Otherwise, when the mode is P, the term can use the full generality of the programming language. For example, the typing rule T-TYPE states that the $\star : \star$ axiom is only available in the programming fragment. Terms that type check with mode L may not depend on code that uses this rule.

The ordering $L < P$ means that logical terms can be used to compute program values, but not vice versa. Our consistency theorem (Lemma 4.18) requires the L fragment of our language to be weakly normalizing. It is therefore crucial to keep nontermination out of the fragment we use for equality proofs. On the other hand, programs that type check under the L fragment can be used freely in the P fragment.

The mode ρ distinguishes between the relevant and irrelevant parts of the computation. Irrelevant arguments, annotated with I, are not necessary at runtime and can be ignored at compile time when checking type equivalence. The ordering between modes $R < I$ means that relevant arguments can be used to compute irrelevant values but not vice versa. Note that there is no ρ on the typing judgment itself. Implicitly, the relevance mode of a is always R because a is the subject of the computation.

[2] A programming language, such as Haskell, includes many sources of inconsistency, such as general recursion, datatypes with negative occurrences of the recursive type, etc. For simplicity, we include only one source of inconsistency in System DE, but our mechanisms extend to all of these features.

$\boxed{\Gamma \vdash a :^{\theta} A}$ *(Typing)*

$$\textsc{T-Type}\quad \frac{\vdash \Gamma}{\Gamma \vdash \star :^{P} \star} \qquad \textsc{T-Var}\quad \frac{\theta_0 \leq \theta \qquad \vdash \Gamma \qquad x :^{R,\theta_0} A \in \Gamma}{\Gamma \vdash x :^{\theta} A} \qquad \textsc{T-Pi}\quad \frac{\Gamma \vdash A :^{\theta_0} \star \qquad \Gamma, x :^{R,\theta_0} A \vdash B :^{\theta} \star}{\Gamma \vdash \Pi x :^{\rho_0,\theta_0} A.B :^{\theta} \star}$$

$$\textsc{T-Abs}\quad \frac{\widetilde{\Gamma} \vdash \Pi x :^{\delta_0} A.B :^{\theta} \star \qquad \Gamma, x :^{\delta_0} A \vdash b :^{\theta} B}{\Gamma \vdash \lambda^{\delta_0} x : A.b :^{\theta} \Pi x :^{\delta_0} A.B} \qquad \textsc{T-App}\quad \frac{\Gamma \vdash b :^{\theta} \Pi x :^{\delta_0} A.B \qquad \Gamma \vdash a :^{\delta_0} A}{\Gamma \vdash b\ a^{\delta_0} :^{\theta} B\{a/x\}}$$

$$\textsc{T-Eq}\quad \frac{\Gamma \vdash a :^{\theta_0} A \qquad \Gamma \vdash b :^{\theta_0} A}{\Gamma \vdash a \equiv^{\theta_0} b :^{\theta} \star} \qquad \textsc{T-Conv}\quad \frac{\Gamma \vdash a :^{\theta} A \qquad \widetilde{\Gamma}; \cdot \vdash^{\theta} \gamma : A \sim B}{\Gamma \vdash a \rhd \gamma :^{\theta} B} \qquad \textsc{T-Reify}\quad \frac{\Gamma; \cdot \vdash^{\theta_0} \gamma : a \sim b}{\Gamma \vdash \mathbf{reify}^{\theta_0}\ \gamma :^{\theta} (a \equiv^{\theta_0} b)}$$

$\boxed{\Gamma \vdash a :^{\delta} A}$ *(Typing, relevance-moded)*

$$\textsc{CT-Leq}\quad \frac{\Gamma \vdash a :^{\theta} A}{\Gamma \vdash a :^{R,\theta} A} \qquad \textsc{CT-Top}\quad \frac{\widetilde{\Gamma} \vdash a :^{\theta} A}{\Gamma \vdash a :^{I,\theta} A}$$

$\boxed{\vdash \Gamma}$ *(Context formation)*

$$\textsc{T-Empty}\quad \frac{}{\vdash \cdot} \qquad \textsc{T-ConsTm}\quad \frac{\vdash \Gamma \qquad \widetilde{\Gamma} \vdash A :^{\theta} \star \qquad x \notin \operatorname{dom} \Gamma}{\vdash \Gamma, x :^{\rho,\theta} A}$$

Fig. 3. Typing rules of System DE

The ordering between modes is reflected by the following properties of the typing judgement.[3] First, we can weaken the assumptions of variables in the context according to their modes. The relation $\Gamma_1 \leq \Gamma_2$ extends the ordering between δ to a pointwise ordering between contexts.

Lemma 3.1 (Narrowing[4]). *If $\Gamma_2 \vdash a :^{\theta} A$, $\vdash \Gamma_1$, and $\Gamma_1 \leq \Gamma_2$, then $\Gamma_1 \vdash a :^{\theta} A$.*

Next, fragment subsumption allows logical terms to be used in programs.

Lemma 3.2 (Subsumption for θ[5]). *If $\Gamma \vdash a :^{L} A$, then $\Gamma \vdash a :^{P} A$*

3.2 Dependent Types and Relevance Tracking

In this section, we provide an overview of the individual typing rules in Figure 3 and highlight how they make use of modes for relevance tracking.

[3] Footnotes in this and the next section refer to the source file and name for the corresponding lemma in our companion artifact [Liu and Weirich 2023]. The language definition itself can be found in the file lp.ott, which has been translated to Coq using the OTT tool [Sewell et al. 2010]. [4] lp_narrowing.v:typing_narrowing [5] subsumption.v:typing_subsumption

Rule T-Var ensures that a variable x can be used as long as it has been marked relevant in the context and when its attached termination mode θ_0 is less than or equal to the θ from the judgment. Variables are introduced into the context in the rules for (dependent) function types (rule T-Pi) and abstractions (rule T-Abs). Variables in the context that are marked irrelevant do not type check with rule T-Var. However, as we describe below, these variables may be used in irrelevant subterms, through the operation of resurrection described below.

A function type has the form $\Pi x :^{\delta} A.B$, with the input type annotated by δ. When x does not appear in B, we also use the notation $A \overset{\delta}{\to} B$. For the θ component of δ, the interaction with dependent functions is straightforward. However, for relevance tracking, note that the relevance component of δ in a dependent function type refers to the usage of the variable in the body of the function itself, and not in the body of the type. For this reason, in rule T-Pi, we add the variable to the context as R and not ρ_0 following Choudhury et al. [2022]. As a result, in the type polymorphic identity function $\Pi x :^{I,P} \star .x \longrightarrow x$, the abstracted type variable x can appear in the body of the type.

In contrast, rule T-Abs ensures that a function marked with mode I is not allowed to use its input in a relevant position because the parameter enters the context with the marked mode when checking the body of the function. This type of relevance tracking is more precise than merely checking whether the input variable occurs in the body of a function. To see why, consider the following example:

$$a = \lambda^{I,\theta_0} x : \mathbb{N}. b\, x \qquad\qquad b = \lambda^{I,\theta_0} y : \mathbb{N}. \mathbf{zero}$$

In a, the irrelevant argument x appears in its body. However, the use of x is safe because it appears in an irrelevant position as an argument to the function b, which is a constant function. Because b does not use its argument, it can be marked as irrelevant. When x appears in a as an irrelevant argument to another function b, the obligation of ensuring that the x is not used is shifted to the function b.

The rule T-App precisely captures this reasoning. When we apply a function b to an argument a with label δ_0, we delegate the type checking of a to the relevance-moded typing relation. The two rules of the judgment $\Gamma \vdash a :^{\delta} A$, shown at the bottom of Figure 3, employ a context operation called *resurrection* [Mishra-Linger and Sheard 2008; Pfenning 2001] when the ρ component of δ is I. In this case, rule CT-Top "resurrects" the context Γ into the context $\widetilde{\Gamma}$ before type checking.

This operation shifts the point of view when type checking irrelevant locations in the term. Given a context Γ, the resurrected context $\widetilde{\Gamma}$ is a context that is identical to Γ but has all relevance modes replaced with R. As a result, previously irrelevant variables are available for use by rule T-Var in those locations.

Resurrection is also used in other rules to check irrelevant subterms, such as when checking the function type in rule T-Abs and when checking the type A in the context well-formedness rule T-ConsTm. This makes sense because evaluation does not depend on these parts of the term.

With the relevance-moded typing judgment, we can state a general subsumption theorem that includes both modes and expresses the allowed dependencies.

Lemma 3.3 (Subsumption[6]). *If* $\Gamma \vdash a :^{\delta_0} A$ *and* $\delta_0 \leq \delta_1$ *then* $\Gamma \vdash a :^{\delta_1} A$

This lemma depends on the following property, which asserts that relevant terms can also be used in irrelevant contexts.

Lemma 3.4 (Resurrection[7]). *If* $\Gamma \vdash a :^{\theta} A$ *then* $\widetilde{\Gamma} \vdash a :^{\theta} A$

We can further show that our type system has the following regularity property: the type of term checks in the same fragment but uses a resurrected context.

[6] subsumption.v:typing_subsumption [7] lp_narrowing.v:typing_meet_ctx_l

$\boxed{\Gamma;\Gamma_0 \vdash^{\theta} \gamma : a \sim b}$ *(Equality proofs)*

E-Reflect

$$\frac{\theta_0 \leq \theta \qquad \vdash \Gamma,\Gamma_0 \qquad \Gamma \vdash a_0 :^{L} a \equiv^{\theta_0} b \qquad \Gamma \vdash a :^{\theta} A \qquad \Gamma \vdash b :^{\theta} A}{\Gamma;\Gamma_0 \vdash^{\theta} \mathbf{reflect}\, a_0 : a \sim b}$$

E-PiCong

$$\frac{\begin{array}{c}\Gamma;\Gamma_0 \vdash^{\theta_0} \gamma_1 : A_1 \sim A_2 \qquad \Gamma;\Gamma_0, x :^{R,\theta_0} A_1 \vdash^{\theta} \gamma_2 : B_1 \sim B_2 \\ \Gamma,\Gamma_0 \vdash \Pi x :^{\rho_0,\theta_0} A_1 . B_1 :^{\theta} \star \qquad \Gamma,\Gamma_0 \vdash \Pi x :^{\rho_0,\theta_0} A_1 . B_2 :^{\theta} \star \qquad B_3 = B_2\{x \rhd \mathbf{sym}\, \gamma_1 / x\}\end{array}}{\Gamma;\Gamma_0 \vdash^{\theta} \Pi x :^{\rho_0,\theta_0} \gamma_1 . \gamma_2 : \Pi x :^{\rho_0,\theta_0} A_1 . B_1 \sim \Pi x :^{\rho_0,\theta_0} A_2 . B_3}$$

E-AbsCong

$$\frac{\widetilde{\Gamma,\Gamma_0} \vdash A :^{\theta_0} \star \qquad \Gamma;\Gamma_0, x :^{\rho_0,\theta_0} A \vdash^{\theta} \gamma_2 : a_1 \sim a_2 \qquad \Gamma,\Gamma_0 \vdash (\lambda^{\rho_0,\theta_0} x : A . a_2) :^{\theta} B}{\Gamma;\Gamma_0 \vdash^{\theta} \lambda^{\rho_0,\theta_0} x : A . \gamma_2 : \lambda^{\rho_0,\theta_0} x : A . a_1 \sim \lambda^{\rho_0,\theta_0} x : A . a_2}$$

E-AppCong

$$\frac{\begin{array}{c}\Gamma;\Gamma_0 \vdash^{\theta} \gamma_1 : a_1 \sim a_2 \\ \Gamma;\Gamma_0 \vdash^{\theta_0} \gamma_2 : b_1 \sim b_2 \qquad \Gamma,\Gamma_0 \vdash a_1\ {b_1}^{R,\theta_0} :^{\theta} A \qquad \Gamma,\Gamma_0 \vdash a_2\ {b_2}^{R,\theta_0} :^{\theta} B \qquad \widetilde{\Gamma,\Gamma_0}; \cdot \vdash^{\theta} \gamma : A \sim B\end{array}}{\Gamma;\Gamma_0 \vdash^{\theta} \gamma_1\ \gamma_2^{+} \rhd \gamma : (a_1\ {b_1}^{R,\theta_0}) \rhd \gamma \sim a_2\ {b_2}^{R,\theta_0}}$$

E-AppCongIrrel

$$\frac{\begin{array}{c}\Gamma;\Gamma_0 \vdash^{\theta} \gamma_1 : a_1 \sim a_2 \qquad \widetilde{\Gamma,\Gamma_0} \vdash b_1 :^{\theta_0} A \\ \widetilde{\Gamma,\Gamma_0} \vdash b_2 :^{\theta_0} A \qquad \Gamma,\Gamma_0 \vdash a_1\ {b_1}^{I,\theta_0} :^{\theta} B_1 \qquad \Gamma,\Gamma_0 \vdash a_2\ {b_2}^{I,\theta_0} :^{\theta} B_2 \qquad \widetilde{\Gamma,\Gamma_0}; \cdot \vdash^{\theta} \gamma : B_1 \sim B_2\end{array}}{\Gamma;\Gamma_0 \vdash^{\theta} \gamma_1\ (b_1\ b_2)^{-} \rhd \gamma : (a_1\ {b_1}^{I,\theta_0}) \rhd \gamma \sim a_2\ {b_2}^{I,\theta_0}}$$

Fig. 4. Coercion proofs (selected rules)

Lemma 3.5 (Regularity[8]). *If* $\Gamma \vdash a :^{\theta} A$ *then* $\widetilde{\Gamma} \vdash A :^{\theta} \star$ *or* $A = \star$.

The relevance tracking in System DE applies to the term level only; some variables may be marked irrelevant in the context when checking a, but may be used in relevant positions in the type A. Therefore resurrection is required to ensure that these variables are accessible. On the other hand, the termination mode θ must stay the same for type A.

3.3 Decidable Type Checking through Logical Coercion Proofs

Type checking in System DE is decidable and straightforward, through the use of typing annotations and *explicit coercion proofs*, notated γ, and inspired by System FC [Sulzmann et al. 2007] and its dependently-typed variant System DC [Weirich et al. 2017]. One difficulty of including potentially nonterminating terms in types is that determining whether they are equal is not decidable. Therefore, like System FC, we maintain decidability of type checking by requiring evidence γ in rule T-Conv. This evidence includes a trace of execution that can be checked and need not be inferred. Because of this explicit evidence, System DE admits the unique typing property.

Lemma 3.6 (Unique typing[9]). *If* $\Gamma \vdash a :^{\theta} A$ *and* $\Gamma \vdash a :^{\theta} B$, *then* $A = B$.

[8] regularity.v:typing_regularity [9] typing_unique.v:typing_unique

Coercion proofs are checked by the judgment of the form $\Gamma; \Gamma_0 \vdash^{\theta} \gamma : a \sim b$, partially shown in Figure 4. This judgment roughly corresponds to the reflexive, symmetric, transitive, and compatible closure of the reduction relations we will see in Section 3.4 *and* includes reflected proofs from the logical fragment (rule E-Reflect). The termination mode θ on this judgement refers to the fragments for a and b (coercion proofs themselves are annotations for the type checker and are never evaluated). The complete set of rules appears in Appendix A.2 and many rules are similar to analogous rules in Systems FC and DC.

However, a significant difference between System DE and prior systems is that coercion proofs can contain logical terms and are not isolated from other parts of the computation. This means that System DE needs only one form of abstraction: modes identify which terms can be used as evidence not syntax. The typing rule T-Reify reifies a coercion proof γ as a witness for the equality type $A \equiv^{\theta} B$. These values are first class, and can be manipulated using the features of the programming language.

Conversely, rule E-Reflect allows logical terms of the equality type to be *reflected* back as coercions. In the simplest case, these reflected terms may be proofs that were previously reified, or they may be variables of the appropriate type. They may also be the result of any computation in the logical language that produces a value of the equivalence type. As shown in Section 2, this design significantly increases the expressiveness of coercions compared to System DC.

To see how the `reify` and `reflect` constructs from the surface language are elaborated into System DE, consider the following proof that $y + 0 = y$ for all natural numbers y.

```
plusNZ ::ᴸ foreach (y ::ᴸ Nat) -> y + Z :=: y
plusNZ [] = reify
plusNZ (S y) = reflect z in reify
  where z :: y + Z :=: y
        z = plusNZ y
```

As usual, + is defined by recursion over its first argument. Therefore, the structure of `plusNZ` is almost identical to that of `appendNilEq`.

The elaborated System DE uses the terminating recursor over natural numbers that we describe in Section 3.5. This logical term encodes an inductive proof and requires the base case a_1 and inductive step a_2 shown below.

$$a_1 : \mathbf{zero} + \mathbf{zero} \equiv^{L} \mathbf{zero} \qquad a_2 : (y + \mathbf{zero} \equiv^{L} y) \longrightarrow (\mathbf{succ}\ y + \mathbf{zero} \equiv^{L} \mathbf{succ}\ y)$$
$$a_1 = \mathbf{reify}^{L}\ \gamma_1 \qquad a_2 = \lambda z.\mathbf{reify}^{L}\ (\ldots \mathbf{reflect}\ z \ldots)$$

The coercion proof γ_1 in the base case a_1 witnesses the explicit reduction sequence from **zero** + **zero** to **zero** and can be automatically constructed by the type checker. The induction case, a_2, is more interesting. This term takes the inductive hypothesis (a variable z of type $y + \mathbf{zero} \equiv^{L} y$) and returns a proof of type $\mathbf{succ}\ y + \mathbf{zero} \equiv^{L} \mathbf{succ}\ y$. In this case, the left hand side and right hand side are not beta-equivalent. Instead, the reflected input z is part of a larger (elided) coercion that witnesses the equality between $\mathbf{succ}\ y + \mathbf{zero}$ and $\mathbf{succ}\ y$. This coercion can also be automatically constructed by the type checker.

The "`reflect...in...`" form from the surface language makes a logical equality proof available in a specific scope, instructing the type inferencer to **reflect** that proof when constructing coercion evidence. In the absence of reflected equality proofs, coercion proofs may only use reduction rule and congruence rules. By reflecting logical terms, such as z, the user can guide the type inferencer to more sophisticated reasoning.

There are limits on what logical terms can be reflected into coercion proofs—these terms must not refer to variables that were introduced into the context as part of congruence rules. For this reason, the judgment maintains two separate contexts: Γ and Γ_0. The local context Γ_0 is populated with variables introduced through congruence rules that involve binders such as rule E-AbsCong and rule E-PiCong. In rule E-Reflect, the reflected term must be typeable using the global context Γ only. In the earlier example, we can use **reflect** z inside the coercion proof of a_2 because the variable z is introduced into the global context through rule T-Abs. The reason for this restriction comes from our consistency proof, which we discuss in Section 4.2.

Another significant difference between System DC and System DE's coercion judgments is that the equivalence rules and operational semantics are stated without the use of an erasure operation. Instead, this system includes two congruence rules for applications. When the argument is relevant, rule E-AppCong requires evidence that the arguments are equal. On the other hand, in applications that involve irrelevant arguments, rule E-AppCongIrrel only requires the arguments to be well-formed and the overall types to be equal.

Unlike System DC, the typed equivalence relation is homogeneous. The two terms that we want to equate must have the exactly same type in the θ fragment. In other words, the relation admits following regularity property:

Lemma 3.7 (DefEq regularity[10]). *If* $\Gamma; \Gamma_0 \vdash^{\theta} \gamma : a \sim b$*, then there exists some* A *such that* $\Gamma, \Gamma_0 \vdash a :^{\theta} A$ *and* $\Gamma, \Gamma_0 \vdash b :^{\theta} A$.

Above, we use Γ, Γ_0 to indicate the concatenation of the contexts Γ and Γ_0.

Making this equality homogeneous makes it easy to work with as a congruence: at any point we can substitute related terms because the two terms have the same type. However, there is a cost: combining homogeneous equality with the explicit conversion rule incurs a small bit of extra bookkeeping in constructing proofs to make sure the terms on both sides share the same type. In rule E-AppCong, in addition to showing the equivalence between the functions a_1 and a_2 and their respective arguments b_1 and b_2, it is necessary to supply an extra coercion proof γ that witnesses the equality between the types of $a_1 \;\; {b_1}^{R,\theta_0}$ and $a_2 \;\; {b_2}^{R,\theta_0}$. Because System DE has dependent function types, changing the input can change the resulting type of the application. A similar asymmetry appears in rule E-PiCong where the quantified variable must be coerced. In contrast, rule E-AbsCong may require the domain types to be identical without loss of expressiveness, because an administrative reduction (rule CR-AbsPush, Figure 5) can be used instead.

3.4 Operational Semantics of System DE

Since System DE requires explicit coercion objects, we need to decide what to do with the coercions when we run into them during reduction. It is convenient to divide our reduction relation into two separate relations—an administrative reduction relation that is used exclusively for shuffling around the coercions and a computational reduction relation that performs more conventional reductions. Figure 5 shows the definitions of the two relations.

Before understanding how the reduction relation works, we need to first look at how *values* and *covalues* are defined in our language (see grammar at top of Figure 5). The definition of a value is standard for a call-by-name language with the exception that natural numbers are strict. System DE also has the concept of a coerced value or covalue, a term that consists of a value or a successor form nested in an arbitrary (possibly zero) number of coercions. Intuitively, a covalue is a term that is not necessarily in head form but is equal to a value once the coercions are erased.

In an application, the computational reduction relation reduces the argument of an application first to a covalue using rule R-App. Then in rule R-AppAbs, it switches to the administrative relation to "push" any coercions surrounding the abstraction inside, exposing the lambda at the top level

[10] regularity.v:defeq_regularity

$$
\begin{array}{llll}
\textit{values} & & & \\
v & ::= & \star \mid \Pi x{:}^{\delta} A.B\ a \equiv^{\theta} b \mid \mathbb{N} & \text{types} \\
 & \mid & \lambda^{\delta} x{:}A.a \mid \mathbf{reify}^{\theta}\ \gamma & \text{functions and reified coercions} \\
 & \mid & \mathbf{zero} \mid \mathbf{succ}\ v & \text{naturals} \\
 & & & \\
\textit{covalues} & & & \\
c & ::= & v \mid c \rhd \gamma \mid \mathbf{succ}\ c &
\end{array}
$$

$\boxed{\Gamma \vdash^{\theta} a \rightsquigarrow b}$ *(Computational reduction)*

R-AppAbs
$$\dfrac{\begin{array}{c} a \textit{ is a covalue} \\ \Gamma \vdash^{\theta} a \rightharpoonup^{*} \lambda^{\delta_0} x{:}A.a_1 \end{array}}{\Gamma \vdash^{\theta} a\ b^{\delta_0} \rightsquigarrow a_1\{b/x\}}$$

R-App
$$\dfrac{\Gamma \vdash^{\theta} a_1 \rightsquigarrow a_2}{\Gamma \vdash^{\theta} a_1\ b^{\delta_0} \rightsquigarrow a_2\ b^{\delta_0}}$$

R-Conv
$$\dfrac{\Gamma \vdash^{\theta} a_1 \rightsquigarrow a_2}{\Gamma \vdash^{\theta} a_1 \rhd \gamma \rightsquigarrow a_2 \rhd \gamma}$$

$\boxed{\Gamma \vdash^{\theta} a \rightharpoonup b}$ *(Administrative reduction)*

CR-AbsPush
$$\dfrac{\begin{array}{c} \Gamma \vdash (\lambda^{\delta_0} x{:}A_1.a_1) \rhd \gamma :^{\theta} A \qquad \theta_0 \leq \theta \qquad \widetilde{\Gamma}; \cdot \vdash^{\theta_0} \gamma : (\Pi x{:}^{\delta_0} A_1.B_1) \sim (\Pi x{:}^{\delta_0} A_2.B_2) \\ a_2 = a_1\{x \rhd \mathbf{sym}\ (\mathbf{pifst}^{\theta_0} \gamma)/x\} \qquad \gamma_2 = \gamma^{\theta_0} @ (\mathbf{reflex}\ x) \rhd (\mathbf{sym}\ (\mathbf{pifst}^{\theta_0} \gamma)) \end{array}}{\Gamma \vdash^{\theta} (\lambda^{\delta_0} x{:}A_1.a_1) \rhd \gamma \rightharpoonup \lambda^{\delta_0} x{:}A_2.(a_2 \rhd \gamma_2)}$$

CR-ConvRefl
$$\dfrac{\widetilde{\Gamma}; \cdot \vdash^{\theta} \gamma : A \sim A}{\Gamma \vdash^{\theta} a \rhd \gamma \rightharpoonup a}$$

CR-ConvCong
$$\dfrac{\Gamma \vdash^{\theta} a \rightharpoonup b}{\Gamma \vdash^{\theta} a \rhd \gamma \rightharpoonup b \rhd \gamma}$$

CR-Combine
$$\dfrac{}{\Gamma \vdash^{\theta} (a \rhd \gamma_1) \rhd \gamma_2 \rightharpoonup a \rhd (\gamma_1 ; \gamma_2)}$$

Fig. 5. Operational semantics

of the term for a β-reduction. We use the notation $\Gamma \vdash^{\theta} a \rightharpoonup^{*} b$ for zero or more administrative reductions.

As an example of an evaluation, consider the following term:

$$((\lambda^{R,\theta_0} x{:}\mathbb{N}.x) \rhd \gamma)\ \mathbf{zero}$$

where γ is defined as a trivial reflexivity proof $\mathbf{reflex}\ (\Pi x{:}^{R,\theta_0} \mathbb{N}.\mathbb{N})$. The term is in the form of a covalue applied to an argument **zero**. To take a step, we apply rule R-AppAbs, which first applies one step of the rule CR-AbsPush to push the coercion object inside, obtaining the following lambda term:

$$\lambda^{R,\theta_0} x{:}\mathbb{N}.((x \rhd (\mathbf{sym}\ (\mathbf{pifst}^{\theta_0} \gamma))) \rhd (\gamma^{\theta_0} @ (\mathbf{reflex}\ x) \rhd (\mathbf{sym}\ (\mathbf{pifst}^{\theta_0} \gamma))))$$

With the lambda term now available, we can finally substitute in the argument **zero** and obtain the following covalue as the result of the computational reduction:

$$(\mathbf{zero} \rhd (\mathbf{sym}\ (\mathbf{pifst}^{\theta_0} \gamma))) \rhd (\gamma^{\theta_0} @ (\mathbf{reflex}\ \star) \rhd (\mathbf{sym}\ (\mathbf{pifst}^{\theta_0} \gamma)))$$

It is possible to apply a few more steps of the administrative reduction relation to reduce the resulting term to **zero**. However, we specifically design the system such that the reduction relation never takes a step when the term is already a covalue. Instead, we delay the reduction of a covalue into a value until we need the term to be in the form of a value before we can make progress. This is not out of consideration of performance. Administrative reduction rules merely shuffle

$\boxed{\Gamma \vdash a :^{\theta} A}$ *(Typing)*

$$\textsc{T-Nat}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbb{N} :^{\theta} \star} \qquad \textsc{T-Zero}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{zero} :^{\theta} \mathbb{N}} \qquad \textsc{T-Succ}\ \frac{\Gamma \vdash a :^{\theta} \mathbb{N}}{\Gamma \vdash \mathbf{succ}\, a :^{\theta} \mathbb{N}}$$

$$\textsc{T-Ind}\ \frac{\begin{array}{c}\widetilde{\Gamma} \vdash \Pi x :^{R,L} \mathbb{N}. A :^{L} \star \\ \Gamma \vdash a_1 :^{L} \mathbb{N} \\ \Gamma \vdash a_2 :^{L} A\{\mathbf{zero}/x\} \\ \Gamma, y :^{R,L} \mathbb{N} \vdash a_3 :^{L} A\{y/x\} \xrightarrow{R,L} A\{\mathbf{succ}\, y/x\}\end{array}}{\Gamma \vdash \mathbf{ind}_{(\Pi x^{R,L} \mathbb{N}. A)}\, a_1\, a_2\, (y.a_3) :^{\theta} A\{a_1/x\}}$$

Fig. 6. Typing rules for natural numbers

around the coercion objects so our term remains well-typed when we perform the more interesting reduction rules such as beta reduction of lambda terms and induction on natural numbers. This way of structuring the reduction relations not only helps us express our type soundness property (Theorem 4.22), but also makes it easier to extend our language to support call-by-value functions (Section 5).

To show that both coercions and irrelevant terms can be safely erased, we prove the forward simulation property between the computational reduction semantics of System DE and the reduction semantics of a simple, untyped, and unannotated language. The erasure operation[11] removes all modes and coercions and replaces all irrelevant arguments with a placeholder term (written $\square$). The following property states that the administrative reduction relation preserves the erased form.

Lemma 3.8 (Simulation (Co)[12]). *If* $\Gamma \vdash^{\theta} a \rightharpoonup b$, *then* $|a| = |b|$.

We can then derive the lock-step simulation property between the System DE terms and an operational semantics for erased terms[13].

Lemma 3.9 (Simulation[14]). *If* $\Gamma \vdash^{\theta} a \rightsquigarrow b$, *then* $|a| \rightsquigarrow |b|$.

We omit the definition of the reduction relation for the erased language since it directly corresponds to the reduction rules of System DE except for the lack of annotations and administrative steps for pushing coercions. Lemma 3.9 holds only because we distinguish between the rules that perform "interesting" reductions and the rules for shuffling around coercions. This property justifies our decision to treat terms appearing in coercion proofs as irrelevant; coercions appear in terms and even interact with the typed reduction relation, but they do not affect the behavior of the program or even lead to additional evaluation steps after erasure.

3.5 Natural Number Induction

Figure 6 includes the typing rules for natural numbers. The term $\mathbf{ind}_A\, a\, b_1\, (y.b_2)$ is a terminating recursor over the natural number argument a, using b_1 when it is **zero** and calling itself recursively in the successor case. It is annotated by the result type A. The typing rule allows it to be used as in an inductive proof, refining the result type in the zero and successor cases. For simplicity, this term is limited to the logical fragment as its primary purpose is for inductive proofs.

Rule R-Ind, the congruence reduction rule for the induction form, evaluates an induction form into a coerced term.

[11] Appendix B.1 [12] simulation.v:cored_simulation [13] Appendix B.2 [14] simulation.v:red_simulation

R-Ind

$$\frac{\begin{array}{c}\Gamma \vdash^{L} a_1 \leadsto b_1 \\ \Gamma \vdash \mathbf{ind}_{(\Pi x :^{R,L} \mathbb{N}.A)}\ a_1\ a_2\ (y.a_3) :^{\theta} B_0 \\ \Gamma \vdash \mathbf{ind}_{(\Pi x :^{R,L} \mathbb{N}.A)}\ b_1\ a_2\ (y.a_3) :^{\theta} B_1 \\ \widetilde{\Gamma}; \cdot \vdash^{\theta} \gamma : B_1 \sim B_0\end{array}}{\Gamma \vdash^{\theta} (\mathbf{ind}_{(\Pi x :^{R,L} \mathbb{N}.A)}\ a_1\ a_2\ (y.a_3)) \leadsto (\mathbf{ind}_{(\Pi x :^{R,L} \mathbb{N}.A)}\ b_1\ a_2\ (y.a_3)) \rhd \gamma}$$

The coercion proof is necessary for the preservation lemma to hold since the type of the induction form is dependent on the scrutinee. In rule T-Ind, when a_1 steps into b_1, the type of the induction form also changes from $A\{a_1/x\}$ to $A\{b_1/x\}$. The coercion proof γ serves as a witness between those two types.

The following admissible rule[15] shows that given $\Gamma \vdash^{L} a_1 \leadsto b_1$, it is always possible to construct some γ such that $\Gamma; \cdot \vdash^{\theta} \gamma : A\{b_1/x\} \sim A\{a_1/x\}$ in order to step a term through rule R-Ind.

R-IndAlt

$$\frac{\Gamma \vdash^{L} a_1 \leadsto b_1 \qquad \Gamma \vdash \mathbf{ind}_{(\Pi x :^{R,L} \mathbb{N}.A)}\ a_1\ a_2\ (y.a_3) :^{\theta} A\{a_1/y\}}{\Gamma \vdash \mathbf{ind}_{(\Pi x :^{R,L} \mathbb{N}.A)}\ a_1\ a_2\ (y.a_3) \leadsto^{\theta} (\mathbf{ind}_{(\Pi x :^{R,L} \mathbb{N}.A)}\ b_1\ a_2\ (y.a_3)) \rhd \gamma}$$

4 TYPE SOUNDNESS

We have proven, in Coq, that this language is type sound using the preservation and progress lemmas. Of these two results, preservation is more similar to previous work [Choudhury et al. 2022; Sulzmann et al. 2007; Weirich et al. 2017], so we provide only a short proof sketch in the next subsection. However, consistency (needed for the progress theorem) requires significant new structure due to the addition of equality reflection.

4.1 Structural Lemmas and Preservation

This system supports a computational weakening and substitution properties.

Lemma 4.1 (Weakening[16]). *If* $\Gamma \vdash a :^{\theta} A$ *and* $\vdash \Gamma, x :^{\delta_0} A_0$, *then* $\Gamma, x :^{\delta_0} A_0 \vdash a :^{\theta} A$.

The rule T-Var allows a variable at level L to be used in a P context. Therefore, the proof of substitution depends on the the subsumption property mentioned earlier (Lemma 3.3).

Lemma 4.2 (Substitution[17]). *If* $\Gamma, x :^{\delta_0} B \vdash a :^{\theta} A$ *and* $\Gamma \vdash b :^{\delta_0} B$, *then* $\Gamma \vdash a\{b/x\} :^{\theta} A\{b/x\}$.

Preservation for administrative reduction and computational reduction follow from substitution.

Lemma 4.3 (Preservation (Co)[18]). *If* $\Gamma \vdash a :^{\theta} A$ *and* $\Gamma \vdash^{\theta} a \rightharpoonup b$, *then* $\Gamma \vdash b :^{\theta} A$.

Lemma 4.4 (Preservation[19]). *If* $\Gamma \vdash a :^{\theta} A$ *and* $\Gamma \vdash^{\theta} a \leadsto b$, *then* $\Gamma \vdash b :^{\theta} A$.

4.2 Consistency

To prove progress (Theorem 4.22), we must first show the consistency of the equality judgment, which states that two definitionally equal terms under the empty context cannot have distinct head forms (Lemma 4.18). In pure type systems, where the equality judgment used in the conversion rule is beta-equivalence, the consistency of the equality judgment can be proven through syntactic means as a corollary of the Church-Rosser property; if two terms have conflicting head forms, then it is impossible for them to reduce to the same term because beta reduction preserves head forms. Choudhury et al. [2022] adapted the above argument to take into account compile-time

[15] sn_proof.v:ind_cong_intro [16] lp_weak.v:typing_weakening [17] lp_subst.v:typing_substc_nil
[18] preservation.v:preservation_cored [19] preservation.v:preservation_red

$\Omega \vdash a \Rightarrow b$	*Parallel reduction*	Figure 8	Appendix C.1
$\Omega \vdash^{\rho} a \Rightarrow b$	*Parallel reduction, relevance-moded*	Figure 8	Appendix C.2
$\Omega \vdash a \Rightarrow^{+} b$	*Multistep parallel reduction*	Definition 4.5	Appendix C.3
$\Omega \vdash a_1 \Leftrightarrow a_2$	*Joinability*	Definition 4.6	Appendix C.4
$\mathcal{V}[[A]]^{\theta}_{\xi}$	*Logical relation for covalues*	Figure 9	
$\mathcal{C}[[A]]^{\theta}_{\xi}$	*Logical relation for terms*	Figure 9	
$\vdash^{L} A$	*Logical types*	Figure 10	
$\xi \vDash \Gamma$	*Valuation well-formedness*	Figure 11	

Fig. 7. Summary of notation and judgement forms used in consistency proof

$\boxed{\Omega \vdash a \Rightarrow b}$ *(Parallel reduction (excerpt))*

$$\text{P-VAR}\quad \frac{x{:}R \in \Omega}{\Omega \vdash x \Rightarrow x} \qquad \text{P-TYPE}\quad \frac{}{\Omega \vdash \star \Rightarrow \star} \qquad \text{P-REIFY}\quad \frac{}{\Omega \vdash \mathbf{reify}^{\theta_0}\ \gamma_1 \Rightarrow \mathbf{reify}^{\theta_0}\ \gamma_2}$$

$$\text{P-APPCONG}\quad \frac{\Omega \vdash a_1 \Rightarrow a_2 \qquad \Omega \vdash^{\rho} b_1 \Rightarrow b_2}{\Omega \vdash a_1\ {b_1}^{\rho,\theta} \Rightarrow a_2\ {b_2}^{\rho,\theta}}$$

$$\text{P-APPABS}\quad \frac{\Omega \vdash a_1 \Rightarrow \lambda^{\rho,\theta} x{:}A.a_2 \qquad \Omega \vdash^{\rho} b_1 \Rightarrow b_2}{\Omega \vdash a_1\ {b_1}^{\rho,\theta} \Rightarrow a_2\{b_2/x\}} \qquad \text{P-CONV}\quad \frac{\Omega \vdash a_1 \Rightarrow a_2}{\Omega \vdash a_1 \rhd \gamma \Rightarrow a_2}$$

$\boxed{\Omega \vdash^{\rho} a \Rightarrow b}$ *(Parallel reduction, relevance-moded)*

$$\text{CP-LEQ}\quad \frac{\Omega \vdash a \Rightarrow b}{\Omega \vdash^{R} a \Rightarrow b} \qquad \text{CP-NLEQ}\quad \frac{}{\Omega \vdash^{I} a \Rightarrow b}$$

Fig. 8. Parallel reduction (selected rules)

irrelevance and inspired our consistency proof for System DE. However, due to rule E-REFLECT, the consistency proof of DE also requires the use of a logical relation because the consistency of the equality judgment relies on the weak normalization of the L fragment.

First, we introduce the definitions that are necessary to state the logical relation. These definitions are summarized in Figure 7. The mode context Ω keeps track of the relevance label associated with each variable. The joinability relation is defined in terms of the parallel reduction relation; two terms are joinable as long as there exists a common term that they both reduce to through the transitive closure of the parallel reduction relation (Figure 8).

Definition 4.5 (Multistep parallel reduction). The relation $\Omega \vdash a \Rightarrow^{+} b$ denotes the transitive closure of $\Omega \vdash a \Rightarrow b$.

Definition 4.6 (Joinable terms). Two terms are *joinable*, written $\Omega \vdash a_1 \Leftrightarrow a_2$, when there exists some b, such that $\Omega \vdash a_1 \Rightarrow^{+} b$ and $\Omega \vdash a_2 \Rightarrow^{+} b$.

One can see System DE's parallel reduction (Figure 8) as an ordinary parallel reduction relation extended with the concept of irrelevance and erasure of coercions. Baking all these concepts into

$$
\begin{aligned}
\mathcal{V}[[\mathbb{N}]]^L_\xi &= \{c \mid \cdot \vdash c :^L \mathbb{N}, \exists v, \cdot \vdash^L c \rightharpoonup^* v\} \\
\mathcal{V}[[\Pi x :^{\rho_0,\theta_0} A.B]]^L_\xi &= \left\{c \,\middle|\, \begin{array}{l} \cdot \vdash c :^L \Pi x :^{\rho_0,\theta_0} \xi(A).\xi(B), \exists a, \cdot \vdash^L c \rightharpoonup^* \lambda^{\rho_0,\theta_0} x : \xi(A).a, \\ \forall b \in \mathcal{C}[[A]]^{\theta_0}_\xi, a\{b/x\} \in \mathcal{C}[[B]]^L_{\xi, x\mapsto b} \end{array}\right\} \\
\mathcal{V}[[b_1 \equiv^{\theta_0} b_2]]^L_\xi &= \{c \mid \cdot \vdash c :^L \xi(b_1) \equiv^{\theta_0} \xi(b_2), \cdot \vdash \xi(b_1) \Leftrightarrow \xi(b_2)\} \\
\mathcal{V}[[\star]]^L_\xi &= \{a \mid \cdot \vdash a :^L \star\} \\
\mathcal{V}[[A]]^P_\xi &= \{a \mid \cdot \vdash a :^P \xi(A)\} \\
\mathcal{C}[[A]]^L_\xi &= \{a \mid \cdot \vdash a :^L \xi(A), \exists c \in \mathcal{V}[[A]]^L_\xi, \cdot \vdash^L a \rightsquigarrow^* c\} \\
\mathcal{C}[[A]]^P_\xi &= \{a \mid \cdot \vdash a :^P \xi(A)\}
\end{aligned}
$$

Fig. 9. Logical relation[21]

the same relation helps us simplify the metatheoretic proofs. Relevance-moded parallel reduction, in the same figure, is able to take shortcuts related to irrelevance, in a manner similar to the equality judgement. In rule P-Reify, because coercion proofs are not relevant, any proof γ_1 can step to any other proof γ_2. In rule P-AppCong, we delegate the reduction of the argument to the relevance-moded reduction relation. When the argument is annotated with I, we fall into rule CP-Nleq and are allowed to convert a term to an arbitrary term because an irrelevant argument should not affect the result of the evaluation. Consider the following term:

$$\lambda^{I,P} x : \star . x$$

The function marks its argument x as irrelevant, but it still uses x in its body by simply returning it. Allowing such functions in the reduction relation would break the confluence property of parallel reduction as seen in the following example:

$$(\lambda^{I,P} x : \star . x)\, \mathbb{N} \Rightarrow (\lambda^{I,P} x : \star . x)\, \star \Rightarrow \star$$
$$(\lambda^{I,P} x : \star . x)\, \mathbb{N} \Rightarrow \mathbb{N}$$

The rule P-Var correctly rejects the function since the irrelevant variable x is used in a relevant context, ruling out the malformed term from earlier.

The joinability relation satisfies the following consistency property. Below, the erased context $|\Gamma_0|$ is obtained from Γ_0 by removing θ and type annotations while keeping only the relevance mode associated with each variable.

Lemma 4.7 (Joinability consistency). [20] *If* $|\Gamma| \vdash a \Leftrightarrow b$*, then* a *and* b *cannot have conflicting head forms.*

When either a or b is not in head form, Lemma 4.7 is vacuously true. When a and b are both in head forms, the conclusion follows from the fact that parallel reduction preserves head forms.

Figure 9 shows the full definition of our logical relation. The logical relation takes the general form of $\mathcal{V}[[A]]^\theta_\xi$ and $\mathcal{C}[[A]]^\theta_\xi$. The former represents the covalue interpretation and the latter the term interpretation. The mode θ specifies the fragment of the type that we are interpreting. Because rule E-Reflect only allows reflection of logical terms, it is sufficient to include a catch-all case when $\theta = P$ so every syntactically well-typed P term trivially belongs to the set. The covalue interpretation is defined to include not just values, but also coerced values. In the function case, for example, we allow c to be a covalue that reduces to a lambda expression through the coercion

[20] par.v:Join_Consistent [21] sn_def.v:SN

$\boxed{\vdash^L A}$ *(Logical types)*

$$\frac{\vdash^L A_1 \qquad \vdash^L B_1}{\vdash^L \Pi x :^{\rho_0, L} A_1.B_1}\ \textsc{LTy-PiLogic} \qquad \frac{\vdash^L B_1}{\vdash^L \Pi x :^{\rho_0, P} A_1.B_1}\ \textsc{LTy-PiProg} \qquad \frac{}{\vdash^L a \equiv^\theta b}\ \textsc{LTy-Eq} \qquad \frac{}{\vdash^L \mathbb{N}}\ \textsc{LTy-Nat}$$

Fig. 10. Types free of computation and polymorphism

$\boxed{\xi \vDash \Gamma}$ *(Valuation well-formedness)*

$$\frac{}{\cdot \vDash \cdot}\ \textsc{VWff-Empty} \qquad \frac{\xi \vDash \Gamma \qquad a \in \mathcal{C}[\![A]\!]^{\theta_0}_{\xi}}{\xi, x \mapsto a \vDash \Gamma, x :^{\rho_0, \theta_0} A}\ \textsc{VWff-Cons}$$

Fig. 11. Well-formedness judgment for valuations

reduction relation. This extra step of administrative reduction is necessary so the logical relation matches our definition of our beta reduction rule as seen in rule R-AppAbs.

The logical relation admits the following formation properties. Every term in the logical relation must also be syntactically well-typed.

Lemma 4.8 (Logical Relation Typing). [22] *If $a \in \mathcal{C}[\![A]\!]^{\theta}_{\xi}$, then $\cdot \vdash a :^{\theta} \xi(A)$.*

Lemma 4.8 implies the following subsumption property about the logical relation, mirroring Lemma 3.3.

Lemma 4.9 (Logical Relation Subsumption). [23] *If $a \in \mathcal{C}[\![A]\!]^{L}_{\xi}$, then $a \in \mathcal{C}[\![A]\!]^{P}_{\xi}$.*

The definition of the logical relation is partial when θ is set to L. $\mathcal{V}[\![A]\!]^{L}_{\xi}$ and $\mathcal{C}[\![A]\!]^{L}_{\xi}$ are undefined when A is a variable or an application. The typing rules of System DE guarantees that such a scenario will never appear for well-formed types. The exclusion of the $\star : \star$ axiom forces variables to only appear behind equality types in an L type. For the convenience of specifying and proving certain lemmas about the logical relation, we define the predicate $\vdash^L A$ (see Figure 10) that characterizes logical types that are free of polymorphism and computation at the type level. We can then say that the logical relation is defined for a type A from the L fragment as long as $\vdash^L A$ holds.

The valuation ξ is a delayed substitution from variables to terms that may be applied with the notation $\xi(A)$. This mapping is needed in the definition of the logical relation to ensure that the definition is well-founded. The following lemma shows that eagerly performing the substitution over the codomain type in the function case gives us the exact same interpretation as our normal definition:

Lemma 4.10 (Substitution for logical relation[24]). *If $\vdash^L A$, then for every term a, $a \in \mathcal{C}[\![A]\!]^{L}_{\xi}$ if and only if $a \in \mathcal{C}[\![\xi(A)]\!]^{L}_{\cdot}$.*

The well-formedness judgment $\xi \vDash \Gamma$ for valuations can be found in Figure 11. Similar to the context well-formedness judgment, the well-formedness judgment for valuation satisfies the following narrowing property:

[22] sn_proof.v:SN_typing [23] sn_proof.v:SN_subsumption [24] sn_proof.v:SN_subst_valuation_iff

Lemma 4.11 (Narrowing (Valuation)[25]). *If $\xi \vDash \Gamma$, then $\xi \vDash \widetilde{\Gamma}$.*

The following lemma is the semantic counterpart of rule T-Conv:

Lemma 4.12 (Semantic conversion[26]). *If $\vdash^L A_1$, $\vdash^L A_2$, and $\xi \vDash \Gamma$, given a coercion proof $\cdot;\cdot \vdash^L \gamma : \xi(A_1) \sim \xi(A_2)$ and $\cdot \vdash \xi(A_1) \Leftrightarrow \xi(A_2)$, for every term $a \in C[\![A_1]\!]^L_\xi$, we must also have $a \rhd \gamma \in C[\![A_2]\!]^L_\xi$.*

Lemma 4.12 is useful not only for the rule T-Conv case of the fundamental theorem (see Theorem 4.15 below), but it is also useful for the rule T-Ind case. Recall that rule R-Ind always evaluates an induction form to a coerced term. Given $\Gamma \vdash^L a \rightsquigarrow^* v$, we cannot conclude that $\Gamma \vdash^\theta \mathbf{ind}_A\, a\, b_1\, (x.b_2) \rightsquigarrow^* \mathbf{ind}_A\, v\, b_1\, (x.b_2)$ because of the coercion proofs produced during the repeated application of the rule R-Ind. Instead, what we actually end up with is the following term:

$$\mathbf{ind}_A\, v\, b_1\, (x.b_2) \rhd \gamma_1 \rhd \gamma_2 \ldots \rhd \gamma_n$$

In the rule T-Ind case of the fundamental theorem, we need to repeatedly apply Lemma 4.12 to show that it suffices to show that $\mathbf{ind}_A\, v\, b_1\, (x.b_2)$ is in the interpreted set to derive that $\mathbf{ind}_A\, a\, b_1\, (x.b_2)$ is in the interpreted set.

Unlike the other lemmas, which can be proven through structural induction over derivations or terms, Lemma 4.12 requires natural number induction based on the following metric:

$$\kappa(\Pi x{:}^{\rho_0,P} A.B) = 1 + \kappa(B)$$
$$\kappa(\Pi x{:}^{\rho_0,L} A.B) = 1 + \kappa(A) + \kappa(B)$$
$$\kappa(A) = 1,\ \text{otherwise}$$

We cannot directly prove Lemma 4.12 by induction over types because the function type is contravariant in its argument type. The directions in which we cast the codomain and the domain of a function are opposite to each other. The termination metric allows us to strengthen the inductive hypothesis and cast from either side.

Before we state the fundamental theorems, we generalize the interpretation of types and equalities to include terms or coercion proofs that are open.

Definition 4.13. Semantic typing[27]

$$\Gamma \vDash a :^\theta A \iff \forall \xi, \xi \vDash \Gamma, \xi(a) \in C[\![A]\!]^\theta_\xi$$

Definition 4.14. Semantic equality[28]

$$\Gamma; \Gamma_0 \vDash a \sim^\theta b \iff \forall \xi, \xi \vDash \Gamma, |\Gamma_0| \vdash \xi(a) \Leftrightarrow \xi(b)$$

We prove the fundamental theorems for both typing and equality by mutual induction over the typing judgment and the equality judgment.

Theorem 4.15 (Fundamental theorem: typing[29]). *If $\Gamma \vdash a :^\theta A$, then $\Gamma \vDash a :^\theta A$.*

Theorem 4.16 (Fundamental theorem: equality[30]). *If $\Gamma; \Gamma_0 \vdash^\theta \gamma : a \sim b$, then $\Gamma; \Gamma_0 \vDash a \sim^\theta b$.*

In the interpretation of the equality judgment (Definition 4.14), the valuation ξ closes over the global context Γ but leaves the local context Γ_0 open. The distinct treatment of Γ and Γ_0 is necessary for Theorem 4.16 to hold. To see why, let us consider the following equality:

$$(\lambda^{\delta_0} x{:}A_1.a_1) \equiv^\theta (\lambda^{\delta_0} x{:}A_1.a_2)$$

[25] sn_proof.v:valuation_meet_ctx_l [26] sn_proof.v:conv_sn_iff [27] sn_def.v:SemTyping [28] sn_proof.v:SemDefEq
[29] sn_proof.v:typing_implies_semtyping [30] sn_proof.v:defeq_implies_semdefeq

To prove that this equality holds, we use the rule E-AbsCong to show that $a_1 \equiv^\theta a_2$ after we extend the context Γ_0 with x. If we used ξ to close over both Γ and Γ_0, the inductive hypothesis would become the following statement: for every closed term b of type A_1 that is in the logical relation, $a_1\{b_1/x\}$ and $a_2\{b_2/x\}$ are joinable. Our goal then is to show that the two lambda terms $\lambda^{\delta_0} x\!:\!A_1.a_1$ and $\lambda^{\delta_0} x\!:\!A_1.a_2$ are also joinable. In the case where δ_0 contains a logical label and A_1 is an uninhabited type such as $\mathbb{N} \equiv^P \star$, we end up with a vacuously true inductive hypothesis since it is impossible to construct a closed term of type A_1, assuming our system is consistent. This makes it impossible to finish the proof that the lambda terms are joinable because there is a chance that we do not know anything about the form of a_1 and a_2. On the other hand, if we limit the interpretation of the context to exclude Γ_0, our inductive hypothesis takes the form that a_1 and a_2 are joinable when x remains unsubstituted, allowing us to complete our proof.

By instantiating Γ and Γ_0 to the empty context, we derive the following corollary from Theorem 4.16.

Lemma 4.17 (DefEq Joinability). [31] *If* $\cdot;\cdot \vdash^\theta \gamma : a \sim b$*, then* $\cdot \vdash a \Leftrightarrow b$*.*

Finally, by composing Lemma 4.17 and Lemma 4.7, we prove the consistency of the equality judgment.

Lemma 4.18 (DefEq consistency[32]). *If* $\cdot;\cdot \vdash^\theta \gamma : a \sim b$*, then* a *and* b *cannot have conflicting head forms.*

Lemma 4.17 shows that two closed terms related by the typed equality judgment must also be related under the untyped joinability relation. One might wonder whether the other direction holds: if $\cdot \vdash a \Leftrightarrow b$ where a and b can be assigned the same type, is there always some γ that witnesses the equality between a and b? We have not proven this property for System DE because this direction is not needed for the type soundness proof. However, the relationship between a typed equality judgment and untyped beta-equivalence has been previously studied in Adams [2006] and Siles and Herbelin [2012]. The latter proves the equivalence between two pure type systems that use untyped beta-equivalence and typed judgemental equality in their respective conversion rules. The easy direction, showing that typed equality implies untyped equality, is related to Lemma 4.17. However, because our untyped equality is joinability instead of untyped beta-equivalence, our proof in this direction is both more difficult and more informative. Showing the reverse of this lemma would require extending Siles and Herbelin's proof technique to include equality reflection.

4.3 Progress

Recall that the computational reduction relation depends on the administrative reduction relation in its definition. As a result, we need to first show properties about the administrative reduction relation before we can prove progress for the computational reduction relation.

The following property says that we can reduce a term nested inside multiple layers of coercions into a term with at most one layer of coercion by repeatedly applying rule CR-Combine.

Lemma 4.19 (Progress-Semi (Co)[33]). *If* $\Gamma \vdash c :^\theta A$*, then either* c *is not a coerced term or there exists some* c_0 *and coercion proof* γ *such that* $\Gamma \vdash^\theta c \rightharpoonup^* c_0 \rhd \gamma$ *where* c_0 *is not a coerced term.*

When a well-typed application takes the form $c\,b$, the following property allows us to take a step through rule R-AppAbs.

Lemma 4.20 (Progress (Co-Abs)[34]). *If* $\cdot \vdash c :^\theta \Pi x\!:^{\delta_0} A.B$*, then there exists some lambda term* v *such that* $\cdot \vdash^\theta c \rightharpoonup^* v$

[31] sn_proof.v:defeq_join
[32] sn_proof.v:defeq_consist
[33] progress.v:covalue_semi_progress
[34] progress.v:covalue_progress_abs

The proof of Lemma 4.20 proceeds by case analysis on c. When c is not a coerced term, it must be in head form. The result then follows immediately by inverting the judgment $\cdot \vdash c :^{\theta} \Pi x :^{\delta_0} A.B$. Unlike dependent type theories with implicit conversion rules, we do not need a canonical form lemma for function types since rule T-Conv would require c to be a coerced term.

The complex case is when c takes the form of a coerced term. Lemma 4.19 allows us to take a few administrative reduction steps to reduce c into $c_0 \rhd \gamma$ for some c_0 and γ where c_0 is not a coerced term (and thus must be in head form). Recall that c is ascribed the type $\Pi x :^{\delta_0} A.B$. Since administrative reduction is typing-preserving (Lemma 4.3), $c_0 \rhd \gamma$ must also have type $\Pi x :^{\delta_0} A.B$. Therefore, γ must witness the equality between $\Pi x :^{\delta_0} A.B$ and the type ascribed to c_0. By Lemma 4.18 and the fact that c_0 is in head form, c_0 must be a lambda term since otherwise γ would witness an equality between the function type and a type with a distinct head form, contradicting the consistency lemma. Finally, we complete the proof of Lemma 4.20 by stepping $c_0 \rhd \gamma$ into a lambda term with rule CR-AbsPush.

With a similar argument, we prove the following fact about natural numbers.

Lemma 4.21 (Progress (Co-Nat)[35]). *If $\cdot \vdash c :^{\theta} \mathbb{N}$, then there exists some term c_0 of either the successor form or the zero form such that $\cdot \vdash^{\theta} c \rightharpoonup^{*} c_0$.*

From Lemma 4.20 and Lemma 4.21, we prove progress for the computational reduction relation by induction over the typing derivation.

Theorem 4.22 (Progress[36]). *If $\cdot \vdash a :^{\theta} A$, then either a is a covalue or there exists some b such that $\cdot \vdash^{\theta} a \rightsquigarrow b$.*

5 EXTENSIONS

More expressive logical sublanguage. Compared to the coercion proof language of GHC's core language, the logical sublanguage of System DE is significantly more expressive as it allows programmers to write inductive proofs. However, compared to a type theory like CIC [Coquand and Paulin 1990] or MLTT [Martin-Löf 1975], the L fragment lacks features such as polymorphism and type-level computations. However, extending the L fragment is just a matter of additional work; there is no fundamental limitation for such an extension. We believe that we would be able to adapt existing well-studied normalization proofs to this framework.

Strong existential types. System DE employs two relevance modes, R and I, with the latter denoting both runtime and compile-time irrelevance. This mechanism is inspired by the design of the DDC calculus [Choudhury et al. 2022]. As observed in DDC, identifying runtime with compile-time irrelevance is insufficient to encode strong existential types—a type where the first component is irrelevant at runtime, but required during type checking. To express this finer distinction, DDC is parameterized by a lattice of relevance levels that can be instantiated with the three levels required here, or more, for other forms of dependency analysis. For simplicity, we stick with two modes in System DE, but it would be straightforward to add more.

Strictness as a mode. As a model for non-strict computation, System DE includes only call-by-name functions. However, a pragmatic programming language, like GHC, also includes support for defining strict functions [Eisenberg and Peyton Jones 2017]. Strictness can be tricky in a dependently-typed language because one must also enforce value restrictions at the type-level [Casinghino et al. 2014]. However, this is another opportunity where modes can be used.

[35] progress.v:covalue_nat_progress [36] progress.v:progress

Extended L and P interactions. In System DE, the consistency of equality reflection is assured because terms from the P fragment can never determine the computation of the L fragment. However, there are conditions under which we would like to safely relax this invariant.

Consider the following definition of a list as the fixpoint of a positive functor [Swierstra 2008].

```
type List' a = fix (ListF a)

data ListF r a where
  NilF :: ListF r a
  ConsF :: a -> r -> ListF r a

fix :: forall a. (a -> a -> a) -> a -> a
fix f a = f a (fix f a)
```

A term of the `List'` type can never be marked as logical because the formation of the type relies on general recursion. However, we can recover an inductive list by bounding the size of `List'` with an irrelevant logical natural number.

```
-- a potentially nonterminating computation
sizeList' :: List' a -> Nat
sizeList' NilF = Z
sizeList' (ConsF a as) = S (sizeList' as)

-- a refinement of the List' type that only includes inductive lists
data InductiveList' a where
  IL' :: foreach (xs:: List' a) (n::^L Nat)
      -> forall (pf ::^L  sizeList' xs :=: n). InductiveList' a
```

We can already define `IndList'` in System DE because potentially diverging functions such as `sizeList'` can appear in equality types. What we would like to do is to show that operations defined over this type, such as `map` and `fold`, terminate.

However, defining such operations in the logical fragment means that we must be able to pattern match a programmatic datatype in a logical context. This sort of interaction between the L and P fragments can be done safely as long as there is evidence that the scrutinee has a normal form. In `IndList'` above, the equality `sizeList' xs :=: n` asserts that `xs` has a finite size and may only be derived when `xs` itself terminates.

More generally, we believe that once we add datatypes to System DE, we can extend our results with a new pattern matching construct that can destruct a P term within an L computation in the presence of appropriate evidence. This extension would be useful for extrinsic reasoning about P-only data types, such as those defined through recursion schemes.

Functional extensionality. System DE is an extensionally-flavored type theory. A proof term of equality from the L fragment can be directly reflected into the equality judgment through rule E-Reflect. Likewise, a coercion proof can be embedded into the term language through rule T-Reify when the local context Γ_0 in the judgment $\Gamma; \Gamma_0 \vdash^{\theta} \gamma : a \sim b$ is empty. In Section 3.3 and Section 4, we explain why the local context is important for us to derive Lemma 4.17. However, not only does this design complicate our typing rules, but it also limits the expressiveness of our language. Because of this restriction, functional extensionality is not a derivable property.

We suppress the expressiveness of the reflection rule not because the more powerful version is unsound, but because our existing proofs do not show its soundness. The issue is that our interpretation of equality as joinability does not identify enough terms. Therefore, instead of restricting rule E-Reflect to make Lemma 4.17 hold, an alternative might be to interpret the equality judgment as observational equality. However, this change would require significant modification to our arguments, so we leave it to future work.

6 RELATED WORK

6.1 Nontermination and Dependent Type Theory

There has been a long history of support for nonterminating computation in dependently typed languages. Dependent Type Theory with the $\star : \star$ axiom is inconsistent as a logic, but can be part of a sound type system for a programming language [Cardelli 1986]. Martin-Löf's partial type theory[Martin-Löf 1983] extended MLTT [Martin-Löf 1975] with a general fixpoint operator, making the type theory inconsistent as a programming logic as all types can inhabited an infinite loop. Palmgren [1993] generalized partial type theory with universe levels and assigned the extended system a domain-theoretic interpretation. Notably, Palmgren [1993] made the observation that the identity type in partial type theory cannot be made extensional since doing so would allow the definitional equality to convert between any two terms that have the same type. The design of DE shows that extensional equality can be soundly included in the language when the reflected equalities are restricted to a terminating fragment of the language.

The Nuprl system [Constable et al. 1986] allows the fixpoint combinator to appear in terms. However, to assign a type to a term, the programmer must provide a termination proof. Constable and Smith [1987] extended Nuprl with partial types, a mechanism that allows types to be inhabited by partial computations. To ensure the consistency of the type system, the typing rule of the fixpoint combinator requires the partial type from its input function to be admissible. System DE, on the other hand, extends a language that already supports partial computations with terminating and erasable proofs. Since we want our design to be applicable to Haskell, we do not want to impose any constraints that would make the programming fragment less expressive.

In proof assistants such as Agda and Coq, the Termination monad [Capretta 2005] and interaction trees [Xia et al. 2019] can be used to model nontermination in a terminating logic through the use of coinductive definitions [Martin-Löf 1988]. These frameworks model programs defined using general recursion, among other effects, and safely reason about their properties. Other approaches to general recursion include the use of a "fuel" argument and Bove and Capretta's inductive special-purpose accessibility predicates [Bove and Capretta 2005]. In contrast, System DE allows sound equational reasoning about programs in their native form, not interpreted monadically. Furthermore, while general recursion is a source of many forms of nontermination, divergence can also come about through other language features, such as non-strictly-positive datatypes or $\star : \star$, the type-in-type axiom.

The Zombie Trellys language [Casinghino 2014; Casinghino et al. 2014] also explored the use of modes and modal types to distinguish the logical and nonterminating parts of programs. We adopt the modalities L and P from that system. However, the programming fragment of of our system is more expressive than that of Zombie: our language admits type-level computation and type-in-type. The reason for this difference is that Zombie's consistency proof requires the definition of a step-indexed logical relation for the Zombie P fragment. Not only is this proof more complex than ours, since it defines separate relations for each of the L and P fragments, but it is not known how to extend this form of argument to languages with type-level computation and type-in-type.

Zombie's heavyweight proof technique stems from its permissive interactions between P fragment and the L fragment, similar to those described in Section 5. Because System DE lacks datatypes, these features are not (yet) supported by System DE. However, by identifying this trade-off, we enable this modal treatment of termination to be applied to an expressive language like Haskell.

Furthermore, we are confident that we can extend System DE with the most important of these interactions, as described above.

F* [Swamy et al. 2013] uses an effect system to identify potentially diverging programs. To reason about a diverging function, programmers may enrich its type signature with refinements that describe it abstractly, through pre- and post-conditions. It is, however, impossible to prove properties about diverging programs extrinsically since F* only allows intrinsically total terms to appear in specifications. Extrinsic reasoning is important to us since Haskell already has a large ecosystem. Being able to reason about potentially diverging programs allows us to incrementally verify an existing code base without introducing breaking changes to its interface.

Liquid Haskell [Vazou 2016], an SMT-based theorem prover for the Haskell programming language, supports refinement reflection [Vazou et al. 2017], a feature that allows a terminating subset of Haskell to be used in specifications. It is possible to verify a program either extrinsically through a lemma encoded as a terminating Haskell function, or intrinsically through pre- and post-conditions. Like F*, only terminating programs can be reflected and therefore functions not proved to be terminating when they are defined cannot be used for extrinsic reasoning. The subset of Haskell that can be reflected is also limited. To ensure decidable type checking, Liquid Haskell unfolds reflected top-level definitions before generating verification conditions that translate Haskell functions as uninterpreted SMT functions[37].

6.2 Irrelevance

Many dependently-typed languages include some form of relevance tracking. As described above, System DE's use of modes is inspired by DDC [Choudhury et al. 2022].

Liquid Haskell [Vazou 2016] supports a limited form of proof irrelevance. Properties in Liquid Haskell can be specified in the form of refinements. Similar to System DE, proofs do not need to be pattern matched before they can be used. It is possible to use the `withTheorem` proof combinator (similar to the `const` function in Haskell) to reflect a lemma so the SMT solver can use it. However, Liquid Haskell does not support specifying a function argument as irrelevant. Instead, it relies on the fact that Haskell is by default a call-by-name language to ensure that variables used only for reasoning do not induce overhead at runtime. The programmer has no way to statically ensure that a variable is indeed not needed at runtime and therefore cannot ignore irrelevant arguments during compile-time reasoning.

F* [Swamy et al. 2013], similar to Liquid Haskell, achieves compile-time proof irrelevance because the SMT solver does not construct evidence for properties that appear in refinements. In addition, F* supports runtime relevance by distinguishing between relevant and irrelevant *total* computations through the effects **`Tot`** and **`GTot`**. The result of the **`Tot`** effect can be used in **`GTot`** but typically not the other way round unless the usage does not affect the result of the **`Tot`** computation. Given a type `t`, one can construct its irrelevant counterpart `erased t`. Through a pair of functions `reveal` and `hide`, one can unbox and use a term of `erased t` in a **`Tot`** computation or box the result of a **`GTot`** computation so it can be passed to a **`Tot`** computation. Unlike System DE, F* cannot mark a potentially diverging but otherwise pure computation as irrelevant.

Quantitative type theory [Atkey 2018], used by both Idris [Brady 2021] and Agda [Abel and Bernardy 2020], tracks both linearity and relevance through usage annotations in the context. Unlike System DE, where terms and types are checked under the same judgment, quantitative type theory includes a special judgment for type well-formedness that completely ignores relevance information. As a result, quantitative type theory, when used as a framework for relevance tracking, can only track runtime irrelevance and cannot take advantage of irrelevance during compile-time reasoning.

[37] With Liquid Haskell's proof by logical evaluation feature, it is possible to use the SMT solver to guide the unfolding of reflected definitions.

Instead, Agda supports compile-time irrelevance through a separate, disjoint mechanism [Abel and Scherer 2012]. A function that takes a compile-time irrelevant argument cannot be applied to a runtime irrelevant argument, even though compile-time irrelevance enforces a strictly stronger irrelevance condition than runtime irrelevance.

6.3 Extensional vs. Intensional Type Theory

The propositional equality type in System DE is related to extensional type theory and its use of equality reflection [Martin-Löf 1975], and differs from that of intensional type theories, such as Agda and Coq. In these systems, casting with a propositional equality proof requires an explicit elimination form that pattern matches the equality proof. As a result, the proof is relevant in the computation. By requiring an explicit elimination form, intensional type theory ensures normalization in all contexts and supporting decidable type checking. In contrast, in the absence of explicit coercions as in System DE, type checking can diverge for extensional type theory. For Haskell, we are not overly concerned with this issue because type expressions can already diverge in the presence of flags such as `UndecidableInstances`.

6.4 Explicit Coercions

Type systems with explicit coercions allow nonterminating expressions to appear in types while retaining decidable type checking. For example, GHC's core language, System FC [Sulzmann et al. 2007], was designed to allow type families with undecidable instances, i.e. type-level computations that do not come with termination guarantees.

System DE is closely related to System DC, a dependently-typed variant of System FC, introduced by Weirich et al. [2017] and based on work by Gundry [2013] and Eisenberg [2016]. System DC includes a language of explicit coercions and coercion proofs, but does not allow equality reflection. Instead, it includes a way for types and terms to abstract over coercion proofs, which are themselves free of computation. By allowing terms of the propositional equality type to appear in coercions, we both increase the expressiveness of the coercion language and eliminate the need for coercion abstraction. However, this inclusion comes at a small cost: the consistency proof for DC is independent of the rest of the system and can be shown via induction over the derivation of the coercion judgment. In System DE, the distinction between the global and local context used by the coercion judgement is a generalization of a similar restriction found in System DC.

7 CONCLUSION

In this work, we present System DE, a dependently-typed language with relevance tracking, equality reflection, decidable type checking, and an expressive logical sublanguage that supports inductive proofs. Both relevance tracking and termination tracking are implemented through the unifying framework of modes.

In future work, in addition to the extensions we discuss in Section 5, we want to explore how to integrate these features into the GHC's type inferencer and core language and design a surface language that can be elaborated into a language with explicit coercions. We believe that the division between logical terms and coercion proofs that is present in System DE will manifest in the distinction between proofs that users must write versus those that can be inferred. That said, we also believe that the design of System DE is applicable not just to Haskell, but to future dependently-typed languages.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their comments and suggestions. This work was supported by the National Science Foundation under Grant No. 2006535.

A SYSTEM SPECIFICATION

This appendix contains the complete set of rules that specify the semantics of our language.

A.1 Coercion Proof Syntax

co, γ	$::=$		coercions
	$\mid$	$\mathbf{reflect}\ a$	reflected logical equality proof
	$\mid$	$\mathbf{reflex}\ a$	reflexivity
	$\mid$	$\mathbf{sym}\ \gamma$	symmetry
	$\mid$	$\gamma_1;\gamma_2$	transitivity
	$\mid$	$\Pi x :^{\delta_0} \gamma_1.\gamma_2$	congruence for dependent function type
	$\mid$	$\lambda^{\delta_0} x : A.\gamma_2$	congruence for functions
	$\mid$	$\gamma_1\ \gamma_2^{+} \rhd \gamma_3$	congruence for relevant function application
	$\mid$	$\gamma_1\ (a\ b)^{-} \rhd \gamma_2$	congruence for irrelevant function application
	$\mid$	$\mathbf{red}\ a\ b$	reduction
	$\mid$	$\mathbf{reify}^{\theta} \gamma_1 \gamma_2$	congruence for reify coercion
	$\mid$	$\mathbf{pifst}^{\theta}\ \gamma$	injectivity for function types
	$\mid$	$\gamma^{\theta_0} @ \gamma_1 \rhd \gamma_2$	injectivity for function types
	$\mid$	$\gamma \blacktriangleright \gamma_1\ \gamma_2$	congruence for conversion
	$\mid$	$\gamma_1 \sim^{\theta} \gamma_2$	congruence for equality
	$\mid$	$\mathbf{succ}\ \gamma$	congruence for successor
	$\mid$	$\mathbf{ind}_B\ \gamma_1\ \gamma_2\ (y.\gamma_3) \rhd \gamma_4$	congruence for induction

A.2 Typing

A.2.1 Typing.

$\boxed{\Gamma \vdash a :^{\theta} A}$ *(Typing)*

$$\textsc{T-Reify}\ \frac{\Gamma; \cdot \vdash^{\theta_0} \gamma : a \sim b}{\Gamma \vdash \mathbf{reify}^{\theta_0}\ \gamma :^{\theta} (a \equiv^{\theta_0} b)} \qquad \textsc{T-Conv}\ \frac{\Gamma \vdash a :^{\theta} A \qquad \widetilde{\Gamma}; \cdot \vdash^{\theta} \gamma : A \sim B}{\Gamma \vdash a \rhd \gamma :^{\theta} B} \qquad \textsc{T-Var}\ \frac{\theta_0 \leq \theta \qquad \vdash \Gamma \qquad x :^{R,\theta_0} A \in \Gamma}{\Gamma \vdash x :^{\theta} A}$$

$$\textsc{T-Pi}\ \frac{\Gamma \vdash A :^{\theta_0} \star \qquad \Gamma, x :^{R,\theta_0} A \vdash B :^{\theta} \star}{\Gamma \vdash \Pi x :^{\rho_0,\theta_0} A.B :^{\theta} \star} \qquad \textsc{T-Abs}\ \frac{\widetilde{\Gamma} \vdash \Pi x :^{\delta_0} A.B :^{\theta} \star \qquad \Gamma, x :^{\delta_0} A \vdash b :^{\theta} B}{\Gamma \vdash \lambda^{\delta_0} x : A.b :^{\theta} \Pi x :^{\delta_0} A.B} \qquad \textsc{T-App}\ \frac{\Gamma \vdash b :^{\theta} \Pi x :^{\delta_0} A.B \qquad \Gamma \vdash a :^{\delta_0} A}{\Gamma \vdash b\ a^{\delta_0} :^{\theta} B\{a/x\}}$$

$$\textsc{T-Eq}\quad \frac{\Gamma \vdash a :^{\theta_0} A \qquad \Gamma \vdash b :^{\theta_0} A}{\Gamma \vdash a \equiv^{\theta_0} b :^{\theta} \star} \qquad \textsc{T-TYPE}\quad \frac{\vdash \Gamma}{\Gamma \vdash \star :^{P} \star} \qquad \textsc{T-Zero}\quad \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{zero} :^{\theta} \mathbb{N}} \qquad \textsc{T-Succ}\quad \frac{\Gamma \vdash a :^{\theta} \mathbb{N}}{\Gamma \vdash \mathbf{succ}\, a :^{\theta} \mathbb{N}}$$

$$\textsc{T-Nat}\quad \frac{\vdash \Gamma}{\Gamma \vdash \mathbb{N} :^{\theta} \star} \qquad \textsc{T-Ind}\quad \frac{\begin{array}{c} \widetilde{\Gamma} \vdash \Pi x{:}^{R,L} \mathbb{N}.A :^{L} \star \\ \Gamma \vdash a_1 :^{L} \mathbb{N} \\ \Gamma \vdash a_2 :^{L} A\{\mathbf{zero}/x\} \\ \Gamma, y{:}^{R,L} \mathbb{N} \vdash a_3 :^{L} A\{y/x\} \overset{R,L}{\to} A\{\mathbf{succ}\, y/x\} \end{array}}{\Gamma \vdash \mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\; a_1\; a_2\; (y.a_3) :^{\theta} A\{a_1/x\}}$$

A.2.2 *Relevance-Moded Typing.*

$\boxed{\Gamma \vdash a :^{\delta} A}$ *(Relevance-moded typing)*

$$\textsc{CT-Leq}\quad \frac{\Gamma \vdash a :^{\theta} A}{\Gamma \vdash a :^{R,\theta} A} \qquad \textsc{CT-Top}\quad \frac{\widetilde{\Gamma} \vdash a :^{\theta} A}{\Gamma \vdash a :^{I,\theta} A}$$

A.2.3 *Context Well-Formedness.*

$\boxed{\vdash \Gamma}$ *(Context well-formedness)*

$$\textsc{T-Empty}\quad \frac{}{\vdash \cdot} \qquad \textsc{T-ConsTm}\quad \frac{\begin{array}{c} \vdash \Gamma \\ \widetilde{\Gamma} \vdash A :^{\theta} \star \qquad x \notin \operatorname{dom} \Gamma \end{array}}{\vdash \Gamma, x{:}^{\rho,\theta} A}$$

A.2.4 *Coercion Proofs.*

$\boxed{\Gamma; \Gamma_0 \vdash^{\theta} \gamma : a \sim b}$ *(Coercion proofs)*

$$\textsc{E-Reflect}\quad \frac{\begin{array}{c} \theta_0 \leq \theta \\ \vdash \Gamma, \Gamma_0 \qquad \Gamma \vdash a_0 :^{L} a \equiv^{\theta_0} b \\ \Gamma \vdash a :^{\theta} A \qquad \Gamma \vdash b :^{\theta} A \end{array}}{\Gamma; \Gamma_0 \vdash^{\theta} \mathbf{reflect}\, a_0 : a \sim b} \qquad \textsc{E-ConvCong}\quad \frac{\begin{array}{c} \Gamma; \Gamma_0 \vdash^{\theta} \gamma : a_1 \sim a_2 \\ \Gamma, \Gamma_0 \vdash a_1 \triangleright \gamma_1 :^{\theta} A \\ \Gamma, \Gamma_0 \vdash a_2 \triangleright \gamma_2 :^{\theta} A \end{array}}{\Gamma; \Gamma_0 \vdash^{\theta} \gamma \triangleright \gamma_1\, \gamma_2 : a_1 \triangleright \gamma_1 \sim a_2 \triangleright \gamma_2}$$

$$\textsc{E-EqCong}\quad \frac{\begin{array}{c} \Gamma; \Gamma_0 \vdash^{\theta_0} \gamma_1 : a_1 \sim a_2 \\ \Gamma; \Gamma_0 \vdash^{\theta_0} \gamma_2 : b_1 \sim b_2 \\ \Gamma, \Gamma_0 \vdash (a_1 \equiv^{\theta_0} b_1) :^{\theta} \star \end{array}}{\Gamma; \Gamma_0 \vdash^{\theta} \gamma_1 \sim^{\theta_0} \gamma_2 : (a_1 \equiv^{\theta_0} b_1) \sim (a_2 \equiv^{\theta_0} b_2)} \qquad \textsc{E-Reflex}\quad \frac{\Gamma, \Gamma_0 \vdash a :^{\theta} A}{\Gamma; \Gamma_0 \vdash^{\theta} \mathbf{reflex}\, a : a \sim a}$$

$$\textsc{E-ReifyCong}\quad \frac{\begin{array}{c} \vdash \Gamma, \Gamma_0 \\ \Gamma, \Gamma_0; \cdot \vdash^{\theta_0} \gamma_1 : a \sim b \\ \Gamma, \Gamma_0; \cdot \vdash^{\theta_0} \gamma_2 : a \sim b \end{array}}{\Gamma; \Gamma_0 \vdash^{\theta} \mathbf{reify}^{\theta_0} \gamma_1 \gamma_2 : (\mathbf{reify}^{\theta_0}\, \gamma_1) \sim (\mathbf{reify}^{\theta_0}\, \gamma_2)} \qquad \textsc{E-Red}\quad \frac{\Gamma, \Gamma_0 \vdash a >^{\theta} b}{\Gamma; \Gamma_0 \vdash^{\theta} \mathbf{red}\, a\, b : a \sim b}$$

E-Sym

$$\frac{\Gamma; \Gamma_0 \vdash^{\theta} \gamma : b \sim a}{\Gamma; \Gamma_0 \vdash^{\theta} \mathbf{sym}\, \gamma : a \sim b}$$

E-Trans

$$\frac{\begin{array}{c}\Gamma; \Gamma_0 \vdash^{\theta} \gamma_1 : a \sim a_1 \\ \Gamma; \Gamma_0 \vdash^{\theta} \gamma_2 : a_1 \sim b\end{array}}{\Gamma; \Gamma_0 \vdash^{\theta} \gamma_1 ; \gamma_2 : a \sim b}$$

E-PiCong

$$\frac{\begin{array}{c}\Gamma; \Gamma_0 \vdash^{\theta_0} \gamma_1 : A_1 \sim A_2 \\ \Gamma; \Gamma_0, x :^{R,\theta_0} A_1 \vdash^{\theta} \gamma_2 : B_1 \sim B_2 \\ \Gamma, \Gamma_0 \vdash \Pi x :^{\rho_0,\theta_0} A_1 . B_1 :^{\theta} \star \\ \Gamma, \Gamma_0 \vdash \Pi x :^{\rho_0,\theta_0} A_1 . B_2 :^{\theta} \star \\ B_3 = B_2\{x \triangleright \mathbf{sym}\, \gamma_1 / x\}\end{array}}{\Gamma; \Gamma_0 \vdash^{\theta} \Pi x :^{\rho_0,\theta_0} \gamma_1 . \gamma_2 : \Pi x :^{\rho_0,\theta_0} A_1 . B_1 \sim \Pi x :^{\rho_0,\theta_0} A_2 . B_3}$$

E-AbsCong

$$\frac{\begin{array}{c}\widetilde{\Gamma, \Gamma_0} \vdash A :^{\theta_0} \star \\ \Gamma; \Gamma_0, x :^{\rho_0,\theta_0} A \vdash^{\theta} \gamma_2 : a_1 \sim a_2 \\ \Gamma, \Gamma_0 \vdash (\lambda^{\rho_0,\theta_0} x : A . a_2) :^{\theta} B\end{array}}{\Gamma; \Gamma_0 \vdash^{\theta} \lambda^{\rho_0,\theta_0} x : A . \gamma_2 : \lambda^{\rho_0,\theta_0} x : A . a_1 \sim \lambda^{\rho_0,\theta_0} x : A . a_2}$$

E-AppCong

$$\frac{\begin{array}{c}\Gamma; \Gamma_0 \vdash^{\theta} \gamma_1 : a_1 \sim a_2 \\ \Gamma; \Gamma_0 \vdash^{\theta_0} \gamma_2 : b_1 \sim b_2 \\ \Gamma, \Gamma_0 \vdash a_1 \; {b_1}^{R,\theta_0} :^{\theta} A \\ \Gamma, \Gamma_0 \vdash a_2 \; {b_2}^{R,\theta_0} :^{\theta} B \\ \widetilde{\Gamma, \Gamma_0}; \cdot \vdash^{\theta} \gamma : A \sim B\end{array}}{\Gamma; \Gamma_0 \vdash^{\theta} \gamma_1 \; \gamma_2^{+} \triangleright \gamma : (a_1 \; {b_1}^{R,\theta_0}) \triangleright \gamma \sim a_2 \; {b_2}^{R,\theta_0}}$$

E-AppCongIrrel

$$\frac{\begin{array}{c}\Gamma; \Gamma_0 \vdash^{\theta} \gamma_1 : a_1 \sim a_2 \\ \widetilde{\Gamma, \Gamma_0} \vdash b_1 :^{\theta_0} A \\ \widetilde{\Gamma, \Gamma_0} \vdash b_2 :^{\theta_0} A \\ \Gamma, \Gamma_0 \vdash a_1 \; {b_1}^{I,\theta_0} :^{\theta} B_1 \\ \Gamma, \Gamma_0 \vdash a_2 \; {b_2}^{I,\theta_0} :^{\theta} B_2 \\ \widetilde{\Gamma, \Gamma_0}; \cdot \vdash^{\theta} \gamma : B_1 \sim B_2\end{array}}{\Gamma; \Gamma_0 \vdash^{\theta} \gamma_1 \; (b_1 \; b_2)^{-} \triangleright \gamma : (a_1 \; {b_1}^{I,\theta_0}) \triangleright \gamma \sim a_2 \; {b_2}^{I,\theta_0}}$$

E-PiFst

$$\frac{\begin{array}{c}\theta_0 \leq \theta_1 \\ \Gamma; \Gamma_0 \vdash^{\theta} \gamma : (\Pi x :^{\rho_0,\theta_0} A_1 . B_1) \sim (\Pi x :^{\rho_0,\theta_0} A_2 . B_2)\end{array}}{\Gamma; \Gamma_0 \vdash^{\theta_1} \mathbf{pifst}^{\theta} \gamma : A_1 \sim A_2}$$

E-PiSnd

$$\frac{\begin{array}{c}\theta_0 \leq \theta \\ \Gamma;\Gamma_0 \vdash^{\theta_0} \gamma : (\Pi x{:}^{\rho_1,\theta_1} A_1.B_1) \sim (\Pi x{:}^{\rho_1,\theta_1} A_2.B_2) \\ \Gamma;\Gamma_0 \vdash^{\theta_1} \gamma_1 : a_1 \sim a_2 \\ \Gamma,\Gamma_0 \vdash a_2 :^{\theta_1} A_2 \\ \Gamma,\Gamma_0;\cdot \vdash^{\theta_1} \gamma_2 : A_2 \sim A_1\end{array}}{\Gamma;\Gamma_0 \vdash^{\theta} \gamma^{\theta_0}@\gamma_1 \rhd \gamma_2 : B_1\{a_1 \rhd \gamma_2/x\} \sim B_2\{a_2/x\}}$$

E-SuccCong

$$\frac{\begin{array}{c}\Gamma;\Gamma_0 \vdash^{\theta} \gamma : a \sim b \\ \Gamma,\Gamma_0 \vdash a :^{\theta} \mathbb{N}\end{array}}{\Gamma;\Gamma_0 \vdash^{\theta} \mathbf{succ}\,\gamma : \mathbf{succ}\,a \sim \mathbf{succ}\,b}$$

E-IndCong

$$\frac{\begin{array}{c}\Gamma;\Gamma_0 \vdash^{L} \gamma_1 : a_1 \sim b_1 \\ \Gamma;\Gamma_0 \vdash^{L} \gamma_2 : a_2 \sim b_2 \\ \Gamma;\Gamma_0, y{:}^{R,L}\mathbb{N} \vdash^{L} \gamma_3 : a_3 \sim b_3 \\ \Gamma,\Gamma_0 \vdash \mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\; a_1\; a_2\; (y.a_3) :^{\theta} A_0 \\ \Gamma,\Gamma_0 \vdash \mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\; b_1\; b_2\; (y.b_3) :^{\theta} B_0 \\ \widetilde{\Gamma,\Gamma_0};\cdot \vdash^{\theta} \gamma : A_0 \sim B_0\end{array}}{\Gamma;\Gamma_0 \vdash^{\theta} \mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\; \gamma_1\; \gamma_2\; (y.\gamma_3) \rhd \gamma : (\mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\; a_1\; a_2\; (y.a_3)) \rhd \gamma \sim \mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\; b_1\; b_2\; (y.b_3)}$$

A.2.5 Primitive Reduction.

$\boxed{\Gamma \vdash a >^{\theta} b}$ *(Primitive Reduction)*

aBeta-ConvRefl

$$\frac{\begin{array}{c}\Gamma \vdash a \rhd \gamma :^{\theta} B \\ \widetilde{\Gamma};\cdot \vdash^{\theta} \gamma : A \sim A\end{array}}{\Gamma \vdash a \rhd \gamma >^{\theta} a}$$

aBeta-AppAbs

$$\frac{\Gamma \vdash (\lambda^{\delta_0} x{:}A.a)\; b^{\delta_0} :^{\theta} A_0}{\Gamma \vdash (\lambda^{\delta_0} x{:}A.a)\; b^{\delta_0} >^{\theta} a\{b/x\}}$$

aBeta-Combine

$$\frac{\Gamma \vdash (a \rhd \gamma_1) \rhd \gamma_2 :^{\theta} A}{\Gamma \vdash (a \rhd \gamma_1) \rhd \gamma_2 >^{\theta} a \rhd (\gamma_1;\gamma_2)}$$

aBeta-AbsPush

$$\frac{\begin{array}{c}\Gamma \vdash (\lambda^{\delta_0} x{:}A_1.a_1) \rhd \gamma :^{\theta} A \\ \theta_0 \leq \theta \\ \widetilde{\Gamma};\cdot \vdash^{\theta_0} \gamma : (\Pi x{:}^{\delta_0} A_1.B_1) \sim (\Pi x{:}^{\delta_0} A_2.B_2) \\ a_2 = a_1\{x \rhd \mathbf{sym}\,(\mathbf{pifst}^{\theta_0}\gamma)/x\} \\ \gamma_2 = \gamma^{\theta_0}@(\mathbf{reflex}\,x) \rhd (\mathbf{sym}\,(\mathbf{pifst}^{\theta_0}\gamma))\end{array}}{\Gamma \vdash (\lambda^{\delta_0} x{:}A_1.a_1) \rhd \gamma >^{\theta} \lambda^{\delta_0} x{:}A_2.(a_2 \rhd \gamma_2)}$$

aBeta-IndSucc

$$\frac{\Gamma \vdash \mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\; (\mathbf{succ}\,a_1)\; a_2\; (y.a_3) :^{\theta} B}{\Gamma \vdash \mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\; (\mathbf{succ}\,a_1)\; a_2\; (y.a_3) >^{\theta} (a_3\{a_1/y\})\; (\mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\; a_1\; a_2\; (y.a_3))^{R,L}}$$

aBeta-IndZero

$$\frac{\Gamma \vdash \mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\; \mathbf{zero}\; a_2\; (y.a_3) :^{\theta} B}{\Gamma \vdash \mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\; \mathbf{zero}\; a_2\; (y.a_3) >^{\theta} a_2}$$

A.3 Operational Semantics

A.3.1 Computational Reduction.

$\boxed{\Gamma \vdash^{\theta} a \rightsquigarrow b}$ *(Computational Reduction)*

$$\textsc{R-AppAbs}\quad \frac{a \textit{ is a covalue} \qquad \Gamma \vdash^{\theta} a \rightharpoonup^{*} \lambda^{\delta_0} x{:}A.a_1}{\Gamma \vdash^{\theta} a\ b^{\delta_0} \rightsquigarrow a_1\{b/x\}}$$

$$\textsc{R-App}\quad \frac{\Gamma \vdash^{\theta} a_1 \rightsquigarrow a_2}{\Gamma \vdash^{\theta} a_1\ b^{\delta_0} \rightsquigarrow a_2\ b^{\delta_0}}$$

$$\textsc{R-Conv}\quad \frac{\Gamma \vdash^{\theta} a_1 \rightsquigarrow a_2}{\Gamma \vdash^{\theta} a_1 \rhd \gamma \rightsquigarrow a_2 \rhd \gamma}$$

$$\textsc{R-Succ}\quad \frac{\Gamma \vdash^{\theta} a_1 \rightsquigarrow a_2}{\Gamma \vdash^{\theta} \mathbf{succ}\, a_1 \rightsquigarrow \mathbf{succ}\, a_2}$$

$$\textsc{R-IndSucc}\quad \frac{\begin{array}{c} a_0 \textit{ is a covalue} \\ \Gamma \vdash^{L} a_0 \rightharpoonup^{*} \mathbf{succ}\, a_1 \\ \Gamma \vdash \mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\ a_0\ a_2\ (y.a_3) :^{\theta} B_0 \\ \Gamma \vdash \mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\ (\mathbf{succ}\, a_1)\ a_2\ (y.a_3) :^{\theta} B_1 \\ \widetilde{\Gamma}; \cdot \vdash^{\theta} \gamma : B_1 \sim B_0 \end{array}}{\Gamma \vdash^{\theta} \mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\ a_0\ a_2\ (y.a_3) \rightsquigarrow (a_3\{a_1/y\}\ (\mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\ a_1\ a_2\ (y.a_3))^{R,L}) \rhd \gamma}$$

$$\textsc{R-IndZero}\quad \frac{\begin{array}{c} a_1 \textit{ is a covalue} \\ \Gamma \vdash^{L} a_1 \rightharpoonup^{*} \mathbf{zero} \\ \Gamma \vdash \mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\ a_1\ a_2\ (y.a_3) :^{\theta} B_0 \\ \Gamma \vdash \mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\ \mathbf{zero}\ a_2\ (y.a_3) :^{\theta} B_1 \\ \widetilde{\Gamma}; \cdot \vdash^{\theta} \gamma : B_1 \sim B_0 \end{array}}{\Gamma \vdash^{\theta} \mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\ a_1\ a_2\ (y.a_3) \rightsquigarrow a_2 \rhd \gamma}$$

$$\textsc{R-Ind}\quad \frac{\begin{array}{c} \Gamma \vdash^{L} a_1 \rightsquigarrow b_1 \\ \Gamma \vdash \mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\ a_1\ a_2\ (y.a_3) :^{\theta} B_0 \\ \Gamma \vdash \mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\ b_1\ a_2\ (y.a_3) :^{\theta} B_1 \\ \widetilde{\Gamma}; \cdot \vdash^{\theta} \gamma : B_1 \sim B_0 \end{array}}{\Gamma \vdash^{\theta} (\mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\ a_1\ a_2\ (y.a_3)) \rightsquigarrow (\mathbf{ind}_{(\Pi x^{R,L}\mathbb{N}.A)}\ b_1\ a_2\ (y.a_3)) \rhd \gamma}$$

A.3.2 Multiple Reduction.

$\boxed{\Gamma \vdash^{\theta} a \rightsquigarrow^{*} b}$ *(Multiple Reductions)*

$$\textsc{Rs-Refl}\quad \frac{}{\Gamma \vdash^{\theta} a \rightsquigarrow^{*} a}$$

$$\textsc{Rs-Step}\quad \frac{\Gamma \vdash^{\theta} a_1 \rightsquigarrow a_2 \qquad \Gamma \vdash^{\theta} a_2 \rightsquigarrow^{*} a_3}{\Gamma \vdash^{\theta} a_1 \rightsquigarrow^{*} a_3}$$

A.3.3 Administrative Reduction.

$\boxed{\Gamma \vdash^{\theta} a \rightharpoonup b}$ *(Administrative Reduction)*

$$\textsc{CR-ConvRefl}\ \frac{\widetilde{\Gamma}; \cdot \vdash^{\theta} \gamma : A \sim A}{\Gamma \vdash^{\theta} a \rhd \gamma \rightharpoonup a} \qquad \textsc{CR-Combine}\ \frac{}{\Gamma \vdash^{\theta} (a \rhd \gamma_1) \rhd \gamma_2 \rightharpoonup a \rhd (\gamma_1;\gamma_2)}$$

$$\textsc{CR-AbsPush}\ \frac{\begin{array}{c}\Gamma \vdash (\lambda^{\delta_0} x\!:\!A_1.a_1) \rhd \gamma :^{\theta} A \\ \theta_0 \leq \theta \\ \widetilde{\Gamma}; \cdot \vdash^{\theta_0} \gamma : (\Pi x\!:^{\delta_0} A_1.B_1) \sim (\Pi x\!:^{\delta_0} A_2.B_2) \\ a_2 = a_1\{x \rhd \mathbf{sym}\,(\mathbf{pifst}^{\theta_0} \gamma)/x\} \\ \gamma_2 = \gamma^{\theta_0} @ (\mathbf{reflex}\, x) \rhd (\mathbf{sym}\,(\mathbf{pifst}^{\theta_0} \gamma))\end{array}}{\Gamma \vdash^{\theta} (\lambda^{\delta_0} x\!:\!A_1.a_1) \rhd \gamma \rightharpoonup \lambda^{\delta_0} x\!:\!A_2.(a_2 \rhd \gamma_2)}$$

$$\textsc{CR-ConvCong}\ \frac{\Gamma \vdash^{\theta} a \rightharpoonup b}{\Gamma \vdash^{\theta} a \rhd \gamma \rightharpoonup b \rhd \gamma} \qquad \textsc{CR-Succ}\ \frac{\Gamma \vdash^{\theta} a_1 \rightharpoonup a_2}{\Gamma \vdash^{\theta} \mathbf{succ}\, a_1 \rightharpoonup \mathbf{succ}\, a_2}$$

A.3.4 Multiple Administrative Reductions.

$\boxed{\Gamma \vdash^{\theta} a \rightharpoonup^{*} b}$ *(Multiple Administrative Reductions)*

$$\textsc{CRs-Refl}\ \frac{}{\Gamma \vdash^{\theta} a \rightharpoonup^{*} a} \qquad \textsc{CRs-Step}\ \frac{\begin{array}{c}\Gamma \vdash^{\theta} a_1 \rightharpoonup a_2 \\ \Gamma \vdash^{\theta} a_2 \rightharpoonup^{*} a_3\end{array}}{\Gamma \vdash^{\theta} a_1 \rightharpoonup^{*} a_3}$$

B ERASURE

This appendix includes definitions needed to state our result about erasure, described in Section 3.4.

B.1 Erasure Operation

$$\begin{aligned}
|\mathbf{reify}^{\theta}\ \gamma| &= \Box \\
|\Pi x\!:^{\delta_0} A.B| &= \Box \\
|\lambda^{\delta_0} x\!:\!A.a| &= \lambda x.|a| \\
|a\ b^{R,\theta}| &= |a|\,|b| \\
|a\ b^{I,\theta}| &= |a|\,\Box \\
|\,\star\,| &= \Box \\
|x| &= x \\
|a \equiv^{\theta} b| &= \Box \\
|a \rhd \gamma| &= |a| \\
|\mathbb{N}| &= \Box \\
|\mathbf{succ}\ a| &= \mathbf{succ}\ |a| \\
|\mathbf{zero}| &= \mathbf{zero} \\
|\mathbf{ind}_A\ a\ b_1\ (x.b_2)| &= \mathbf{ind}\ |a|\ |b_1|\ (x.|b_2|)
\end{aligned}$$

B.2 Untyped Reduction

$\boxed{a \rightsquigarrow b}$ *(Untyped reduction)*

$$\textsc{ER-App}\ \frac{a_1 \rightsquigarrow a_2}{a_1\ b \rightsquigarrow a_2\ b} \qquad \textsc{ER-AppAbs}\ \frac{}{(\lambda x.a)\ b \rightsquigarrow a\{b/x\}} \qquad \textsc{ER-Succ}\ \frac{a_1 \rightsquigarrow a_2}{\mathbf{succ}\ a_1 \rightsquigarrow \mathbf{succ}\ a_2} \qquad \textsc{ER-IndZero}\ \frac{}{\mathbf{ind}\ \mathbf{zero}\ a_2\ (y.a_3) \rightsquigarrow a_2}$$

$$\textsc{ER-IndSucc}\ \frac{}{\mathbf{ind}\ (\mathbf{succ}\ a_1)\ a_2\ (y.a_3) \rightsquigarrow (a_3\{a_1/y\})\ (\mathbf{ind}\ a_1\ a_2\ (y.a_3))} \qquad \textsc{ER-Ind}\ \frac{a_1 \rightsquigarrow b_1}{\mathbf{ind}\ a_1\ a_2\ (y.a_3) \rightsquigarrow \mathbf{ind}\ b_1\ a_2\ (y.a_3)}$$

C PARALLEL REDUCTION

This appendix lists additional definitions necessary for our consistency proof, as described in Section 4.2.

C.1 Parallel Reduction (Untyped and Mode-Aware)

$\boxed{\Omega \vdash a \Rightarrow b}$ *(Parallel reduction)*

$$\textsc{P-Var}\ \frac{x{:}R \in \Omega}{\Omega \vdash x \Rightarrow x} \qquad \textsc{P-Type}\ \frac{}{\Omega \vdash \star \Rightarrow \star} \qquad \textsc{P-Reify}\ \frac{}{\Omega \vdash \mathbf{reify}^{\theta_0}\ \gamma_1 \Rightarrow \mathbf{reify}^{\theta_0}\ \gamma_2} \qquad \textsc{P-AppCong}\ \frac{\Omega \vdash a_1 \Rightarrow a_2 \quad \Omega \vdash^{\rho} b_1 \Rightarrow b_2}{\Omega \vdash a_1\ {b_1}^{\rho,\theta} \Rightarrow a_2\ {b_2}^{\rho,\theta}}$$

$$\textsc{P-AppAbs}\ \frac{\Omega \vdash a_1 \Rightarrow \lambda^{\rho,\theta} x{:}A.a_2 \quad \Omega \vdash^{\rho} b_1 \Rightarrow b_2}{\Omega \vdash a_1\ {b_1}^{\rho,\theta} \Rightarrow a_2\{b_2/x\}} \qquad \textsc{P-Conv}\ \frac{\Omega \vdash a_1 \Rightarrow a_2}{\Omega \vdash a_1 \rhd \gamma \Rightarrow a_2} \qquad \textsc{P-ConvCong}\ \frac{\Omega \vdash a_1 \Rightarrow a_2}{\Omega \vdash a_1 \rhd \gamma_1 \Rightarrow a_2 \rhd \gamma_2}$$

$$\textsc{P-Pi}\ \frac{\Omega \vdash A_1 \Rightarrow A_2 \quad \Omega, x{:}\,R \vdash B_1 \Rightarrow B_2}{\Omega \vdash \Pi x{:}^{\delta} A_1.B_1 \Rightarrow \Pi x{:}^{\delta} A_2.B_2} \qquad \textsc{P-Abs}\ \frac{\Omega, x{:}\,\rho \vdash b_1 \Rightarrow b_2}{\Omega \vdash \lambda^{\rho,\theta} x{:}A_1.b_1 \Rightarrow \lambda^{\rho,\theta} x{:}A_2.b_2}$$

$$\textsc{P-Eq}\ \frac{\Omega \vdash a_1 \Rightarrow a_2 \quad \Omega \vdash b_1 \Rightarrow b_2}{\Omega \vdash a_1 \equiv^{\theta} b_1 \Rightarrow a_2 \equiv^{\theta} b_2} \qquad \textsc{P-Zero}\ \frac{}{\Omega \vdash \mathbf{zero} \Rightarrow \mathbf{zero}} \qquad \textsc{P-Succ}\ \frac{\Omega \vdash a_1 \Rightarrow a_2}{\Omega \vdash \mathbf{succ}\ a_1 \Rightarrow \mathbf{succ}\ a_2} \qquad \textsc{P-Nat}\ \frac{}{\Omega \vdash \mathbb{N} \Rightarrow \mathbb{N}}$$

P-IndZero
$$\frac{\Omega \vdash a_1 \Rightarrow \mathbf{zero} \quad \Omega \vdash a_2 \Rightarrow b_2 \quad \Omega, y\colon R \vdash a_3 \Rightarrow b_3}{\Omega \vdash \mathbf{ind}_A\ a_1\ a_2\ (y.a_3) \Rightarrow b_2}$$

P-IndSucc
$$\frac{\Omega \vdash a_1 \Rightarrow \mathbf{succ}\ b_1 \quad \Omega \vdash a_2 \Rightarrow b_2 \quad \Omega, y\colon R \vdash a_3 \Rightarrow b_3}{\Omega \vdash \mathbf{ind}_A\ a_1\ a_2\ (y.a_3) \Rightarrow (b_3\{b_1/y\})\ (\mathbf{ind}_{A_0}\ b_1\ b_2\ (y.b_3))^{R,L}}$$

P-IndCong
$$\frac{\Omega \vdash a_1 \Rightarrow b_1 \quad \Omega \vdash a_2 \Rightarrow b_2 \quad \Omega, y\colon R \vdash a_3 \Rightarrow b_3}{\Omega \vdash \mathbf{ind}_A\ a_1\ a_2\ (y.a_3) \Rightarrow \mathbf{ind}_{A_0}\ b_1\ b_2\ (y.b_3)}$$

C.2 Parallel Reduction, Relevance-Moded

$\boxed{\Omega \vdash^{\rho} a \Rightarrow b}$ *(Parallel reduction, relevance-moded)*

CP-Leq
$$\frac{\Omega \vdash a \Rightarrow b}{\Omega \vdash^{R} a \Rightarrow b}$$

CP-Nleq
$$\frac{}{\Omega \vdash^{I} a \Rightarrow b}$$

C.3 Multistep Mode-Aware Parallel Reduction

$\boxed{\Omega \vdash a \Rightarrow^{+} b}$ *(Transitive closure of parallel reduction)*

MP-One
$$\frac{\Omega \vdash a \Rightarrow b}{\Omega \vdash a \Rightarrow^{+} b}$$

MP-Step
$$\frac{\Omega \vdash a \Rightarrow b \quad \Omega \vdash b \Rightarrow^{+} a'}{\Omega \vdash a \Rightarrow^{+} a'}$$

C.4 Joinability

$\boxed{\Omega \vdash a \Leftrightarrow b}$ *(Joinability)*

join
$$\frac{\Omega \vdash a_1 \Rightarrow^{+} b \quad \Omega \vdash a_2 \Rightarrow^{+} b}{\Omega \vdash a_1 \Leftrightarrow a_2}$$

REFERENCES

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

Andreas Abel and Gabriel Scherer. 2012. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science* 8, 1 (2012), 1:29. https://doi.org/10.2168/LMCS-8(1:29)2012

Robin Adams. 2006. Pure type systems with judgemental equality. *Journal of Functional Programming* 16, 2 (2006), 219–246.

Agda Development Team. 2023. *Agda.* https://wiki.portal.chalmers.se/agda/Main/HomePage

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

Bruno Barras and Bruno Bernardo. 2008. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. In *Foundations of Software Science and Computational Structures*, Roberto Amadio (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 365–379. https://doi.org/10.1007/978-3-540-78499-9_26

Ana Bove and Venanzio Capretta. 2005. Modelling general recursion in type theory. *Mathematical Structures in Computer Science* 15, 4 (2005), 671–708. https://doi.org/10.1017/S0960129505004822

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

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

Venanzio Capretta. 2005. General Recursion via Coinductive Types. *Logical Methods in Computer Science* Volume 1, Issue 2 (July 2005). https://doi.org/10.2168/LMCS-1(2:1)2005

Luca Cardelli. 1986. *A Polymorphic Lambda Calculus with Type:Type.* Technical Report 10. Digital Equipment Corporation, SRC. https://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-10.pdf

Chris Casinghino. 2014. *Combining Proofs and Programs.* Ph. D. Dissertation. University of Pennsylvania. https://repository.upenn.edu/handle/20.500.14332/28018

Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. 2014. Combining Proofs and Programs in a Dependently Typed Language. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 33–45. https://doi.org/10.1145/2535838.2535883

Pritam Choudhury, Harley Eades III, and Stephanie Weirich. 2022. A Dependent Dependency Calculus. In *Programming Languages and Systems, ESOP 2022 (Lecture Notes in Computer Science, Vol. 13240)*, Ilya Sergey (Ed.). Springer International Publishing, Cham, 403–430. https://doi.org/10.1007/978-3-030-99336-8_15

David Thrane Christiansen, Iavor S. Diatchki, Robert Dockins, Joe Hendrix, and Tristan Ravitch. 2019. Dependently Typed Haskell in Industry (Experience Report). *Proc. ACM Program. Lang.* 3, ICFP, Article 100 (jul 2019), 16 pages. https://doi.org/10.1145/3341704

R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. 1986. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Inc., USA.

Robert L Constable and Scott Fraser Smith. 1987. *Partial objects in constructive type theory.* Cornell University. Department of Computer Science.

Coq Development Team. 2019. *The Coq Proof Assistant.* https://doi.org/10.5281/zenodo.3476303

Thierry Coquand and Christine Paulin. 1990. Inductively defined types. In *COLOG-88*, Per Martin-Löf and Grigori Mints (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 50–66. https://doi.org/10.1007/3-540-52335-9_47

Richard Alan Eisenberg. 2016. *Dependent Types in Haskell: Theory and Practice.* Ph. D. Dissertation. University of Pennsylvania. https://repository.upenn.edu/handle/20.500.14332/29166

Richard A. Eisenberg and Simon Peyton Jones. 2017. Levity Polymorphism. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 525–539. https://doi.org/10.1145/3062341.3062357

GHC Development Team. 2023. *The Glasgow Haskell Compiler.* https://www.haskell.org/ghc/

Adam Gundry. 2013. *Type Inference, Haskell and Dependent Types.* Ph. D. Dissertation. University of Strathclyde. https://doi.org/10.48730/jt3g-ws74

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly Typed Heterogeneous Collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell* (Snowbird, Utah, USA) *(Haskell '04)*. Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/1017472.1017488

Yiyun Liu and Stephanie Weirich. 2023. *Artifact associated with Dependently-Typed Programming with Logical Equality Reflection.* https://doi.org/10.1145/3580401

Per Martin-Löf. 1975. An intuitionistic theory of types: predicative part. In *Logic Colloquium '73, Proceedings of the Logic Colloquium*, H.E. Rose and J.C. Shepherdson (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 80. North-Holland, 73–118. https://doi.org/10.1016/S0049-237X(08)71945-1

Per Martin-Löf. 1983. The domain interpretation of type theory. In *Workshop on the Semantics of Programming Languages, Abstracts and Notes. Programming Methodology Group, Chalmers Univ. Technology and Univ. Göteborg*.

Per Martin-Löf. 1988. Mathematics of infinity. In *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings (Lecture Notes in Computer Science, Vol. 417)*, Per Martin-Löf and Grigori Mints (Eds.). Springer, 146–197. https://doi.org/10.1007/3-540-52335-9_54

Nathan Mishra-Linger and Tim Sheard. 2008. Erasure and Polymorphism in Pure Type Systems. In *Foundations of Software Science and Computational Structures*, Roberto Amadio (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 350–364. https://doi.org/10.1007/978-3-540-78499-9_25

Erik Palmgren. 1993. An Information System Interpretation of Martin-Löf' s Partial Type Theory with Universes. *Information and Computation* 106, 1 (1993), 26–60. https://doi.org/10.1006/inco.1993.1048

F. Pfenning. 2001. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 221–230. https://doi.org/10.1109/LICS.2001.932499

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok StrniŠa. 2010. Ott: Effective Tool Support for the Working Semanticist. *J. Funct. Program.* 20, 1 (jan 2010), 71–122. https://doi.org/10.1017/S0956796809990293

Vincent Siles and Hugo Herbelin. 2012. Pure type system conversion is always typable. *Journal of Functional Programming* 22, 2 (2012), 153–180. https://doi.org/10.1017/S0956796812000044

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

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying Higher-Order Programs with the Dijkstra Monad. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 387–398. https://doi.org/10.1145/2491956.2491978

Wouter Swierstra. 2008. Data types à la carte. *Journal of Functional Programming* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

Niki Vazou. 2016. *Liquid Haskell: Haskell as a theorem prover.* University of California, San Diego. https://goto.ucsd.edu/~nvazou/thesis/main.pdf

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement Reflection: Complete Verification with SMT. *Proc. ACM Program. Lang.* 2, POPL, Article 53 (dec 2017), 31 pages. https://doi.org/10.1145/3158141

Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. 2013. System FC with Explicit Kind Equality. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 275–286. https://doi.org/10.1145/2500365.2500599

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

Robert Wright, Michel Steuwer, and Ohad Kammar. 2022. Idris2-Table: evaluating dependently-typed tables with the Brown Benchmark for Table Types (Extended Abstract). Talk based on library https://github.com/madman-bob/idris2-table.

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (dec 2019), 32 pages. https://doi.org/10.1145/3371119

Received 2023-03-01; accepted 2023-06-27