Eta-Equivalence in Core Dependent Haskell

Anastasiya Kravchuk-Kirilyuk
Princeton University, NJ, USA
ayk2@princeton.edu

Antoine Voizard
University of Pennsylvania, Philadelphia, PA, USA
voizard@seas.upenn.edu

Stephanie Weirich
University of Pennsylvania, Philadelphia, PA, USA
sweirich@cis.upenn.edu

Abstract

We extend the core semantics for Dependent Haskell with rules for η-equivalence. This semantics is defined by two related calculi, Systems D and DC. The first is a Curry-style dependently-typed language with nontermination, irrelevant arguments, and equality abstraction. The second, inspired by the Glasgow Haskell Compiler's core language FC, is the explicitly-typed analogue of System D, suitable for implementation in GHC. Our work builds on and extends the existing metatheory for these systems developed using the Coq proof assistant.

2012 ACM Subject Classification Software and its engineering → Functional languages; Software and its engineering → Polymorphism; Theory of computation → Type theory

Keywords and phrases Dependent types, Haskell, Irrelevance, Eta-equivalence

Digital Object Identifier 10.4230/LIPIcs.TYPES.2019.7

Supplementary Material `https://github.com/sweirich/corespec/tree/master`

Funding This material is based upon work supported by the National Science Foundation under Grant No. 1521539 and Grant No. 1704041. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

1 Introduction

In typed programming languages, the definition of type equality determines the expressiveness of the type system. If more types can (soundly) be shown to be equal, then more programs will type check. In dependently-typed languages, the definition of type equality relies on a definition of term equality, because terms may appear in types. Therefore, a dependently-typed language that can equate more terms can also admit more programs.

Many dependently-typed programming languages, such as Coq (since version 8.4) and Agda (from its initial design) include rules for η-equivalence when comparing functions for equality. These rules benefit programmers. For example, if a function f has type

$$f \;:\; P\ x \;\to \mathsf{Int}$$

then it can be called with an argument of type

$$P\ (\lambda y.\ x\ y)$$

because the term $(\lambda y.\ x\ x)$ is η-equivalent to x.

Dependent Haskell [20, 47] is a proposal to add dependent types to the Haskell programming language, as implemented by the Glasgow Haskell Compiler. This design unifies the term and type languages of Haskell so that terms may appear directly in types, removing the need for awkward singleton encodings of richly-typed data structures [21, 27, 45].

The specification of this language extension [47] is founded on two related dependently typed core calculi, called Systems D and DC. These two systems differ in their annotations: the latter language, which is inspired by and extends the FC intermediate language of GHC [42, 46], includes enough information to support simple, syntax-directed type checking. On the other hand, System D, is a Curry-style language meant to model the runtime behavior of the language, and to inspire type inference for the source language. (At the source level, type inference for Dependent Haskell will require more annotations than System D, which includes no annotations, and many fewer than System DC, which annotates everything.)

However, the specification of Systems D and DC, as presented in prior work, did not include rules for η-equivalence. The goal of this paper is to describe our experience with adding η-equivalence rules to these two systems, demonstrating that η-equivalence is compatible with Dependent Haskell.

While this extension is small—it involves three new rules for System D and two new rules for DC—it was not at all clear that it would work out from the beginning. Both Systems D and DC include support for *irrelevant arguments*, i.e the marking of some lambda-bound variables as not relevant for run-time execution. For Dependent Haskell, this feature is essential. Haskellers expect a type-erasure semantics and GHC erases type arguments during compilation. Irrelevance generalizes this idea to include not just type arguments but all terms that are used irrelevantly, enabling the generation of efficient code.

Unfortunately, η-equivalence, when combined with irrelevance in dependently-typed languages, is a subtle topic. Much prior work has laid out the issues, though in contexts that are not exactly the same as that found in Dependent Haskell. We describe this landscape in Section 6.3, and show how our work compares to and does not match any existing treatment of these features. In particular, our system features the type:type axiom, employs a typed definition of equivalence that ignores type annotations, supports large eliminations, includes a variant with decidable type checking, does not restrict how irrelevant arguments may be used in types, and comes with a completely mechanized type soundness proof.

In particular, this work extends the type soundness proof that was developed in prior work with support for η-equivalence. Prior work included a mechanized formalization of the meta-properties of both Systems D and DC, developed using the Coq proof assistant [43]. In this work, we have extended that development with these new rules and have updated the proofs accordingly. This mechanized proof gives us complete confidence in our extension, even in the face of a few curious findings.

As a result, this project also gives us a chance to report a success story for proof engineering. As the extension described in this paper is small compared to the overall system, we would expect the changes to the proof to be similarly minor, and they are. Furthermore, the three different forms of η-equivalence that we add are themselves quite similar to each other. Because of this relationship, a newcomer (the first author, an undergraduate at the time) could join the project and was able to adapt the changes needed for the usual η-equivalence rule to the novel ones for this setting. Although this process required careful understanding of binding representations, especially in the representation of the new rules, the mechanical proof served as an essential benefit to the overall research endeavor.

2 Overview of System D and System DC

This work presents and extends the languages Systems D and DC from prior work [47]. Therefore, we begin our discussion with an overview of these systems and their properties.

	D	DC
Typing	$\Gamma \vDash a : A$	$\Gamma \vdash a : A$
Definitional equality (terms)	$\Gamma; \Delta \vDash a \equiv b : A$	$\Gamma; \Delta \vdash \gamma : a \sim b$
Proposition well-formedness	$\Gamma \vDash \phi$ ok	$\Gamma \vdash \phi$ ok
Definitional equality (props)	$\Gamma; \Delta \vDash \phi_1 \equiv \phi_2$	$\Gamma; \Delta \vdash \gamma : \phi_1 \sim \phi_2$
Context well-formedness	$\vDash \Gamma$	$\vdash \Gamma$
Signature well-formedness	$\vDash \Sigma$	$\vdash \Sigma$
Primitive reduction	$\vDash a > b$	
One-step reduction	$\vDash a \leadsto b$	$\Gamma \vdash a \leadsto b$

Figure 1 Summary of judgement forms.

System D is an implicit language; its syntax only contains terms that are relevant for computation. It is based on a Curry-style variant of a dependently-typed lambda calculus, with the type:type axiom. Functions are not annotated with their domain types and computations may not terminate. As a result, type checking in System D is undecidable. Compared to other Curry-style languages [32, 33], this language annotates the locations of irrelevant abstractions and irrelevant applications. Such generalizations and instantiations may occur only at the marked locations. Full Curry-style languages allow generalization and instantiation at any point in the derivation.

In contrast, System DC is an explicit language. It extends System D with enough annotations so that type checking is not only decidable, it is straightforward through a simple syntax-directed algorithm. While System D is intended to serve as a *specification* of what Dependent Haskell should mean, System DC is intended to serve as a core *implementation* language for the Glasgow Haskell Compiler (GHC) [20, 22], when it is extended with dependent types. The annotations allow the compiler to check core language terms during compilation, eliminating potential sources of bugs during compilation.

Because the annotated language DC is, in some sense, a *reification* of the derivations of D; DC can thus be seen as a syntax-directed version of D. To emphasize this connection in our formal system, we reuse the same metavariables for analogous syntactic forms in both languages.[1] The judgement forms are summarized in Figure 1. By convention, judgements for D use a double turnstile ($\vDash$) whereas judgements for DC use a single turnstile ($\vdash$). As we make precise below, judgements in these two languages are connected: we can apply an erasure operation to DC derivations to produce analogous judgements in D, and given a derivation in D, it is possible to add enough annotations to produce an analogous judgement in DC.

The judgement forms in these languages include the usual typing judgement, a typed equivalence relation (augmented in DC with an explicit proof witness in γ), a first-class notion of equality propositions ϕ, and a judgement when two propositions are equivalent (also augmented with a proof witness in DC), as well as well-formedness checks for typing contexts Γ and top-level signatures of recursive definitions Σ.

Computation in both languages is specified operationally, using a small-step, call-by-name, evaluation relation $\leadsto$. These one-step relations are decidable and produce a unique reduct in each case. This computation is also type sound, which we demonstrate through preservation and progress theorems [49].

[1] In fact, our Coq development uses the same syntax for both languages and relies on the judgement forms to identify the pertinent sets of constructs.

System D

terms, types	a, b, A, B	$::=$	$\mathsf{type} \mid x \mid F \mid \lambda^{\rho} x.b \mid a\ b^{\rho} \mid \Box \mid \Pi^{\rho} x\!:\!A.B$
		$\mid$	$\Lambda c.a \mid a[\gamma] \mid \forall c\!:\!\phi.A$
coercions	γ	$::=$	$\bullet$
values	v	$::=$	$\lambda^{+} x.a \mid \lambda^{-} x.v \mid \Lambda c.a$
		$\mid$	$\mathsf{type} \mid \Pi^{\rho} x\!:\!A.B \mid \forall c\!:\!\phi.A$

System DC

terms, types	a, b, A, B	$::=$	$\mathsf{type} \mid x \mid F \mid \lambda^{\rho} x\!:\!A.b \mid a\ b^{\rho} \mid \Pi^{\rho} x\!:\!A.B$
		$\mid$	$\Lambda c\!:\!\phi.a \mid a[\gamma] \mid \forall c\!:\!\phi.A$
		$\mid$	$a \rhd \gamma$
coercions (excerpt)	γ	$::=$	$c \mid \mathbf{refl}\ a \mid \mathbf{sym}\ \gamma \mid \gamma_1 ; \gamma_2 \mid \mathbf{red}\ a\ b \mid \ \ldots$
			$\mathbf{eta}\ a$
values	v	$::=$	$\lambda^{+} x\!:\!A.a \mid \lambda^{-} x\!:\!A.v \mid \Lambda c\!:\!\phi.a$
		$\mid$	$\mathsf{type} \mid \Pi^{\rho} x\!:\!A.B \mid \forall c\!:\!\phi.A$

Shared syntax

propositions	ϕ	$::=$	$a \sim_A b$
relevance	ρ	$::=$	$+ \mid -$
contexts	Γ	$::=$	$\varnothing \mid \Gamma, x : A \mid \Gamma, c : \phi$
available set	Δ	$::=$	$\varnothing \mid \Delta, c$
signature	Σ	$::=$	$\varnothing \mid \Sigma \cup \{F \sim a : A\}$

Figure 2 Syntax of D and DC. The syntactic differences between the two systems are highlighted in yellow. The sole addition for η-equivalence (the coercion form **eta** a) is highlighted in green.

The syntax of D, the implicit language, is shown at the top of Figure 2. This language, inspired by pure type systems [12], uses a shared syntax for terms and types. The language includes:

- a single sort (type) for classifying types,
- functions ($\lambda^{+}x.a$) with dependent types ($\Pi^{+}x\!:\!A.B$), and their associated application form ($a\ b^{+}$),
- functions with irrelevant arguments ($\lambda^{-}x.a$), their types ($\Pi^{-}x\!:\!A.B$), and instantiation form ($a\ \Box^{-}$),
- coercion abstractions ($\Lambda c.a$), their types ($\forall c\!:\!\phi.B$), and instantiation form ($a[\bullet]$),
- and top-level recursive definitions (F).

In this syntax, term and type variables, x, are bound in the bodies of functions and their types. Similarly, coercion variables, c, are bound in the bodies of coercion abstractions and their types. (Technically, irrelevant variables and coercion variables are prevented by the typing rules from actually appearing in the bodies of their respective abstractions.) We use the same syntax for relevant and irrelevant functions, marking which one we mean with a relevance annotation ρ. We sometimes omit relevance annotations ρ from applications $a\ b^\rho$ when they are clear from context. We also write nondependent relevant function types $\Pi^+x\!:\!A.B$ as $A \to B$, when x does not appear free in B, and write nondependent coercion abstraction types $\forall c\!:\!\phi.A$ as $\phi \Rightarrow A$, when c does not appear free in A.

The metavariable Δ, called the *available set*, represents a set of coercion variables. This set is used to restrict the usage of coercion variables in certain situations; only variables appearing in the set are available.[2] The operation $\widetilde{\Gamma}$ returns the available set made of all the coercion variables in the domain of context Γ. In other words, it is the available set that permits the use of all coercion variables in Γ.

The syntax of DC, also shown in the figure, includes the same features as D but with more typing annotations. In particular, this language removes the trivial argument for irrelevant instantiation (instead specifying the actual argument it stands for) and adds domain information to the bound variable in the abstraction forms. Finally, it replaces implicit type conversions by an explicit coercion term $a \triangleright \gamma$ as well as a language of coercion proofs (not completely shown in the figure). The addition of η-equivalence requires a new form of coercion proof, written **eta** a, that corresponds to all three new equivalence rules in D.

The erasure operation, written $|a|$ translates terms from System DC to System D by removing all type annotations and coercion proofs. For example, rules of this function include $|\lambda^\rho x\!:\!A.a| = \lambda^\rho x.|a|$ and $|a \triangleright \gamma| = |a|$.

2.1 Type checking in System D and System DC

Unlike System D, System DC enjoys unique typing, meaning that any given term has at most one type. Thanks to this uniqueness property and to the presence of typing annotations, type checking is decidable in System DC. In fact, the syntax of System DC can be seen as encoding not just a D term, but a D *typing derivation*. That is, any DC term uniquely identifies a typing derivation for the underlying (erased) D term.

In System D, type checking is undecidable due to two reasons. The first is that System D includes Curry-style System F as a sublanguage, where type checking is known to be undecidable [48, 36]. Since type arguments are implicit in Curry-style languages, irrelevant quantification is a feature of System D. The second reason for undecidable type checking in System D is the presence of an implicit conversion rule. In order to maintain decidable type checking in an environment where implicit conversion is allowed, System DC uses explicit coercion proofs whenever type conversion is performed. Below, we discuss these two features which contribute to the undecidability of type checking in System D. However, even though type checking is undecidable, we sketch what a partial type inference algorithm for System D might look like in Section 2.3.

2.1.1 Irrelevant quantification

Because Haskell includes parametric polymorphism, which has a type erasure semantics, Dependent Haskell includes a way to indicate which terms should be erased before execution.[3] Thus, the rules that govern the treatment of irrelevant, or implicit, quantification appear in Figure 3.

[2] This is analogous to marking available coercion variables in the context.
[3] Although it is possible to infer such information [14], we annotate it here to avoid a reliance on whole program optimization.

$$\textsc{E-Pi}\ \frac{\Gamma, x : A \vDash B : \mathsf{type}}{\Gamma \vDash \Pi^{\rho} x\!:\!A.B : \mathsf{type}} \qquad \textsc{An-Pi}\ \frac{\Gamma, x : A \vdash B : \mathsf{type}}{\Gamma \vdash \Pi^{\rho} x\!:\!A.B : \mathsf{type}}$$

$$\textsc{E-Abs}\ \frac{\Gamma, x : A \vDash a : B \qquad (\rho = +) \vee (x \notin \mathsf{fv}\ a)}{\Gamma \vDash \lambda^{\rho} x.a : \Pi^{\rho} x\!:\!A.B} \qquad \textsc{An-Abs}\ \frac{\Gamma, x : A \vdash a : B \qquad (\rho = +) \vee (x \notin \mathsf{fv}\ |a|)}{\Gamma \vdash \lambda^{\rho} x\!:\!A.a : \Pi^{\rho} x\!:\!A.B}$$

$$\textsc{E-App}\ \frac{\Gamma \vDash b : \Pi^{+} x\!:\!A.B \qquad \Gamma \vDash a : A}{\Gamma \vDash b\ a^{+} : B\{a/x\}} \qquad \textsc{An-App}\ \frac{\Gamma \vdash b : \Pi^{\rho} x\!:\!A.B \qquad \Gamma \vdash a : A}{\Gamma \vdash b\ a^{\rho} : B\{a/x\}}$$

$$\textsc{E-IApp}\ \frac{\Gamma \vDash b : \Pi^{-} x\!:\!A.B \qquad \Gamma \vDash a : A}{\Gamma \vDash b\ \Box^{-} : B\{a/x\}}$$

Figure 3 Rules for relevant and irrelevant arguments in System D (left) and System DC (right).

D and DC's approach to implicit quantification follows ICC [32], ICC* [13], and EPTS [33]. When possible, the typing rules use the metavariable ρ to generalize over the relevance of the abstraction. For example, irrelevance places no restrictions on the usage of the bound variable in the body of the dependent function type, so the same rule suffices in each case (see rules E-Pi and An-Pi).

However, for abstractions, if the argument is irrelevant, then the variable cannot appear in the body of the System D term (rule E-Abs). On the other hand, System DC includes annotations, which are not relevant, so the DC rule only restricts the variable from appearing in the *erasure* of the body (rule An-Abs).

In DC, an application term is type-checked in the same way no matter whether it is relevant or not, so we are able to use the same rule in both cases (rule An-App). However, in D, if the application is to an irrelevant argument, then the argument does not appear in the term. Instead, it must be replaced by the trivial term $\Box$ (rule E-IApp). Type-checking an irrelevant application in D thus requires guessing the actual argument used at this occurrence. Due to this, we need two separate rules for relevant and irrelevant application in D (rule E-App and rule E-IApp respectively).

2.1.2 Explicit coercions

As mentioned previously, System D includes an implicit conversion rule, shown on the left below (rule E-Conv). This rule depends on the type equality judgement to allow the system to work up-to the definition of this type equality. At any point in a System D derivation, the type of a term can silently be replaced with an equivalent type.

$$\textsc{E-Conv}\ \frac{\Gamma \vDash a : A \qquad \Gamma; \widetilde{\Gamma} \vDash A \equiv B : \mathsf{type}}{\Gamma \vDash a : B} \qquad \textsc{An-Conv}\ \frac{\Gamma \vdash a : A \qquad \Gamma; \widetilde{\Gamma} \vdash \gamma : A \sim B \qquad \Gamma \vdash B : \mathsf{type}}{\Gamma \vdash a \rhd \gamma : B}$$

To enable decidable type checking, System DC includes an explicit justification γ in rule An-Conv, called a coercion proof, whenever type conversion is used. These coercions are reifications of the type equality derivations of System D; a coercion proof γ specifies a unique equality derivation. Equality is homogeneously typed in System D, if we have $\Gamma; \Delta \vDash a \equiv b : A$, then both terms a and b must have type A. In DC the relationship is more nuanced. If we have a coercion proof $\Gamma; \Delta \vdash \gamma : a \sim b$ where $\Gamma \vdash a : A$ and $\Gamma \vdash b : B$, then there must exist an additional coercion proof witnessing the equality between types A and B. In other words, the types of coercible terms must be equal according to System D. For example, compare the reflexivity rule in System D below (rule E-Refl) with the two different reflexivity rules in System DC (rule An-Refl and rule An-EraseEq). While the first DC rule is the classic form of the reflexivity rule, we still need the second form to account for the case when two terms a and b have different type annotations. To derive reflexivity between a and b in this case, we must furthermore know that their *types* are equal, witnessed by the coercion proof γ. Note also that we cannot get away with having rule An-EraseEq alone, since rule An-Refl is the only rule which can derive reflexivity for type. For example, in order to prove $\mathbf{Int} \sim_{\mathsf{type}} \mathbf{Int}$ with rule An-EraseEq, we need the base case rule An-Refl to prove $\mathsf{type} \sim_{\mathsf{type}} \mathsf{type}$.

$$\textsc{E-Refl}\quad \frac{\Gamma \vDash a : A}{\Gamma; \Delta \vDash a \equiv a : A} \qquad \textsc{An-Refl}\quad \frac{\Gamma \vdash a : A}{\Gamma; \Delta \vdash \mathbf{refl}\ a : a \sim a} \qquad \textsc{An-EraseEq}\quad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B \qquad |a| = |b| \qquad \Gamma; \widetilde{\Gamma} \vdash \gamma : A \sim B}{\Gamma; \Delta \vdash (a \mid=\mid_{\gamma} b) : a \sim b}$$

The type equality judgement in System D includes primitive (i.e. β) reductions, shown in rule E-Beta below. The analogous rule in System DC uses an explicit coercion, $\mathbf{red}\ a_1\ a_2$ in the coercion checking rule An-Beta to indicate a reduction. Both rules use the primitive reduction relation of System D, available in DC through erasure. Although this relation is deterministic, there are multiple ways to annotate a System D term. Thus, the coercion rule must annotate both terms, a_1 and a_2 involved in the redex. Furthermore, because these annotations may differ, these terms may have different types in DC, as long as those types are also related through erasure.

$$\textsc{E-Beta}\quad \frac{\Gamma \vDash a_1 : B \qquad \vDash a_1 > a_2}{\Gamma; \Delta \vDash a_1 \equiv a_2 : B} \qquad \textsc{An-Beta}\quad \frac{\Gamma \vdash a_1 : B_0 \qquad \Gamma \vdash a_2 : B_1 \qquad |B_0| = |B_1| \qquad \vDash |a_1| > |a_2|}{\Gamma; \Delta \vdash \mathbf{red}\ a_1\ a_2 : a_1 \sim a_2}$$

The System D type equality judgement is undecidable because it includes the operational semantics and the language is nonterminating. This nontermination is due to the type:type axiom and general recursion, the latter already available in Haskell. Furthermore, because System D is nonterminating, types themselves may diverge and thus don't necessarily have normal forms (this is already the case for GHC, in the presence of certain language extensions).

2.2 Coercion abstraction

D and DC inherit the *coercion abstraction* feature from System FC, the existing core language of GHC [42, 46]. This feature is primarily used to implement GADTs in GHC but is also available for explicit use by Haskell programmers.

Coercion abstraction means that equality is first class. Terms may abstract over equality propositions (denoted by ϕ in rules E-CAbs and An-CAbs) and can discharge those assumptions in contexts where the proposition is derivable (rules E-CApp and An-CApp). Once an equality has been assumed in the context, it may contribute to an equivalence derivation as long as the coercion variable is available (i.e. found in the available set Δ).

$$\textsc{E-CAbs}\quad \frac{\Gamma, c : \phi \vDash a : B}{\Gamma \vDash \Lambda c.a : \forall c{:}\phi.B} \qquad \textsc{An-CAbs}\quad \frac{\Gamma, c : \phi \vdash a : B}{\Gamma \vdash \Lambda c{:}\phi.a : \forall c{:}\phi.B}$$

$$\textsc{E-CApp}\quad \frac{\Gamma \vDash a_1 : \forall c{:}(a \sim_A b).B_1 \qquad \Gamma; \widetilde{\Gamma} \vDash a \equiv b : A}{\Gamma \vDash a_1[\bullet] : B_1\{\bullet/c\}} \qquad \textsc{An-CApp}\quad \frac{\Gamma \vdash a_1 : \forall c{:}a \sim_{A_1} b.B \qquad \Gamma; \widetilde{\Gamma} \vdash \gamma : a \sim b}{\Gamma \vdash a_1[\gamma] : B\{\gamma/c\}}$$

$$\textsc{E-Assn}\quad \frac{\vDash \Gamma \qquad c : (a \sim_A b) \in \Gamma \qquad c \in \Delta}{\Gamma; \Delta \vDash a \equiv b : A} \qquad \textsc{An-Assn}\quad \frac{\vdash \Gamma \qquad c : a \sim_A b \in \Gamma \qquad c \in \Delta}{\Gamma; \Delta \vdash c : a \sim b}$$

The role of the set Δ is to prevent the usage of certain coercion variables, namely those introduced in a congruence proof between two coercion abstraction types. More details about this issue are available in prior work [47].

2.3 Type inference for System D

Even though complete type inference for System D is undecidable, we still intend it to be a model for the source language of the Glasgow Haskell Compiler. Type inference in GHC currently elaborates implicitly-typed Source Haskell to an explicitly-typed core language, similar to System DC. This inference algorithm works by gathering constraints and then solving those constraints using a variant of mixed-prefix unification combined with type-family reduction [44]. This algorithm already supports numerous features related to System D, including GADTs, type-level computation, higher-rank polymorphism and the type:type axiom. There are also experimental extensions of this algorithm in support of type-level lambdas [26], higher-kinds [50], and first-class polymorphism [39]. The most straightforward extension of GHC's algorithm with dependent types is based on parallel reduction; to determine whether two types are equivalent one must find a term that they both reduce to. In System D, this reduction may not terminate, so this process describes a semi-decision procedure.

3 Adding η-equivalence to Systems D and DC

Extending Systems D and DC with η-equivalence requires the addition of the following three rules to System D and two analogous rules in System DC. These three rules encode the usual η-equivalence properties for normal functions, irrelevant functions, and coercion abstractions. As our equivalence relation is typed, we must ensure that both left and right hand sides are well typed with the same type. This precondition also ensures that the bound variable does not appear free in b.

$$\textsc{E-EtaRel}\quad \frac{\Gamma \vDash b : \Pi^+ x{:}A.B}{\Gamma; \Delta \vDash \lambda^+ x.b\ x^+ \equiv b : \Pi^+ x{:}A.B} \qquad \textsc{E-EtaIrrel}\quad \frac{\Gamma \vDash b : \Pi^- x{:}A.B}{\Gamma; \Delta \vDash \lambda^- x.b\ \Box^- \equiv b : \Pi^- x{:}A.B}$$

$$\textsc{E-EtaC}\quad \frac{\Gamma \vDash b : \forall c{:}\phi.B}{\Gamma; \Delta \vDash \Lambda c.b[\bullet] \equiv b : \forall c{:}\phi.B}$$

In the annotated language, we only need two rules for coercion proofs because we can unify the two application forms in the annotated language (i.e. we can generalize over ρ).

$$\textsc{An-Eta}\quad \frac{\Gamma \vdash b : \Pi^{\rho} x\!:\!A.B}{\Gamma; \Delta \vdash \mathbf{eta}\, b : \lambda^{\rho} x.b\ x^{\rho} \sim b} \qquad \textsc{An-EtaC}\quad \frac{\Gamma \vdash b : \forall c\!:\!\phi.B}{\Gamma; \Delta \vdash \mathbf{eta}\, b : \Lambda c.b[c] \sim b}$$

We use the single marker **eta** b as the explicit proof witness for both rules. We can overload this form because the annotated term b includes enough information to recover its type, and the type of b is enough to determine which of the η-equivalence properties are needed.

The five rules shown in this section are all that was needed to extend the definition of both languages with η-equivalence. Note that we do not include any η rules (i.e. reduction or expansion) in the operational semantics (i.e. the one step reduction relations $\vDash a \rightsquigarrow b$ and $\Gamma \vdash a \rightsquigarrow b$). The computational behavior of the system is unchanged by this extension. Instead, our goal is to extend the systems' reasoning about this existing computational behavior through the added equivalences. Although the rules for η-equivalence for relevant and irrelevant function have appeared in various prior work (see Section 6), the η-equivalence rule for coercion abstraction is new to this extension.

4 Extending proofs

The addition of the five rules above means that we must extend all existing proofs of Systems D and DC and show that after the inclusion of the new rules these systems retain the desired properties. The properties developed in prior work [47] include the following results.

- Consistency of definitional equality for System D
- Type soundness (progress and preservation) for both languages
- Decidable type checking for System DC
- Annotation and erasure lemmas relating the two languages

In this section, we provide an overview of these proofs and discuss their interaction with this extension. In the formal statements of our results below, we include the source file and definition in our Coq proofs[4] that justifies that result.

The type soundness proof comes in two parts. We prove the progress lemma for System D, and then use the annotation lemma to translate that result to System DC. We prove the preservation lemmas for both systems directly, but it would also be possible to only prove preservation for System DC and then use the erasure lemma to translate that proof to System D.

By far, the largest modification was needed for the proof of the progress lemma for System D, which in turn relies on the consistency of definitional equality.

4.1 Progress lemma overview

In order to show proof of progress, we must first show the consistency of definitional equality in our setting (see Corollary 7 below). Consistency means that in certain contexts, types that have different head forms cannot be proven definitionally equal.

▶ **Definition 1** (Consistent[5]). *Two types A and B are consistent, written* **consistent** $A\,B$, *when it is* not *the case that they are types with conflicting head forms. We formalize this property with the following two judgements.*

[4] Available from `https://github.com/sweirich/corespec/tree/master/src/FcEtt`.
[5] `ett.ott:consistent`

$\boxed{\mathsf{hft}(A)}$ (Types with head forms)

$$\frac{}{\mathsf{hft}(\mathsf{type})}\ \textsc{value-type-Star} \qquad \frac{}{\mathsf{hft}(\Pi^{\rho} x\!:\!A.B)}\ \textsc{value-type-Pi} \qquad \frac{}{\mathsf{hft}(\forall c\!:\!\phi.B)}\ \textsc{value-type-CPi}$$

$\boxed{\textbf{consistent}\ a\ b}$ (Types that do not differ in their heads)

$$\frac{}{\textbf{consistent}\ \mathsf{type}\ \mathsf{type}}\ \textsc{consistent-a-Star} \qquad \frac{}{\textbf{consistent}\ (\Pi^{\rho} x_1\!:\!A_1.B_1)\ (\Pi^{\rho} x_2\!:\!A_2.B_2)}\ \textsc{consistent-a-Pi}$$

$$\frac{}{\textbf{consistent}\ (\forall c_1\!:\!\phi_1.A_1)\ (\forall c_2\!:\!\phi_2.A_2)}\ \textsc{consistent-a-CPi} \qquad \frac{\neg(\mathsf{hft}(b))}{\textbf{consistent}\ a\ b}\ \textsc{consistent-a-Step-R} \qquad \frac{\neg(\mathsf{hft}(a))}{\textbf{consistent}\ a\ b}\ \textsc{consistent-a-Step-L}$$

We use two auxiliary relations, *parallel reduction* and *joinability*, when proving consistency.

Parallel reduction, written $\vDash a \Rightarrow b$, is not part of the specification of System D[6]. This relation is a strongly confluent, but not necessarily terminating, rewrite relation on terms. In one step of parallel reduction, multiple redexes in one term may be reduced at the same time. For example, we can reduce $(z\,((\lambda x.x)\,1)\,((\lambda y.y)\,2))$ to $(z\,1\,2)$ in one step, even though two different beta-reductions need to be performed at the same time.

Two types are *joinable* when they reduce to some common term using any number of steps of parallel reduction.

▶ **Definition 2** (Joinable[7]). *Two types are joinable, written* $\vdash a_1 \Leftrightarrow a_2$*, when there exists some* b *such that* $\vdash a_1 \Rightarrow^* b$ *and* $\vdash a_2 \Rightarrow^* b$*.*

We use these two relations to prove consistency in two steps. First, we show that definitionally equal types are joinable. Second, we show that joinable types are consistent.

In proving the first step, it is important to note that only *some* definitionally equal types are joinable. This is illustrated by the following example. If a has type type, and there is a coercion assumption $a \sim_{\mathsf{type}} \textbf{Int}$ available in the context, then under this assumption a and **Int** are two definitionally equal types. However, these two types are not joinable. Because our consistency proof is based on parallel reduction, and because parallel reduction ignores assumed equality propositions, we state our result only for equality derivations with no available coercion assumptions. Thus, we restrict the set of all available assumptions we can use to derive equality to the empty set.

▶ **Theorem 3** (Equality implies Joinability[8]). *If* $\Gamma; \varnothing \vDash a \equiv b : A$ *then* $\vdash a \Leftrightarrow b$

This restriction in the lemma is necessary because the type system does not rule out clearly bogus assumptions, such as $\textbf{Int} \sim_{\mathsf{type}} \textbf{Bool}$. Because we cannot use such assumptions to derive equality, they cannot be allowed to appear in the context. As a result, in order to be able to prove that consistent types are definitionally equal, the context must not make any such assumptions available.

To prove the second step, we use the fact that parallel reduction is a strongly confluent relation, and thus head forms must be preserved by parallel reduction. The confluence property is stated below.

[6] `ett.ott:Par`
[7] `ett.ott:join`
[8] `ext_consist.v:consistent_defeq`

▶ **Theorem 4** (Confluence[9]). *If* $\vDash a \Rightarrow a_1$ *and* $\vDash a \Rightarrow a_2$ *then there exists* b*, such that* $\vDash a_1 \Rightarrow b$ *and* $\vDash a_2 \Rightarrow b$*.*

Our proof of confluence for System D follows the the proof of Church-Rosser for the untyped lambda calculus given in Barendregt [11], sections 3.2 and 3.2. The proof with β-reduction is attributed to Tait and Martin-Löf, and its extension with η-reduction is attributed to Hindley [25] and Rosen [38].

The confluence property essentially shows that even if a term can take several reduction paths, those paths can never diverge to produce terms with conflicting head forms. Thus, since joinability is defined in terms of parallel reduction, and parallel reduction is strongly confluent, it is true that joinability implies consistency.

▶ **Lemma 5** (Joinability is transitive[10]). *If* $\vdash A_1 \Leftrightarrow B$ *and* $\vdash B \Leftrightarrow A_2$ *then* $\vdash A_1 \Leftrightarrow A_2$

▶ **Theorem 6** (Joinability implies consistency[11]). *If* $\vdash A \Leftrightarrow B$ *then* **consistent** $A\, B$*.*

▶ **Corollary 7** (Consistency). *If* $\Gamma; \Delta \vDash a \equiv b : A$ *then* **consistent** $a\, b$*.*

The consistency result allows us to prove the progress lemma for System D. This progress lemma is stated with respect to the one-step reduction relation and the definition of *value* given in Figure 2.

▶ **Lemma 8** (Progress[12]). *If* $\Gamma \vDash a : A$*,* Γ *contains no coercion assumptions, and no term variable* x *in the domain of* Γ *occurs free in* a*, then either* a *is a value or there exists some* a' *such that* $\vDash a \leadsto a'$*.*

4.2 Progress lemma update

The addition of η-equivalence required three new rules to be added to the parallel reduction relation. These rules encode η-reduction, meaning that any outer abstractions of the correct form can be removed. Because parallel reduction is an untyped relation, there is no analogous typing precondition as in the equivalence rules. However, these rules also have the condition that the bound variable not appear free in b or b'. (In our rules below, this condition is not explicitly mentioned because it is guaranteed by the usual Barendregt variable convention. We discuss how we maintain this property in our Coq development in Section 5.)

$$\textsc{Par-Eta}\quad \frac{\vDash b \Rightarrow b'}{\vDash \lambda^{+}x.b\ x^{+} \Rightarrow b'} \qquad \textsc{Par-EtaIrrel}\quad \frac{\vDash b \Rightarrow b'}{\vDash \lambda^{-}x.b\ \Box^{-} \Rightarrow b'} \qquad \textsc{Par-EtaC}\quad \frac{\vDash b \Rightarrow b'}{\vDash \Lambda c.b[\bullet] \Rightarrow b'}$$

We can view joinability as a semi-decision algorithm. Two terms are equal when they join to the same common reduct, though this process may diverge. This algorithm is a technical device only; we don't suggest its direct use in any implementation. Indeed, in the presence of η-reduction, joinability could equate more terms than definitional equality because it doesn't always preserve typing (see below).

[9] `ext_consist.v:confluence`
[10] `ext_consist.v:join_transitive`
[11] `ext_consist.v:join_consistent`
[12] `ext_consist.v:progress`

4.3 Parallel reduction and type preservation

There are three types of reduction included in this development: primitive reduction $\vDash a > b$, one-step reduction $\vDash a \leadsto b$, and parallel reduction $\vDash a \Rightarrow b$. In the original formulation of System D, all three of these reduction relations were type-preserving.

The first two relations are unchanged by this extension, so type preservation still holds for those relations[13].

However, parallel reduction is an untyped relation. It does not depend on type information, even in the case of η-equivalence. As a result, after the addition of η-equivalence rules, the parallel reduction relation is no longer type-preserving.

▶ **Example 9** (Parallel reduction does not preserve types)**.** There is some a such that $\Gamma \vDash a : A$ and $\vDash a \Rightarrow a'$ where there is no derivation of $\Gamma \vDash a' : A$.

This property fails in the case where $\lambda^{+}x.b\ x^{+}$ reduces to b, but x is required in the context for b to type check, even though it does not appear free in b.

For example, let A be $\Pi^{-}x\!:\!\mathsf{type}.\Pi^{+}z\!:\!\mathsf{type}.(x \to x)$ and consider the following derivation of the application of some function y with this type to two arguments: an implicit one and then an explicit one. In both cases in the derivation, the argument is just x, which is abstracted in the conclusion of the derivation.

$$\dfrac{\dfrac{\dfrac{\varnothing, y : A, x : \mathsf{type} \vDash y : A \qquad \varnothing, y : A, x : \mathsf{type} \vDash x : \mathsf{type}}{\varnothing, y : A, x : \mathsf{type} \vDash y\ \Box^{-} : \Pi^{+}z\!:\!\mathsf{type}.(x \to x)} \qquad \varnothing, y : A, x : \mathsf{type} \vDash x : \mathsf{type}}{\varnothing, y : A, x : \mathsf{type} \vDash y\ \Box^{-}\ x^{+} : x \to x}}{\varnothing, y : A \vDash \lambda^{+}x.(y\ \Box^{-})\ x^{+} : \Pi^{+}x\!:\!\mathsf{type}.(x \to x)}$$

Now, the term $\lambda^{+}x.y\ \Box^{-}\ x^{+}$ reduces to $y\ \Box^{-}$ using rule Par-Eta. However, there is no implicit argument that we can fill in so that this term will have type $\Pi^{+}x\!:\!\mathsf{type}.(x \to x)$.

Subject reduction also does not hold for η-reduction in the case of irrelevant arguments.[14] In particular, there is a case where $\lambda^{-}x.b\ \Box^{-}$ reduces to b and the two terms do not have the same type. This situation is not the same as above: the issue is that in a derivation of $\lambda^{-}x.b\ \Box^{-}$ there is no requirement that the argument $\Box$ be the same type as x.

For example, suppose y has type $\Gamma \vdash y : \Pi^{-}x : A.B$ and we have $f : A \to A'$ in the context Γ where the type A does not equal A'. Then we can construct a derivation of $\Gamma \vdash \lambda^{-}x.(y\ \Box^{-}) : \Pi^{-}x : A'.B\{f\,x/x\}$ by using the term $f\,x$ as the implicit argument. A similar counterexample also applies to η-reduction for coercion abstraction.

Thus, in the presence of η-reduction, preservation does not hold for parallel reduction. However, this loss is not significant to the soundness of the type systems of System D and System DC. None of our results require this property. The only place where this may come up is in a parallel-reduction based type inference algorithm for GHC (see Section 2.3). In this case, parallel reduction must preserve enough type information during reduction to ensure that the result is still well-typed.

4.4 Additional updates

Other updates to the proof include new cases in the erasure and annotation lemmas and in the uniqueness and decidability of type checking in DC. These lemmas are proven by mutual induction on the typing derivations shown in Figure 1. As the new rules are for the definitional/provable equality judgements, we only list that part of the lemma statement.

[13] `ext_red.v:Beta_preservation`, `ext_red.v:reduction_preservation`

[14] This issue was previously observed in the implementation of the Agda compiler: see `https://github.com/agda/agda/issues/2464`.

▶ **Lemma 10** (Erasure[15]). *If* $\Gamma; \Delta \vdash \gamma : a \sim b$ *then for all* A *such that* $\Gamma \vdash a : A$*, we have* $|\Gamma|; \Delta \vDash |a| \equiv |b| : |A|$.

▶ **Lemma 11** (Annotation[16]). *If* $\Gamma; \Delta \vDash a \equiv b : A$ *then for all* Γ_0*, such that* $|\Gamma_0| = \Gamma$*, there exists some* γ*,* a_0*,* b_0 *and* A_0*, such that* $\Gamma_0; \Delta \vdash \gamma : a_0 \sim b_0$ *and* $\Gamma_0 \vdash a_0 : A_0$ *and* $\Gamma_0 \vdash b_0 : A_0$ *where* $|a_0| = a$ *and* $|b_0| = b$ *and* $|A_0| = A$.

▶ **Lemma 12** (Unique typing for DC[17]). *If* $\Gamma; \Delta \vdash \gamma : A_1 \sim B_1$ *and* $\Gamma; \Delta \vdash \gamma : A_2 \sim B_2$*, then* $A_1 = A_2$ *and* $B_1 = B_2$.

▶ **Lemma 13** (Decidable typing for DC[18]). *Given* Γ*,* Δ*, and* γ*, it is decidable whether there exists some* A *and* B *such that* $\Gamma; \Delta \vdash \gamma : A \sim B$.

5 Proof engineering

The development of our Coq formalization for Systems D and DC was assisted with the use of two tools for mechanized reasoning about programming language metatheory. The first tool, Ott [40], takes as input a specification of the syntax and type system and produces both Coq definitions and LaTeX figures. The inference rules of this paper were typeset with this shared specification, though some rules in the main body of the paper have been slightly modified for clarity. We include the complete and unmodified specification of the system in Appendix A.

In addition to producing inductive definitions for the syntax and judgements, the Ott tool also produces substitution and free variable functions. To make working with these definitions more convenient, we also use the LNgen tool [9], that automatically states and proves many lemmas about these operations.

This extension increased the overall size of the original development by about ten percent, just looking at the line counts of the two versions. In Figure 4 we order the proof files by largest difference in line count[19] to see that the most significant effort was the update to the progress proofs for System D. The preservation proof file (`ext_red.v`) shrank due to the removal of the preservation lemma for the parallel reduction relation. The table includes some modifications (such as inserting a newline, or slight refactoring of proof scripts) that have no effect on the development. Files with unchanged line counts are omitted from this figure.

The `ett_ind.v` file contains tactics that are tailored to our language development. These tactics automatically apply inference rules, pick fresh variables with respect to binders, etc. As we have added new rules to the language definition, we needed to update these tactics. To assist in the rest of this proof development, we developed a tactic for automatically rewriting a term given a hypothesis of the form found in the η-rules (and similar).

The `ext_invert.v` file contains inversion lemmas for System D. New with this extension is the addition of a lemma that asserts that $\bullet$ is the only coercion proof found in System D terms.

[15] `erase.v:typing_erase`
[16] `erase.v:annotation_mutual`
[17] `fc_unique.v:unique_mutual`
[18] `fc_dec.v:FC_typechecking_decidable`
[19] These numbers were calculated using the `cloc` tool, version 1.76, available from `http://github.com/AlDanial/cloc`.

	File name	(1)	(1η)	(2)	(3)	(3η)
Specification (generated)	`ett_ott.v`	1337	1386	49	29	78
Progress (D)	`ext_consist.v`	1427	2054	627	205	832
Progress (D)	`ett_par.v`	660	1044	384	35	419
Erasure/annotation (D and DC)	`erase.v`	2002	2182	180	2	182
Decidability (DC)	`fc_dec_fun.v`	1561	1695	134	45	179
Progress (DC)	`fc_consist.v`	768	901	133	48	181
Inversion and regularity (D)	`ext_invert.v`	1057	1174	117	0	117
Inversion lemmas (DC)	`fc_invert.v`	650	665	15	82	97
Dec. of type checking (DC)	`fc_get.v`	774	844	70	1	71
General tactics	`ett_ind.v`	439	493	54	8	62
Preservation (D)	`ext_red.v`	290	241	-49	91	42
Context includes all vars (DC)	`fc_context_fv.v`	221	257	36	0	36
Context includes all vars (D)	`ext_context_fv.v`	143	178	35	0	35
Dec. of type checking (DC)	`fc_dec_aux.v`	395	399	4	18	22
Substitution (DC)	`fc_subst.v`	1270	1292	22	0	22
Unique typing (DC)	`fc_unique.v`	261	277	16	0	16
Reduction determinism (D)	`ext_red_one.v`	111	123	12	0	12
Substitution (D)	`ext_subst.v`	550	561	11	1	12
Primitive reduction	`beta.v`	71	78	7	4	11
Subst. prop. for coercions (DC)	`congruence.v`	349	354	5	0	5
Weakening (D)	`ext_weak.v`	139	141	2	3	5
Preservation (DC)	`fc_preservation.v`	247	245	-2	4	2
Well-formedness (D)	`ext_wf.v`	93	93	0	3	3
Dec. of type checking (DC)	`fc_dec_fuel.v`	223	223	0	2	2
Erasure properties	`erase_syntax.v`	486	486	0	1	1
General tactics	`tactics.v`	182	182	0	1	1
Total		17499	19404		554	2445

Figure 4 Comparison between line counts in the original [47] and extended proof developments. The columns are (1) - number of lines in the original, (1η) - number of lines in the extended version, (2) - change in line counts between the versions, (3) - size of diff for original, and (3η) - size of diff for the extended version. Files that are identical between the versions are not included in the table, but appear in the total line count. Note, all line counts include only non-blank, non-comment lines of code.

5.1 Stating rules for η-equivalence

One issue that we faced in our development is the precise characterization of the new η-equivalence rules using Ott. In the end, our actual formalization specifies these rules in a slightly different form than as presented in Section 3. For example, rule Par-Eta reads as follows, where we have named the body of the abstraction a and constrain it to be equal to the application as a premise of the rule.

$$\frac{\text{PAR-ETA} \\ \vDash b \Rightarrow b' \qquad a = b\ x^+}{\vDash \lambda^+ x.a \Rightarrow b'}$$

Although informally, this is a minor change, the precise statement of the rule determines the definitions that will be produced in Coq.

The generated Coq definition uses the *locally nameless* representation and co-finite quantification [8] for the bound variable inside the abstraction. Given any choice for the bound variable x (except for some variables that must be avoided in the set L), we can show that *opening* the body of the abstraction[20] produces an application of b to that variable. Furthermore, because this equation must hold for almost any variable x, we know that x could not have appeared in the term b to begin with.

```
Inductive Par : context -> available_set -> tm -> tm -> Prop :=
 ...
 | Par_Eta : forall (L:vars) (G:context) (D:available_set) (a b' b:tm),
     Par G D b b' ->
      (forall x, x \notin  L ->
          open a (Var_f x) = App b Rel (Var_f x))  ->
     Par G D (UAbs Rel a) b'
```

In the Ott version of the rule, we need not explicitly mention that x cannot appear free in b due to this use of cofinite quantification. Thus, the usual side condition on η-reduction is implied by our formulation of the rule in Ott and does not need to be stated again.

5.2 Confluence proof update

Updating the confluence proof with the new cases for these rules was fairly straightforward. In particular, Coq was easily able to point out the new cases that needed to be added.

One wrinkle was that the new cases required a change from an induction on the syntax of the term to an induction on the *height* of the term. The reason for this modification is that the new η-rules reduce b, which is not an *immediate* subterm of $\lambda^+ x.b\ x^+$. However, it is clear that in comparison to $\lambda^+ x.b\ x^+$ the term b has a smaller height. The induction on height of term was also effective for the other cases where we were dealing with immediate subterms. Furthermore, our tool support (LNgen) already defined an appropriate height function for terms which we were able to use for this purpose. Consequently, although we needed to adjust the use of induction in each case, the overall modifications were minor.

[20] The process of replacing the bound variable, represented by an index, with a free one.

6 Related work

6.1 Mechanized metatheory for dependent types

Mechanical reasoning via proof assistants has long been applied to dependent type theories. We will not attempt to describe all results. However, we will mention two recent developments:

Sozeau et al. [41] present the first implementation of a type checker for the kernel of Coq, which is proven correct in Coq with respect to its formal specification. More specifically, their work models an extension of the Predicative Calculus of (Co)-Inductive Constructions: a Pure Type System with an infinite hierarchy of universes, universe polymorphism, an impredicative sort, and inductive and co-inductive type families. However, although the Coq system includes η from version 8.4, this formalization does not include η-conversion. Like this work, their proofs of the metatheory of this system include a confluence proof of a parallel reduction relation, following Tait and Martin-Löf.

In [3], Abel, Öhman and Vezzozi mechanically prove (in Agda) the correctness of an algorithm for deciding conversion in a dependent type theory with one universe, an inductive type, and η-equality for function types. The algorithm that they verify is similar to the one used by Agda and is derived from Harper and Pfenning's definition of LF [24], as refined and extended by Scherer and Abel [4, 2]. The proof of correctness of this algorithm is based on a Kripke logical relations argument, parameterized by suitable notion of equivalence of terms.

6.2 Dependent types, type:type and η-equivalence

Similarly, the literature is rich with work pertaining to η-equivalence in type theories. Below, we will focus on the interaction with type:type systems. In the next subsection, we discuss the interactions with irrelevant arguments.

Many versions of the type:type language do not include η-equivalence in the definition of conversion. For example, Coquand presents a semi-decision procedure for type checking a language with type:type [18]. This algorithm compares types for equality through weak-head normalization only. Similarly, Abel and Altenkirch [1] provide a more modern implementation of the type checking algorithm for a very similar language (still without η-conversion), and prove completeness on terminating terms (with a terminating type).

One difficulty with η-reduction in this setting is the problem with confluence for Church-style calculi. To avoid a dependency between type checking and reduction, many dependent type systems rely on an untyped reduction relation. However, in Church-style systems, parallel reduction is only confluent for well-typed terms; ill-typed terms may not have a common reduct. For example, the term $(\lambda x : A.(\lambda y : B.y)\, x)$ can η-convert to $\lambda y : B.y$ or β-convert to $\lambda x{:}A.x$. These terms are only equal when A = B, but that is only guaranteed by well-typed terms. As System D is a Curry-style system however, it does not suffer from this issue.

Two versions of type:type that include η-equivalence are Cardelli [15] and Coquand and Takeyama [19]. Both of these works justify the soundness of the type systems and the consistency of the conversion relation using a denotational semantics. Furthermore, in both of these systems, the denotational semantics ignores the annotated domain types of lambda-expressions.

Coquand and Takeyama additionally provide a semi-decidable type checking algorithm. Their conversion algorithm is not based on parallel reduction; instead it follows Coquand's algorithm[17], reducing expressions to their weak-head-normal-forms before a structural comparison. When one of the terms being compared is a lambda expression and the other is not, the algorithm invents a fresh variable, applies both terms to this fresh variable and then continues checking for conversion.

	Q	DC	TE	η-F	η-T	Π	MM
P01 [37]	LF	✓	✓	✓	✓		
AS12 [4]	MLTT	✓	✓	✓	✓		
AVW17 [5]	MLTT	✓	✓	✓	✓	**2.**	
NVD17 [35]	MLTT	✓	✓			**3.**	
ND18 [34]	MLTT	**4.**	✓	✓	✓	**3.**	
A18 [7]	MLTT	**4.**	✓	**5.**	**5.**	✓	
M01 [32]	ECC			✓		✓	
BB08 [13]	ECC	✓		✓		✓	
MLS08 [33] IPTS	PTS					✓	
MLS08 [33] EPTS	PTS	✓				✓	
System D [47]	TT		✓	**1.**		✓	✓
System DC [47]	TT	✓	✓	**1.**		✓	✓

Notes:
1. Contribution of the current paper.
2. Only arguments of type *size* can be used without restriction.
3. Includes several different quantifiers, some with restriction, some without.
4. Not explicitly discussed in the paper. (But there are enough annotations that type checking is likely decidable.)
5. Definitional equality rules are not discussed in the paper, so the status is unclear.

Figure 5 Dependent type systems with irrelevance.

6.3 Irrelevant quantification and η-equivalence

In this section, we survey prior work on dependently-typed languages that include some form of irrelevant quantification and discuss their interaction with η-equivalence. The contents of this section are summarized in Figure 5, which compares these systems along the features described below.

Note that the terms "irrelevance" and "irrelevant quantification" have multiple meanings in the literature. Our primary focus is on erasability, the ability for terms to quantify over arguments that need not be present at runtime. However, this terminology often includes compile-time irrelevance, or the blindness of type equality to such erasable parts of terms. It can also refer to erasability in the compile-time type equivalence algorithm. These terms are also related to, but not the same as, "parametricity" or "parametric quantification", which characterizes functions that map equivalent arguments to equivalent results.

Below, we describe the various columns in this table that we use to lay out the design space of dependent type systems with irrelevance. Our purpose in this taxonomy is merely to define terms and summarize properties that we discuss below. We do not intend this table to characterize the contributions of prior work.

What form of type quantification is supported (Q)? First, we distinguish prior work by whether, and how, they support *type quantification*—that is, the ability for the system to quantify over types as well as terms. Type quantification is the foundation for *parametric polymorphism*, a key feature of modern programming languages, enabling modularity and code reuse. In dependent type systems, type quantification can take different forms, which have varying degrees of expressiveness. Prior work is based on the following foundations for type quantification:

LF [23], variants of the Logical Framework. This system includes dependency on terms only and does not allow quantification over types.

MLTT [30, 31], variants of Martin-Löf Type Theory. These systems feature predicative polymorphism only, where types are stratified into an infinite hierarchy of universes. A type from one universe can quantify only over types from lower universes.

ECC [16, 28], variants of the extended calculus of constructions. These systems feature an impredicative sort (called `Prop`), in addition to an infinite hierarchy of predicative universes. The types in the impredicative sort can quantify over themselves, all others must be stratified.

TT [29, 15], variants of core systems that include the `type:type` axiom. In these systems there is only a single sort of type, which includes types that quantify over all types. Systems D and DC include this form of quantification to make the system simpler for Haskell programmers, who are used to the impredicative polymorphism of System F.

PTS [10], pure type systems. These systems do not fix a single regime of type quantification. Instead, they may be instantiated with many different treatments of quantification, including all of the forms described above.

Is type checking decidable (DC)? Next, we distinguish systems based on whether they support decidable type checking (✓) or not ( ). Some calculi include enough annotations so that a decidable type checking algorithm can be defined, others merely specify when terms are well-typed. Sometimes the "same" system can be cast in two different variants. For example, System D does not support decidable type checking, System DC augments the syntax of terms with annotations for this purpose.[21]

Is the definition of equality typed (TE)? Does the conversion rule in the type system use a typed (✓) or untyped ( ) definition of equivalence? A typed equivalence requires a typed judgemental equality ([6]) and each transitive step used in the derivation to be between well-typed terms. In contrast, an untyped equivalence is usually defined in terms of β- or $\beta\eta$- conversion of terms, only checking that the endpoints are well typed.

This distinction can affect expressiveness in both directions. On the one hand, an untyped relation might equate terms with different types, or justify an equality using ill-typed terms. There may be no analogous derivation in a typed relation. On the other hand, some equivalence rules (like η for the unit type, see below) can only be included in the system when type information is present, thus *expanding* the relation.

The inclusion of typed equivalence relation means that the algorithm used for type checking may depend not just on the syntax of terms but also on their types during execution. This type information may be used to prevent two terms from being equated (for example, if one of the terms doesn't type check), or it may be used to enable two terms to be equated (such as in the case of the η-equivalence rule for the unit type).

Does the equality include η-equivalence rules for functions (η-F)? In this column, we include rules for functions regardless of whether they take relevant or irrelevant arguments. Note that some systems ([32]) do not mark the introduction and elimination sites of functions with irrelevant arguments. As a result, the corresponding equivalence rules are unnecessary. Similarly to other features, η-F (as well as η-T below) is important for programming as it may be used to derive equalities between types that mention functions, and thus to type-check more programs.

[21] Note, one typical location of annotation is the type of bound variables. Systems are often called "Church"-style when they include this annotation and "Curry"-style when they do not. However, this annotation is independent of the decidability of the type system, and many type systems that do not include this annotation support complete typing algorithms.

Does the equality include η-equivalence rules for products and unit (η-T)? Does the equality include type-directed η-equivalence rules for products or the unit type? For example, the rule for the unit type equates all terms of this type. Because this rule is type dependent, it can only be added to systems that use a typed definition of equivalence. These rules are typically implemented in the type system through a type-directed equivalence algorithm [24, 2].

At a high-level, the type-directed algorithm works in two stages. First, in the type-directed phase, if the terms being compared have function types, the two terms are applied to a fresh variable. This process takes care of η-equality. If the terms do not have function types, then the algorithm continues by converting both terms to weak-head normal form. If their heads match, then the algorithm recurses with the type-directed stage again on each of the corresponding subterms.

Is the codomain of the irrelevant Π-type unrestricted (Π)? In some systems, the *type* of an irrelevant abstraction is restricted so that the dependent argument must *also* be used irrelevantly. In other systems, the variable can appear freely without restrictions. Still others only allow unrestricted use for certain types of variables [5], or give users a choice [35, 34]. We discuss systems that include such restrictions, and their reasons for it, in Section 6.4. Systems D and DC do not restrict the codomain of irrelevant Π-types.

Mechanized metatheory (MM)? Have the metatheoretic results in the paper been developed and checked using a proof assistant? Our work is unique in this respect compared to similar systems.

6.4 Irrelevant quantification and restrictions on Π types

In this paper, we use irrelevance to mean erasure—i.e. the property that some arguments may be removed from the term without affecting the runtime behavior of the operational semantics. However, there is also a question of what happens to these arguments during type checking. Do these arguments affect the definition of type equality? If not, can they similarly be erased as part of a type checking algorithm?

Abel and Scherer [4] noted that although some arguments are irrelevant at run-time, they can still be relevant when determining type equality. If the definitional equality of the type system is typed, and if the type system allows *large eliminations*, i.e. the definition of a type via case analysis, then it can be difficult to incorporate type erasure into a type-directed equivalence algorithm. Fundamentally, the algorithm is driven by type information (instead of the structure of terms) and if irrelevant arguments can influence those types, they cannot be erased.

The key difficulty is demonstrated by the following example, taken from Abel and Scherer [4]. In the presence of large eliminations, and without any other restrictions, one would be able to type check the following term t, reproduced below in the syntax of DC extended with booleans.[22]

[22] Note that many systems support the large elimination needed for this example, even in the absence of inductive types. For example, in Systems D and DC we can use a Church-style encoding of booleans.

$$
\begin{array}{l}
T \ : \ \mathbf{Bool} \to \mathsf{type} \\
T = \lambda^{+} x \!:\! \mathbf{Bool}. \mathbf{if}\ x\ \mathbf{then}\ (\mathbf{Bool} \to \mathbf{Bool})\ \mathbf{else}\ \mathbf{Bool}
\end{array}
$$

$$
\begin{array}{ll}
t & = \lambda^{-} F : \Pi^{-} x \!:\! \mathbf{Bool}. (T\ x \to T\ x) \to \mathsf{type}. \\
& \quad \lambda^{+} f : (F\ \mathbf{False}^{-}\ (\lambda^{+} x \!:\! \mathbf{Bool}. x)^{+}) \to \mathbf{Bool}. \\
& \quad \lambda^{+} n : F\ \mathbf{True}^{-}\ (\lambda^{+} x \!:\! (\mathbf{Bool} \to \mathbf{Bool}). \lambda^{+} y \!:\! \mathbf{Bool}. x\ y^{+})^{+}. \\
& \qquad f\ (n \rhd \gamma)^{+}
\end{array}
$$

The DC coercion proof γ marks the point where conversion must be used in this example. This term is well-typed in a setting where the type system can derive an equality between the type of the parameter to f and the type of the argument n. These two types differ in only their irrelevant components, so they should be equated. In System DC, which, like ICC*, includes rules that erase types as part of type equivalence, we can define a coercion proof γ that witnesses the equality between the two types. Such a proof is composed transitively by first using the erasure-based reflexivity rule (rule AN-ERASEEQ) to change the implicit argument to F, and then using η-equivalence with the explicit argument.

$$
\begin{array}{rcl}
|F\ \mathbf{False}^{-}\ (\lambda^{+} x \!:\! \mathbf{Bool}. x)^{+}| & = & F\ \Box^{-}\ (\lambda^{+} x. x)^{+} \\
& =_{\beta\eta} & F\ \Box^{-}\ (\lambda^{+} x. \lambda^{+} y. x\ y^{+})^{+} \\
& = & |F\ \mathbf{True}^{-}\ (\lambda^{+} x \!:\! (\mathbf{Bool} \to \mathbf{Bool}). \lambda^{+} y \!:\! \mathbf{Bool}. x\ y^{+})^{+}|
\end{array}
$$

This example causes no difficulty for type checking in DC because it does not use a type-directed equivalence algorithm. Indeed, all of the information required by the algorithm is already present in the term.

However, it is difficult to extend a type-directed equivalence algorithm, particularly one that includes the η-equivalence rule for the unit type, so that it can equate these two types. Therefore, Abel and Scherer proposed restrictions on the use of irrelevantly quantified variables, not just in abstractions, but also in the codomain of irrelevant quantifiers. These restrictions were lifted in [5] for sized types, on the observation that they were irrelevant to the *shape* of types and therefore were not relevant to the operation of the type-equivalence algorithm. Nuyts and Devriese [35] expand on this idea and develop a modal type theory that includes, along with other modalities, irrelevance and shape-irrelevance in a unified framework.

However, note that the issue with this example is the desire to use erasure as part of a type-directed algorithm, not in the use of a typed equivalence in the language definition itself, nor the fact that the definition of type-equivalence ignores irrelevant components.

Because System DC does not rely on this sort of algorithm, it demonstrates that decidable type checking, irrelevance and large eliminations are compatible. Indeed, System DC requires the use of erasure in one of its key coercion proofs. On the other hand, one could worry that this example would cause trouble for System D. The fact that type checking is already undecidable in that language is not an excuse: a compiler like GHC will need to implement some type inference algorithm and should identify some subset of the language that it will support. This example demonstrates that type-directed algorithms are not a good fit for this setting, but does not rule out the algorithms sketched in Section 2.3.

7 Conclusion

Overall, this work demonstrates the benefits of developing the metatheory of type systems using a proof assistant. Although establishing the original development in prior work [47] took significant effort, we are able to build on that foundation when considering extensions of the system.

Furthermore, the availability of this sort of proof as a software engineering artifact makes it easier to bring on new collaborators. Because all of the proofs are machine-checked, newcomers can easily find what parts of the system need extension, even without understanding all details of how everything fits together. As a result, this sort of effort can be shared among many more collaborators, who can assist in maintaining the results.

Finally, the confidence gained from machine-checked proofs is also important. The failure of preservation for parallel η-reduction is obvious only in hindsight, and could have been easily overlooked in a pen-and-paper proof. At the same time, the automatic reassurance that this failure does not interact with the main soundness and decidability results is also welcome.

References

1 Andreas Abel and Thorsten Altenkirch. A partial type checking algorithm for Type:Type. *Electronic Notes in Theoretical Computer Science*, 229(5):3–17, 2011. Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008). `doi:10.1016/j.entcs.2011.02.013`.

2 Andreas Abel, Thierry Coquand, and Peter Dybjer. Normalization by evaluation for Martin-Löf type theory with typed equality judgements. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 3–12. IEEE, 2007.

3 Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *Proceedings of the acm on programming languages*, 2(POPL):23, 2017.

4 Andreas Abel and Gabriel Scherer. On irrelevance and algorithmic equality in predicative type theory. *Logical Methods in Computer Science*, 8(1), 2012. `doi:10.2168/LMCS-8(1:29)2012`.

5 Andreas Abel, Andrea Vezzosi, and Théo Winterhalter. Normalization by evaluation for sized dependent types. *PACMPL*, 1(ICFP):33:1–33:30, 2017. `doi:10.1145/3110277`.

6 ROBIN ADAMS. Pure type systems with judgemental equality. *Journal of Functional Programming*, 16(2):219–246, 2006. `doi:10.1017/S0956796805005770`.

7 Robert Atkey. The syntax and semantics of quantitative type theory. In *LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom*, 2018. `doi:10.1145/3209108.3209189`.

8 Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 3–15, January 2008.

9 Brian Aydemir and Stephanie Weirich. LNgen: Tool support for locally nameless representations. Technical Report MS-CIS-10-24, Computer and Information Science, University of Pennsylvania, June 2010.

10 H. P. Barendregt. *Lambda Calculi with Types*, page 117–309. Oxford University Press, Inc., USA, 1993.

11 Hendrik Pieter Barendregt. *The Lambda Calculus - its Syntax and Semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985.

12 Henk Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

13 Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. In Roberto Amadio, editor, *Foundations of Software Science and Computational Structures*, pages 365–379, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

14 Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, Durham University, 2005.

15 Luca Cardelli. A polymorphic λ-calculus with Type:Type. Technical report, DEC SRC, 1986. URL: `http://lucacardelli.name/Papers/TypeType.A4.pdf`.

16 Thierry Coquand. A calculus of constructions. manuscript, November 1986.

17 Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, New York, NY, USA, 1991.

18 Thierry Coquand. An algorithm for type-checking dependent types. *Science of computer programming.*, 26(1-3):167,177, 1996-05.

19 Thierry Coquand and Makoto Takeyama. An implementation of type: type. In *International Workshop on Types for Proofs and Programs*, pages 53–62. Springer, 2000.

20 Richard A. Eisenberg. *Dependent Types in Haskell: Theory and Practice*. PhD thesis, University of Pennsylvania, 2016.

21 Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *ACM SIGPLAN Haskell Symposium*, 2012.

22 Adam Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde, 2013.

23 Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993. `doi:10.1145/138027.138060`.

24 Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Trans. Comput. Logic*, 6(1):61–101, January 2005. `doi:10.1145/1042038.1042041`.

25 J. Roger Hindley. *The Church-Rosser property and a result in combinatory logic*. PhD thesis, University of Newcastle upon Tyne, 1964.

26 Csongor Kiss, Tony Field, Susan Eisenbach, and Simon Peyton Jones. Higher-order type-level programming in haskell. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. `doi:10.1145/3341706`.

27 Sam Lindley and Conor McBride. Hasochism: the pleasure and pain of dependently typed Haskell programming. In *ACM SIGPLAN Haskell Symposium*, 2013.

28 Z. Luo. ECC, an extended calculus of constructions. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 386–395, 1989.

29 Per Martin-Löf. A theory of types. Unpublished manuscript, 1971.

30 Per Martin-Löf. An intuitionistic theory of types: predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73, Proceedings of the Logic Colloquium*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1975.

31 Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, 1984.

32 Alexandre Miquel. The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications*, TLCA'01, pages 344–359, Berlin, Heidelberg, 2001. Springer-Verlag.

33 Nathan Mishra-Linger and Tim Sheard. Erasure and polymorphism in pure type systems. In Roberto M. Amadio, editor, *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, volume 4962 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2008. `doi:10.1007/978-3-540-78499-9_25`.

34 Andreas Nuyts and Dominique Devriese. Degrees of relatedness: A unified framework for parametricity, irrelevance, ad hoc polymorphism, intersections, unions and algebra in dependent type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 779–788. ACM, 2018. doi:10.1145/3209108.3209119.

35 Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. Parametric quantifiers for dependent type theory. *Proc. ACM Program. Lang.*, 1(ICFP):32:1–32:29, August 2017. doi:10.1145/3110276.

36 Frank Pfenning. On the undecidability of partial polymorphic type reconstruction. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

37 Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In J. Halpern, editor, *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, pages 221–230, Boston, Massachusetts, June 2001. IEEE Computer Society Press.

38 Barry K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *J. ACM*, 20(1):160–187, January 1973. doi:10.1145/321738.321750.

39 Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. Guarded impredicative polymorphism. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 783–796. ACM, 2018. doi:10.1145/3192366.3192389.

40 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1), January 2010.

41 Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq coq correct! verification of type checking and erasure for coq, in coq. *Proc. ACM Program. Lang.*, 4(POPL):8:1–8:28, 2020. doi:10.1145/3371076.

42 M. Sulzmann, M. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In François Pottier and George C. Necula, editors, *Proceedings of TLDI'07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007*, pages 53–66. ACM, 2007.

43 The Coq Development Team. The Coq proof assistant, version 8.8.0, April 2018. doi:10.5281/zenodo.1219885.

44 Dimitrios Vytiniotis, Simon L. Peyton Jones, Tom Schrijvers, and Martin Sulzmann. Outsidein(x) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412, 2011. doi:10.1017/S0956796811000098.

45 Stephanie Weirich. Depending on types, 2014. Invited keynote given at ICFP 2014.

46 Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. System FC with explicit kind equality. In *Proceedings of The 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 275–286, Boston, MA, September 2013.

47 Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A. Eisenberg. A specification for dependent types in Haskell. *Proc. ACM Program. Lang.*, 1(ICFP):31:1–31:29, August 2017. doi:10.1145/3110275.

48 J.B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1):111–156, 1999. doi:10.1016/S0168-0072(98)00047-5.

49 A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, November 1994. doi:10.1006/inco.1994.1093.

50 Ningning Xie, Richard A. Eisenberg, and Bruno C. d. S. Oliveira. Kind inference for datatypes. *Proc. ACM Program. Lang.*, 4(POPL):53:1–53:28, 2020. doi:10.1145/3371121.

A Complete system specification

The complete type system appears in here including the actual rules that we used, automatically generated by Ott. For presentation purposes, we have removed some redundant hypotheses from these rules in the main body of the paper when they were implied via regularity. We have proven (in Coq) that these additional premises are admissible, so their removal does not change the type system.[23] These redundant hypotheses are marked by square brackets in the complete system below.

We need to include these redundant hypotheses in our rules for two reasons. First, sometimes these hypotheses simplify the reasoning and allow us to prove properties more independently of one another. For example, in the rule E-Beta rule, we require a_2 to have the same type as a_1. However, this type system supports the preservation lemma so this typing premise will always be derivable. But, it is convenient to prove the regularity property early, so we include that hypothesis in the definition of the type system.

Another source of redundancy comes from our use of the Coq proof assistant. Some of our proofs require the use of induction on judgements that are not direct premises, but are derived from other premises via regularity. These derivations are always the same height or shorter than the original, so this use of induction is justified. However, while Coq natively supports proofs by induction on derivations, it does not natively support induction on the *heights* of derivations. Therefore, to make these induction hypotheses available for reasoning, we include them as additional premises.

Finally, instead of the usual syntactic distinction of values (as in Figure 2), our formalization identifies values using the judgement $[\mathsf{Value}\ a]$, overloaded for both System D and System DC terms.

B Top-level signatures

Our results are proven with respect to the following top-level signatures:

$$\Sigma_1 = \emptyset \cup \{\mathbf{Fix} \sim \lambda^{-} x\!:\!\mathsf{type}.\lambda^{+} y\!:\!x.(y\,(\mathbf{Fix}[x]\,y)) : \Pi^{-} x\!:\!\mathsf{type}.(x \to x) \to x\}$$

$$\Sigma_0 = |\Sigma_1|$$

However, our Coq proofs use these signature definitions opaquely. As a result, any pair of top-level signatures are compatible with the definition of the languages as long as they satisfy the following properties.

1. $\vDash \Sigma_0$
2. $\vdash \Sigma_1$
3. $\Sigma_0 = |\Sigma_1|$

[23] `ext_invert.v:E_Pi2,E_Abs2,E_CPi2,E_CAbs2,E_Fam2`, `ext_invert.v:E_Wff2,E_PiCong2,E_AbsCong2,E_CPiCong2,E_CAbsCong2`, `ext_red.v:E_Beta2`, `fc_invert.v:An_Pi_exists2,An_Abs_exists2,An_CPi_exists2,An_CAbs_exists2,An_Fam2`, `fc_invert.v:An_Sym2,An_Trans2,An_AbsCong_exists2`, `fc_invert.v:An_AppCong2,An_CPiCong_exists2,An_CAppCong2`

C Reduction relations

C.1 Primitive reduction

$\boxed{\vDash a > b}$ *(primitive reductions on erased terms)*

$$\textsc{Beta-AppAbs}\quad \frac{}{\vDash (\lambda^{+}x.v)\ b^{+} > v\{b/x\}}$$

$$\textsc{Beta-AppAbsIrrel}\quad \frac{[\mathsf{Value}\ (\lambda^{-}x.v)]}{\vDash (\lambda^{-}x.v)\ \square^{-} > v\{\square/x\}}$$

$$\textsc{Beta-CAppCAbs}\quad \frac{}{\vDash (\Lambda c.a')[\bullet] > a'\{\bullet/c\}}$$

$$\textsc{Beta-Axiom}\quad \frac{F \sim a : A \in \Sigma_0}{\vDash F > a}$$

C.2 System D one-step reduction

$\boxed{\vDash a \leadsto b}$ *(single-step head reduction for implicit language)*

$$\textsc{E-AbsTerm}\quad \frac{\vDash a \leadsto a'}{\vDash \lambda^{-}x.a \leadsto \lambda^{-}x.a'}$$

$$\textsc{E-AppLeft}\quad \frac{\vDash a \leadsto a'}{\vDash a\ b^{+} \leadsto a'\ b^{+}}$$

$$\textsc{E-AppLeftIrrel}\quad \frac{\vDash a \leadsto a'}{\vDash a\ \square^{-} \leadsto a'\ \square^{-}}$$

$$\textsc{E-CAppLeft}\quad \frac{\vDash a \leadsto a'}{\vDash a[\bullet] \leadsto a'[\bullet]}$$

$$\textsc{E-AppAbs}\quad \frac{}{\vDash (\lambda^{+}x.v)\ a^{+} \leadsto v\{a/x\}}$$

$$\textsc{E-AppAbsIrrel}\quad \frac{[\mathsf{Value}\ (\lambda^{-}x.v)]}{\vDash (\lambda^{-}x.v)\ \square^{-} \leadsto v\{\square/x\}}$$

$$\textsc{E-CAppCAbs}\quad \frac{}{\vDash (\Lambda c.b)[\bullet] \leadsto b\{\bullet/c\}}$$

$$\textsc{E-Axiom}\quad \frac{F \sim a : A \in \Sigma_0}{\vDash F \leadsto a}$$

C.3 System DC one-step reduction

$\boxed{\Gamma \vdash a \leadsto b}$ *(single-step, weak head reduction to values for annotated language)*

$$\textsc{An-AppLeft}\quad \frac{\Gamma \vdash a \leadsto a'}{\Gamma \vdash a\ b^{\rho} \leadsto a'\ b^{\rho}}$$

$$\textsc{An-AppAbs}\quad \frac{[\mathsf{Value}\ (\lambda^{\rho}x\!:\!A.w)]}{\Gamma \vdash (\lambda^{\rho}x\!:\!A.w)\ a^{\rho} \leadsto w\{a/x\}}$$

$$\textsc{An-CAppLeft}\quad \frac{\Gamma \vdash a \leadsto a'}{\Gamma \vdash a[\gamma] \leadsto a'[\gamma]}$$

$$\textsc{An-CAppCAbs}\quad \frac{}{\Gamma \vdash (\Lambda c\!:\!\phi.b)[\gamma] \leadsto b\{\gamma/c\}}$$

$$\textsc{An-AbsTerm}\quad \frac{\Gamma \vdash A : \mathsf{type} \qquad \Gamma, x : A \vdash b \leadsto b'}{\Gamma \vdash (\lambda^{-}x\!:\!A.b) \leadsto (\lambda^{-}x\!:\!A.b')}$$

$$\textsc{An-Axiom}\quad \frac{F \sim a : A \in \Sigma_1}{\Gamma \vdash F \leadsto a}$$

$$\textsc{An-ConvTerm}\quad \frac{\Gamma \vdash a \leadsto a'}{\Gamma \vdash a \triangleright \gamma \leadsto a' \triangleright \gamma}$$

$$\textsc{An-Combine}\quad \frac{[\mathsf{Value}\ v]}{\Gamma \vdash (v \triangleright \gamma_1) \triangleright \gamma_2 \leadsto v \triangleright (\gamma_1 ; \gamma_2)}$$

$$\textsc{An-Push}\quad \frac{\begin{array}{c}[\mathsf{Value}\ v] \\ \Gamma; \widetilde{\Gamma} \vdash \gamma : \Pi^{\rho}x_1\!:\!A_1.B_1 \sim \Pi^{\rho}x_2\!:\!A_2.B_2 \\ b' = b \triangleright \mathbf{sym}\,(\mathbf{piFst}\,\gamma) \\ \gamma' = \gamma@(b' \mid = \mid_{(\mathbf{piFst}\,\gamma)}\ b)\end{array}}{\Gamma \vdash (v \triangleright \gamma)\ b^{\rho} \leadsto (v\ b'^{\rho}) \triangleright \gamma'}$$

$$\textsc{An-CPush}\quad \frac{\begin{array}{c}[\mathsf{Value}\ v] \\ \Gamma; \widetilde{\Gamma} \vdash \gamma : \forall c_1\!:\!\phi_1.A_1 \sim \forall c_2\!:\!\phi_2.A_2 \\ \gamma_1' = \gamma_1 \triangleright \mathbf{sym}\,(\mathbf{cpiFst}\,\gamma) \\ \gamma' = \gamma@(\gamma_1' \sim \gamma_1)\end{array}}{\Gamma \vdash (v \triangleright \gamma)[\gamma_1] \leadsto (v[\gamma_1']) \triangleright \gamma'}$$

C.4 Parallel reduction

$\boxed{\vDash a \Rightarrow b}$ *(parallel reduction (implicit language))*

$$\textsc{Par-Refl}\ \frac{}{\vDash a \Rightarrow a}$$

$$\textsc{Par-Beta}\ \frac{\begin{array}{c}\vDash a \Rightarrow (\lambda^+ x.a') \\ \vDash b \Rightarrow b'\end{array}}{\vDash a\ b^+ \Rightarrow a'\{b'/x\}}$$

$$\textsc{Par-BetaIrrel}\ \frac{\vDash a \Rightarrow (\lambda^- x.a')}{\vDash a\ \Box^- \Rightarrow a'\{\Box/x\}}$$

$$\textsc{Par-App}\ \frac{\vDash a \Rightarrow a' \qquad \vDash b \Rightarrow b'}{\vDash a\ b^+ \Rightarrow a'\ b'^+}$$

$$\textsc{Par-AppIrrel}\ \frac{\vDash a \Rightarrow a'}{\vDash a\ \Box^- \Rightarrow a'\ \Box^-}$$

$$\textsc{Par-CBeta}\ \frac{\vDash a \Rightarrow (\Lambda c.a')}{\vDash a[\bullet] \Rightarrow a'\{\bullet/c\}}$$

$$\textsc{Par-CApp}\ \frac{\vDash a \Rightarrow a'}{\vDash a[\bullet] \Rightarrow a'[\bullet]}$$

$$\textsc{Par-Abs}\ \frac{\vDash a \Rightarrow a'}{\vDash \lambda^\rho x.a \Rightarrow \lambda^\rho x.a'}$$

$$\textsc{Par-Pi}\ \frac{\vDash A \Rightarrow A' \qquad \vDash B \Rightarrow B'}{\vDash \Pi^\rho x{:}A.B \Rightarrow \Pi^\rho x{:}A'.B'}$$

$$\textsc{Par-CAbs}\ \frac{\vDash a \Rightarrow a'}{\vDash \Lambda c.a \Rightarrow \Lambda c.a'}$$

$$\textsc{Par-CPi}\ \frac{\begin{array}{c}\vDash A \Rightarrow A' \qquad \vDash B \Rightarrow B' \\ \vDash a \Rightarrow a' \qquad \vDash A_1 \Rightarrow A_1'\end{array}}{\vDash \forall c{:}A \sim_{A_1} B.a \Rightarrow \forall c{:}A' \sim_{A_1'} B'.a'}$$

$$\textsc{Par-Axiom}\ \frac{F \sim a : A \in \Sigma_0}{\vDash F \Rightarrow a}$$

$$\textsc{Par-Eta}\ \frac{\vDash b \Rightarrow b' \qquad a = b\ x^+}{\vDash \lambda^+ x.a \Rightarrow b'}$$

$$\textsc{Par-EtaIrrel}\ \frac{\vDash b \Rightarrow b' \qquad a = b\ \Box^-}{\vDash \lambda^- x.a \Rightarrow b'}$$

$$\textsc{Par-EtaC}\ \frac{\vDash b \Rightarrow b' \qquad a = b[\bullet]}{\vDash \Lambda c.a \Rightarrow b'}$$

D Full system specification: System D type system

$\boxed{\Gamma \vDash a : A}$ *(typing)*

$$\textsc{E-Star}\ \frac{\vDash \Gamma}{\Gamma \vDash \mathsf{type} : \mathsf{type}}$$

$$\textsc{E-Var}\ \frac{\vDash \Gamma \qquad x : A \in \Gamma}{\Gamma \vDash x : A}$$

$$\textsc{E-Pi}\ \frac{\begin{array}{c}\Gamma, x : A \vDash B : \mathsf{type} \\ [\Gamma \vDash A : \mathsf{type}]\end{array}}{\Gamma \vDash \Pi^\rho x{:}A.B : \mathsf{type}}$$

$$\textsc{E-Abs}\ \frac{\begin{array}{c}\Gamma, x : A \vDash a : B \\ [\Gamma \vDash A : \mathsf{type}] \\ (\rho = +) \lor (x \notin \mathsf{fv}\ a)\end{array}}{\Gamma \vDash \lambda^\rho x.a : \Pi^\rho x{:}A.B}$$

$$\textsc{E-App}\ \frac{\begin{array}{c}\Gamma \vDash b : \Pi^+ x{:}A.B \\ \Gamma \vDash a : A\end{array}}{\Gamma \vDash b\ a^+ : B\{a/x\}}$$

$$\textsc{E-IApp}\ \frac{\begin{array}{c}\Gamma \vDash b : \Pi^- x{:}A.B \\ \Gamma \vDash a : A\end{array}}{\Gamma \vDash b\ \Box^- : B\{a/x\}}$$

$$\textsc{E-Conv}\ \frac{\begin{array}{c}\Gamma \vDash a : A \\ \Gamma; \widetilde{\Gamma} \vDash A \equiv B : \mathsf{type} \\ [\Gamma \vDash B : \mathsf{type}]\end{array}}{\Gamma \vDash a : B}$$

$$\textsc{E-CPi}\ \frac{\begin{array}{c}\Gamma, c : \phi \vDash B : \mathsf{type} \\ [\Gamma \vDash \phi\ \mathsf{ok}]\end{array}}{\Gamma \vDash \forall c{:}\phi.B : \mathsf{type}}$$

$$\textsc{E-CAbs}\ \frac{\begin{array}{c}\Gamma, c : \phi \vDash a : B \\ [\Gamma \vDash \phi\ \mathsf{ok}]\end{array}}{\Gamma \vDash \Lambda c.a : \forall c{:}\phi.B}$$

$$\textsc{E-CApp}\ \frac{\begin{array}{c}\Gamma \vDash a_1 : \forall c{:}(a \sim_A b).B_1 \\ \Gamma; \widetilde{\Gamma} \vDash a \equiv b : A\end{array}}{\Gamma \vDash a_1[\bullet] : B_1\{\bullet/c\}}$$

$$\textsc{E-Fam}\ \frac{\begin{array}{c}\vDash \Gamma \qquad F \sim a : A \in \Sigma_0 \\ [\varnothing \vDash A : \mathsf{type}]\end{array}}{\Gamma \vDash F : A}$$

$\boxed{\Gamma \vDash \phi\ \mathsf{ok}}$ *(Prop wellformedness)*

$$\textsc{E-Wff}\ \frac{\begin{array}{c}\Gamma \vDash a : A \qquad \Gamma \vDash b : A \\ [\Gamma \vDash A : \mathsf{type}]\end{array}}{\Gamma \vDash a \sim_A b\ \mathsf{ok}}$$

$\boxed{\Gamma;\Delta \vDash \phi_1 \equiv \phi_2}$ *(prop equality)*

$$\textsc{E-PropCong}\quad \frac{\Gamma;\Delta \vDash A_1 \equiv A_2 : A \qquad \Gamma;\Delta \vDash B_1 \equiv B_2 : A}{\Gamma;\Delta \vDash A_1 \sim_A B_1 \equiv A_2 \sim_A B_2}$$

$$\textsc{E-IsoConv}\quad \frac{\Gamma;\Delta \vDash A \equiv B : \mathsf{type} \qquad \Gamma \vDash A_1 \sim_A A_2\ \ \mathsf{ok} \qquad \Gamma \vDash A_1 \sim_B A_2\ \ \mathsf{ok}}{\Gamma;\Delta \vDash A_1 \sim_A A_2 \equiv A_1 \sim_B A_2}$$

$$\textsc{E-CPiFst}\quad \frac{\Gamma;\Delta \vDash \forall c{:}\phi_1.B_1 \equiv \forall c{:}\phi_2.B_2 : \mathsf{type}}{\Gamma;\Delta \vDash \phi_1 \equiv \phi_2}$$

$\boxed{\Gamma;\Delta \vDash a \equiv b : A}$ *(definitional equality)*

$$\textsc{E-Assn}\quad \frac{\vDash \Gamma \qquad c : (a \sim_A b) \in \Gamma \qquad c \in \Delta}{\Gamma;\Delta \vDash a \equiv b : A}$$

$$\textsc{E-Refl}\quad \frac{\Gamma \vDash a : A}{\Gamma;\Delta \vDash a \equiv a : A}$$

$$\textsc{E-Sym}\quad \frac{\Gamma;\Delta \vDash b \equiv a : A}{\Gamma;\Delta \vDash a \equiv b : A}$$

$$\textsc{E-Trans}\quad \frac{\Gamma;\Delta \vDash a \equiv a_1 : A \qquad \Gamma;\Delta \vDash a_1 \equiv b : A}{\Gamma;\Delta \vDash a \equiv b : A}$$

$$\textsc{E-Beta}\quad \frac{\Gamma \vDash a_1 : B \qquad [\Gamma \vDash a_2 : B] \qquad \vDash a_1 > a_2}{\Gamma;\Delta \vDash a_1 \equiv a_2 : B}$$

$$\textsc{E-PiCong}\quad \frac{\begin{array}{c}\Gamma;\Delta \vDash A_1 \equiv A_2 : \mathsf{type}\\ \Gamma, x : A_1;\Delta \vDash B_1 \equiv B_2 : \mathsf{type}\\ [\Gamma \vDash A_1 : \mathsf{type}]\\ [\Gamma \vDash \Pi^\rho x{:}A_1.B_1 : \mathsf{type}]\\ [\Gamma \vDash \Pi^\rho x{:}A_2.B_2 : \mathsf{type}]\end{array}}{\Gamma;\Delta \vDash (\Pi^\rho x{:}A_1.B_1) \equiv (\Pi^\rho x{:}A_2.B_2) : \mathsf{type}}$$

$$\textsc{E-AbsCong}\quad \frac{\begin{array}{c}\Gamma, x : A_1;\Delta \vDash b_1 \equiv b_2 : B\\ [\Gamma \vDash A_1 : \mathsf{type}]\\ (\rho = +) \vee (x \notin \mathsf{fv}\ b_1)\\ (\rho = +) \vee (x \notin \mathsf{fv}\ b_2)\end{array}}{\Gamma;\Delta \vDash (\lambda^\rho x.b_1) \equiv (\lambda^\rho x.b_2) : \Pi^\rho x{:}A_1.B}$$

$$\textsc{E-AppCong}\quad \frac{\Gamma;\Delta \vDash a_1 \equiv b_1 : \Pi^+ x{:}A.B \qquad \Gamma;\Delta \vDash a_2 \equiv b_2 : A}{\Gamma;\Delta \vDash a_1\ {a_2}^+ \equiv b_1\ {b_2}^+ : B\{a_2/x\}}$$

$$\textsc{E-IAppCong}\quad \frac{\Gamma;\Delta \vDash a_1 \equiv b_1 : \Pi^- x{:}A.B \qquad \Gamma \vDash a : A}{\Gamma;\Delta \vDash a_1\ \Box^- \equiv b_1\ \Box^- : B\{a/x\}}$$

$$\textsc{E-PiFst}\quad \frac{\Gamma;\Delta \vDash \Pi^\rho x{:}A_1.B_1 \equiv \Pi^\rho x{:}A_2.B_2 : \mathsf{type}}{\Gamma;\Delta \vDash A_1 \equiv A_2 : \mathsf{type}}$$

$$\textsc{E-CPiCong}\quad \frac{\begin{array}{c}\Gamma;\Delta \vDash \phi_1 \equiv \phi_2\\ \Gamma, c : \phi_1;\Delta \vDash A \equiv B : \mathsf{type}\\ [\Gamma \vDash \phi_1\ \ \mathsf{ok}]\\ [\Gamma \vDash \forall c{:}\phi_1.A : \mathsf{type}]\\ [\Gamma \vDash \forall c{:}\phi_2.B : \mathsf{type}]\end{array}}{\Gamma;\Delta \vDash \forall c{:}\phi_1.A \equiv \forall c{:}\phi_2.B : \mathsf{type}}$$

$$\textsc{E-PiSnd}\quad \frac{\Gamma;\Delta \vDash \Pi^\rho x{:}A_1.B_1 \equiv \Pi^\rho x{:}A_2.B_2 : \mathsf{type} \qquad \Gamma;\Delta \vDash a_1 \equiv a_2 : A_1}{\Gamma;\Delta \vDash B_1\{a_1/x\} \equiv B_2\{a_2/x\} : \mathsf{type}}$$

$$\textsc{E-CAbsCong}\quad \frac{\Gamma, c : \phi_1;\Delta \vDash a \equiv b : B \qquad [\Gamma \vDash \phi_1\ \ \mathsf{ok}]}{\Gamma;\Delta \vDash (\Lambda c.a) \equiv (\Lambda c.b) : \forall c{:}\phi_1.B}$$

$$\textsc{E-CAppCong}\quad \frac{\Gamma;\Delta \vDash a_1 \equiv b_1 : \forall c{:}(a \sim_A b).B \qquad \Gamma;\widetilde{\Gamma} \vDash a \equiv b : A}{\Gamma;\Delta \vDash a_1[\bullet] \equiv b_1[\bullet] : B\{\bullet/c\}}$$

$$\textsc{E-CPiSnd}\quad \frac{\begin{array}{c}\Gamma;\Delta \vDash \forall c{:}(a_1 \sim_A a_2).B_1 \equiv \forall c{:}(a_1' \sim_{A'} a_2').B_2 : \mathsf{type}\\ \Gamma;\widetilde{\Gamma} \vDash a_1 \equiv a_2 : A\\ \Gamma;\widetilde{\Gamma} \vDash a_1' \equiv a_2' : A'\end{array}}{\Gamma;\Delta \vDash B_1\{\bullet/c\} \equiv B_2\{\bullet/c\} : \mathsf{type}}$$

$$\textsc{E-Cast}\quad \frac{\begin{array}{c}\Gamma;\Delta \vDash a \equiv b : A\\ \Gamma;\Delta \vDash a \sim_A b \equiv a' \sim_{A'} b'\end{array}}{\Gamma;\Delta \vDash a' \equiv b' : A'}$$

$$\textsc{E-EqConv}\quad \frac{\begin{array}{c}\Gamma;\Delta \vDash a \equiv b : A\\ \Gamma;\widetilde{\Gamma} \vDash A \equiv B : \mathsf{type}\end{array}}{\Gamma;\Delta \vDash a \equiv b : B}$$

$$\textsc{E-IsoSnd}\quad \frac{\Gamma;\Delta \vDash a \sim_A b \equiv a' \sim_{A'} b'}{\Gamma;\Delta \vDash A \equiv A' : \mathsf{type}}$$

$$\textsc{E-EtaRel}\quad \frac{\begin{array}{c}\Gamma \vDash b : \Pi^+ x{:}A.B\\ a = b\ x^+\end{array}}{\Gamma;\Delta \vDash \lambda^+ x.a \equiv b : \Pi^+ x{:}A.B}$$

$$\textsc{E-EtaIrrel}\quad \frac{\begin{array}{c}\Gamma \vDash b : \Pi^- x{:}A.B\\ a = b\ \Box^-\end{array}}{\Gamma;\Delta \vDash \lambda^- x.a \equiv b : \Pi^- x{:}A.B}$$

$$\textsc{E-EtaC}\quad \frac{\begin{array}{c}\Gamma \vDash b : \forall c{:}\phi.B\\ a = b[\bullet]\end{array}}{\Gamma;\Delta \vDash \Lambda c.a \equiv b : \forall c{:}\phi.B}$$

$\boxed{\vDash \Gamma}$ *(context wellformedness)*

$$\textsc{E-Empty}\quad \frac{}{\vDash \varnothing}$$

$$\textsc{E-ConsTm}\quad \frac{\begin{array}{c}\vDash \Gamma \qquad \Gamma \vDash A : \mathsf{type}\\ x \notin \mathsf{dom}\,\Gamma\end{array}}{\vDash \Gamma, x : A}$$

$$\textsc{E-ConsCo}\quad \frac{\begin{array}{c}\vDash \Gamma\\ \Gamma \vDash \phi\ \ \mathsf{ok} \qquad c \notin \mathsf{dom}\,\Gamma\end{array}}{\vDash \Gamma, c : \phi}$$

$\boxed{\vDash \Sigma}$ *(signature wellformedness)*

$$\textsc{Sig-Empty}\quad \frac{}{\vDash \varnothing}$$

$$\textsc{Sig-ConsAx}\quad \frac{\begin{array}{c}\vDash \Sigma \qquad \varnothing \vDash A : \mathsf{type}\\ \varnothing \vDash a : A \qquad F \notin \mathsf{dom}\,\Sigma\end{array}}{\vDash \Sigma \cup \{F \sim a : A\}}$$

E Full system specification: System DC type system

$\boxed{\Gamma \vdash a : A}$ *(typing)*

$$\textsc{An-Star}\quad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{type} : \mathsf{type}}$$

$$\textsc{An-Var}\quad \frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\textsc{An-Pi}\quad \frac{\begin{array}{c}\Gamma, x : A \vdash B : \mathsf{type}\\ [\Gamma \vdash A : \mathsf{type}]\end{array}}{\Gamma \vdash \Pi^\rho x{:}A.B : \mathsf{type}}$$

$$\textsc{An-Abs}\quad \frac{\begin{array}{c}[\Gamma \vdash A : \mathsf{type}]\\ \Gamma, x : A \vdash a : B\\ (\rho = +) \vee (x \notin \mathsf{fv}\ |a|)\end{array}}{\Gamma \vdash \lambda^\rho x{:}A.a : \Pi^\rho x{:}A.B}$$

$$\textsc{An-App}\quad \frac{\begin{array}{c}\Gamma \vdash b : \Pi^\rho x{:}A.B\\ \Gamma \vdash a : A\end{array}}{\Gamma \vdash b\ a^\rho : B\{a/x\}}$$

$$\textsc{An-Conv}\quad \frac{\begin{array}{c}\Gamma \vdash a : A\\ \Gamma;\widetilde{\Gamma} \vdash \gamma : A \sim B\\ \Gamma \vdash B : \mathsf{type}\end{array}}{\Gamma \vdash a \rhd \gamma : B}$$

$$\textsc{An-CPi}\quad \frac{\begin{array}{c}[\Gamma \vdash \phi\ \ \mathsf{ok}]\\ \Gamma, c : \phi \vdash B : \mathsf{type}\end{array}}{\Gamma \vdash \forall c{:}\phi.B : \mathsf{type}}$$

$$\textsc{An-CAbs}\quad \frac{\begin{array}{c}[\Gamma \vdash \phi\ \ \mathsf{ok}]\\ \Gamma, c : \phi \vdash a : B\end{array}}{\Gamma \vdash \Lambda c{:}\phi.a : \forall c{:}\phi.B}$$

$$\textsc{An-CApp}\quad \frac{\begin{array}{c}\Gamma \vdash a_1 : \forall c{:}a \sim_{A_1} b.B\\ \Gamma;\widetilde{\Gamma} \vdash \gamma : a \sim b\end{array}}{\Gamma \vdash a_1[\gamma] : B\{\gamma/c\}}$$

$$\textsc{An-Fam}\quad \frac{\begin{array}{c}\vdash \Gamma \qquad F \sim a : A \in \Sigma_1\\ [\varnothing \vdash A : \mathsf{type}]\end{array}}{\Gamma \vdash F : A}$$

$\boxed{\Gamma \vdash \phi\ \mathsf{ok}}$ *(prop wellformedness)*

$$\textsc{An-Wff}\quad \frac{\begin{array}{c}\Gamma \vdash a : A \\ \Gamma \vdash b : B \qquad |A| = |B|\end{array}}{\Gamma \vdash a \sim_A b\ \mathsf{ok}}$$

$\boxed{\Gamma ; \Delta \vdash \gamma : \phi_1 \sim \phi_2}$ *(coercion between props)*

$$\textsc{An-PropCong}\quad \frac{\begin{array}{c}\Gamma ; \Delta \vdash \gamma_1 : A_1 \sim A_2 \\ \Gamma ; \Delta \vdash \gamma_2 : B_1 \sim B_2 \\ \Gamma \vdash A_1 \sim_A B_1\ \mathsf{ok} \\ \Gamma \vdash A_2 \sim_A B_2\ \mathsf{ok}\end{array}}{\Gamma ; \Delta \vdash (\gamma_1 \sim_A \gamma_2) : (A_1 \sim_A B_1) \sim (A_2 \sim_A B_2)}$$

$$\textsc{An-CPiFst}\quad \frac{\Gamma ; \Delta \vdash \gamma : \forall c{:}\phi_1 . A_2 \sim \forall c{:}\phi_2 . B_2}{\Gamma ; \Delta \vdash \mathbf{cpiFst}\, \gamma : \phi_1 \sim \phi_2}$$

$$\textsc{An-IsoSym}\quad \frac{\Gamma ; \Delta \vdash \gamma : \phi_1 \sim \phi_2}{\Gamma ; \Delta \vdash \mathbf{sym}\, \gamma : \phi_2 \sim \phi_1}$$

$$\textsc{An-IsoConv}\quad \frac{\begin{array}{c}\Gamma ; \Delta \vdash \gamma : A \sim B \\ \Gamma \vdash a_1 \sim_A a_2\ \mathsf{ok} \\ \Gamma \vdash a_1' \sim_B a_2'\ \mathsf{ok} \\ |a_1| = |a_1'| \qquad |a_2| = |a_2'|\end{array}}{\Gamma ; \Delta \vdash \mathbf{conv}\ (a_1 \sim_A a_2) \sim_\gamma (a_1' \sim_B a_2') : (a_1 \sim_A a_2) \sim (a_1' \sim_B a_2')}$$

$\boxed{\Gamma ; \Delta \vdash \gamma : A \sim B}$ *(coercion between types)*

$$\textsc{An-Assn}\quad \frac{\begin{array}{c}\vdash \Gamma \\ c : a \sim_A b\ \in \Gamma \qquad c \in \Delta\end{array}}{\Gamma ; \Delta \vdash c : a \sim b}$$

$$\textsc{An-Refl}\quad \frac{\Gamma \vdash a : A}{\Gamma ; \Delta \vdash \mathbf{refl}\, a : a \sim a}$$

$$\textsc{An-EraseEq}\quad \frac{\begin{array}{c}\Gamma \vdash a : A \\ \Gamma \vdash b : B \qquad |a| = |b| \\ \Gamma ; \widetilde{\Gamma} \vdash \gamma : A \sim B\end{array}}{\Gamma ; \Delta \vdash (a \mathrel{|{=}|_\gamma} b) : a \sim b}$$

$$\textsc{An-Sym}\quad \frac{\begin{array}{c}\Gamma \vdash b : B \qquad \Gamma \vdash a : A \\ [\Gamma ; \widetilde{\Gamma} \vdash \gamma_1 : B \sim A] \\ \Gamma ; \Delta \vdash \gamma : b \sim a\end{array}}{\Gamma ; \Delta \vdash \mathbf{sym}\, \gamma : a \sim b}$$

$$\textsc{An-Trans}\quad \frac{\begin{array}{c}\Gamma ; \Delta \vdash \gamma_1 : a \sim a_1 \\ \Gamma ; \Delta \vdash \gamma_2 : a_1 \sim b \\ [\Gamma \vdash a : A] \\ [\Gamma \vdash a_1 : A_1] \\ [\Gamma ; \widetilde{\Gamma} \vdash \gamma_3 : A \sim A_1]\end{array}}{\Gamma ; \Delta \vdash (\gamma_1 ; \gamma_2) : a \sim b}$$

$$\textsc{An-Beta}\quad \frac{\begin{array}{c}\Gamma \vdash a_1 : B_0 \\ \Gamma \vdash a_2 : B_1 \\ |B_0| = |B_1| \\ \vDash |a_1| > |a_2|\end{array}}{\Gamma ; \Delta \vdash \mathbf{red}\, a_1\, a_2 : a_1 \sim a_2}$$

$$\textsc{An-PiCong}\quad \frac{\begin{array}{c}\Gamma ; \Delta \vdash \gamma_1 : A_1 \sim A_2 \\ \Gamma, x : A_1 ; \Delta \vdash \gamma_2 : B_1 \sim B_2 \\ B_3 = B_2\{x \triangleright \mathbf{sym}\, \gamma_1 / x\} \\ \Gamma \vdash \Pi^\rho x{:}A_1 . B_1 : \mathsf{type} \\ \Gamma \vdash \Pi^\rho x{:}A_2 . B_3 : \mathsf{type} \\ \Gamma \vdash (\Pi^\rho x{:}A_1 . B_2) : \mathsf{type}\end{array}}{\Gamma ; \Delta \vdash \Pi^\rho x{:}\gamma_1 . \gamma_2 : (\Pi^\rho x{:}A_1 . B_1) \sim (\Pi^\rho x{:}A_2 . B_3)}$$

An-AbsCong

$$\dfrac{\begin{array}{c}\Gamma;\Delta\vdash\gamma_1:A_1\sim A_2\\ \Gamma,x:A_1;\Delta\vdash\gamma_2:b_1\sim b_2\\ b_3=b_2\{x\triangleright\mathbf{sym}\,\gamma_1/x\}\\ [\Gamma\vdash A_1:\mathsf{type}]\\ \Gamma\vdash A_2:\mathsf{type}\\ (\rho=+)\vee(x\notin\mathsf{fv}\,|b_1|)\\ (\rho=+)\vee(x\notin\mathsf{fv}\,|b_3|)\\ [\Gamma\vdash(\lambda^{\rho}x\!:\!A_1.b_2):B]\end{array}}{\Gamma;\Delta\vdash(\lambda^{\rho}x\!:\!\gamma_1.\gamma_2):(\lambda^{\rho}x\!:\!A_1.b_1)\sim(\lambda^{\rho}x\!:\!A_2.b_3)}$$

An-AppCong

$$\dfrac{\begin{array}{c}\Gamma;\Delta\vdash\gamma_1:a_1\sim b_1\\ \Gamma;\Delta\vdash\gamma_2:a_2\sim b_2\\ \Gamma\vdash a_1\ {a_2}^{\rho}:A\\ \Gamma\vdash b_1\ {b_2}^{\rho}:B\\ [\Gamma;\widetilde{\Gamma}\vdash\gamma_3:A\sim B]\end{array}}{\Gamma;\Delta\vdash\gamma_1\ \gamma_2^{\rho}:a_1\ {a_2}^{\rho}\sim b_1\ {b_2}^{\rho}}$$

An-PiFst

$$\dfrac{\Gamma;\Delta\vdash\gamma:\Pi^{\rho}x\!:\!A_1.B_1\sim\Pi^{\rho}x\!:\!A_2.B_2}{\Gamma;\Delta\vdash\mathbf{piFst}\,\gamma:A_1\sim A_2}$$

An-PiSnd

$$\dfrac{\begin{array}{c}\Gamma;\Delta\vdash\gamma_1:\Pi^{\rho}x\!:\!A_1.B_1\sim\Pi^{\rho}x\!:\!A_2.B_2\\ \Gamma;\Delta\vdash\gamma_2:a_1\sim a_2\\ \Gamma\vdash a_1:A_1\\ \Gamma\vdash a_2:A_2\end{array}}{\Gamma;\Delta\vdash\gamma_1@\gamma_2:B_1\{a_1/x\}\sim B_2\{a_2/x\}}$$

An-CPiCong

$$\dfrac{\begin{array}{c}\Gamma;\Delta\vdash\gamma_1:\phi_1\sim\phi_2\\ \Gamma,c:\phi_1;\Delta\vdash\gamma_3:B_1\sim B_2\\ B_3=B_2\{c\triangleright\mathbf{sym}\,\gamma_1/c\}\\ \Gamma\vdash\forall c\!:\!\phi_1.B_1:\mathsf{type}\\ [\Gamma\vdash\forall c\!:\!\phi_2.B_3:\mathsf{type}]\\ \Gamma\vdash\forall c\!:\!\phi_1.B_2:\mathsf{type}\end{array}}{\Gamma;\Delta\vdash(\forall c\!:\!\gamma_1.\gamma_3):(\forall c\!:\!\phi_1.B_1)\sim(\forall c\!:\!\phi_2.B_3)}$$

An-CAbsCong

$$\dfrac{\begin{array}{c}\Gamma;\Delta\vdash\gamma_1:\phi_1\sim\phi_2\\ \Gamma,c:\phi_1;\Delta\vdash\gamma_3:a_1\sim a_2\\ a_3=a_2\{c\triangleright\mathbf{sym}\,\gamma_1/c\}\\ \Gamma\vdash(\Lambda c\!:\!\phi_1.a_1):\forall c\!:\!\phi_1.B_1\\ \Gamma\vdash(\Lambda c\!:\!\phi_2.a_3):\forall c\!:\!\phi_2.B_2\\ \Gamma\vdash(\Lambda c\!:\!\phi_1.a_2):B\\ \Gamma;\widetilde{\Gamma}\vdash\gamma_4:\forall c\!:\!\phi_1.B_1\sim\forall c\!:\!\phi_2.B_2\end{array}}{\Gamma;\Delta\vdash(\lambda c\!:\!\gamma_1.\gamma_3@\gamma_4):(\Lambda c\!:\!\phi_1.a_1)\sim(\Lambda c\!:\!\phi_2.a_3)}$$

An-CAppCong

$$\dfrac{\begin{array}{c}\Gamma;\Delta\vdash\gamma_1:a_1\sim b_1\\ \Gamma;\widetilde{\Gamma}\vdash\gamma_2:a_2\sim b_2\\ \Gamma;\widetilde{\Gamma}\vdash\gamma_3:a_3\sim b_3\\ \Gamma\vdash a_1[\gamma_2]:A\\ \Gamma\vdash b_1[\gamma_3]:B\\ [\Gamma;\widetilde{\Gamma}\vdash\gamma_4:A\sim B]\end{array}}{\Gamma;\Delta\vdash\gamma_1(\gamma_2,\gamma_3):a_1[\gamma_2]\sim b_1[\gamma_3]}$$

An-CPiSnd

$$\dfrac{\begin{array}{c}\Gamma;\Delta\vdash\gamma_1:(\forall c_1\!:\!a\sim_A a'.B_1)\sim(\forall c_2\!:\!b\sim_B b'.B_2)\\ \Gamma;\widetilde{\Gamma}\vdash\gamma_2:a\sim a'\\ \Gamma;\widetilde{\Gamma}\vdash\gamma_3:b\sim b'\end{array}}{\Gamma;\Delta\vdash\gamma_1@(\gamma_2\sim\gamma_3):B_1\{\gamma_2/c_1\}\sim B_2\{\gamma_3/c_2\}}$$

An-Cast

$$\dfrac{\begin{array}{c}\Gamma;\Delta\vdash\gamma_1:a\sim a'\\ \Gamma;\Delta\vdash\gamma_2:a\sim_A a'\sim b\sim_B b'\end{array}}{\Gamma;\Delta\vdash\gamma_1\triangleright\gamma_2:b\sim b'}$$

An-IsoSnd

$$\dfrac{\Gamma;\Delta\vdash\gamma:(a\sim_A a')\sim(b\sim_B b')}{\Gamma;\Delta\vdash\mathbf{isoSnd}\,\gamma:A\sim B}$$

An-Eta

$$\dfrac{\begin{array}{c}\Gamma\vdash b:\Pi^{\rho}x\!:\!A.B\\ a=b\ x^{\rho}\end{array}}{\Gamma;\Delta\vdash\mathbf{eta}\,b:(\lambda^{\rho}x\!:\!A.a)\sim b}$$

$$\textsc{An-EtaC}\quad \frac{\Gamma \vdash b : \forall c{:}\phi.B \qquad a = b[c]}{\Gamma;\Delta \vdash \mathbf{eta}\, b : (\Lambda c{:}\phi.a) \sim b}$$

$\boxed{\vdash \Gamma}$ *(context wellformedness)*

$$\textsc{An-Empty}\quad \frac{}{\vdash \varnothing}$$

$$\textsc{An-ConsTm}\quad \frac{\vdash \Gamma \qquad \Gamma \vdash A : \mathsf{type} \qquad x \notin \mathsf{dom}\,\Gamma}{\vdash \Gamma, x : A}$$

$$\textsc{An-ConsCo}\quad \frac{\vdash \Gamma \qquad \Gamma \vdash \phi\ \mathsf{ok} \qquad c \notin \mathsf{dom}\,\Gamma}{\vdash \Gamma, c : \phi}$$

$\boxed{\vdash \Sigma}$ *(signature wellformedness)*

$$\textsc{An-Sig-Empty}\quad \frac{}{\vdash \varnothing}$$

$$\textsc{An-Sig-ConsAx}\quad \frac{\vdash \Sigma \qquad \varnothing \vdash A : \mathsf{type} \qquad \varnothing \vdash a : A \qquad F \notin \mathsf{dom}\,\Sigma}{\vdash \Sigma \cup \{F \sim a : A\}}$$