# A Specification for Dependently-Typed Haskell (Extended version)

STEPHANIE WEIRICH, University of Pennsylvania
ANTOINE VOIZARD, University of Pennsylvania
PEDRO HENRIQUE AZEVEDO DE AMORIM, Ecole Polytechnique and University of Campinas
RICHARD EISENBERG, Bryn Mawr College

We propose a semantics for Dependent Haskell, an extension of Haskell with full-spectrum dependent types. Our semantics consists of two strongly related languages. The first is a Curry-style dependently-typed language with nontermination, irrelevant arguments, and equality abstraction. The second, strongly inspired by GHC's core language FC, is its explicitly-typed analogue, suitable for implementation. In contrast to prior work, our design demonstrates that homogeneous equality is compatible with explicit equality proofs. All of our results—chiefly, type safety, along with theorems that relate our two semantics—have been formalized using the Coq proof assistant.

CCS Concepts: •**Software and its engineering → General programming languages;**

Additional Key Words and Phrases: Haskell, Dependent Types

**ACM Reference format:**
Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard Eisenberg. 2017. A Specification for Dependently-Typed Haskell (Extended version). *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 34 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Our goal is to design *Dependent Haskell*, an extension of Haskell with *full-spectrum* dependent types. The main feature of Dependent Haskell is that, unlike current Haskell, it makes no distinction between types and terms; both compile-time and runtime computation share the same syntax and semantics.

For example, in current Haskell,[1] length-indexed vectors may be indexed only by type-level structures. So in the definition below, we say that `Vec` is a GADT (Cheney and Hinze 2003; Peyton Jones et al. 2006; Vytiniotis et al. 2011) indexed by the *promoted datatype* `Nat` (Yorgey et al. 2012).[2]

```
data Nat :: Type where        data Vec :: Type -> Nat -> Type where
  O :: Nat                      Nil  :: Vec a O
  S :: Nat -> Nat               (:>) :: a -> Vec a m -> Vec a (S m)
```

If we want to compute one of these promoted natural numbers to use as the index of a vector, we must define a type-level function. Regular Haskell functions are not applicable to type-level data. As a result, programmers must duplicate their definitions if they would like them to be available at both compile-time and runtime.

However, Dependent Haskell makes no such distinctions. In this language, the definition of `Vec` is written exactly as above—but the meaning is that elements of type `Nat` are just normal values. As a result, we can use standard Haskell terms, such as `one` and `plus` below, directly in types.

[1]Glasgow Haskell Compiler (GHC), version 8.0.1, with extensions
[2]In this version of GHC, the kind of ordinary types can be written `Type` as well as `*`. We prefer the new spelling.

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

```
one :: Nat                plus :: Nat -> Nat -> Nat
one = S O                 plus O     y = y
                          plus (S x) y = S (plus x y)

example :: Vec Char (one `plus` one `plus` one)
example = 'G' :> 'H' :> 'C' :> Nil
```

Dependent Haskell is a planned extension of the Glasgow Haskell Compiler (GHC). GHC is an ideal vehicle for type system research. Not only is it a mature implementation with an industrial-strength optimizer, but the design of the compiler itself is well suited to experimentation. In particular, GHC's front-end elaborates source Haskell programs to an explicitly typed core language, called FC (Sulzmann et al. 2007). As a result, researchers can explore semantic consequences of their designs independent of interactions with type inference. However, because FC defines the semantics of the Haskell language as implemented by GHC, all type system extensions must first be realized in FC.

The FC language is based on an explicitly typed variant of System F (Girard 1971; Reynolds 1974) with *type equality coercions*. These coercions provide evidence for type equalities, necessary to support type-level computation (Schrijvers et al. 2008) and GADTs. The presence of explicit types and type equality evidence means that FC has a decidable, syntax-directed type checking algorithm.

**This paper defines the semantics of Dependent Haskell by developing a dependently typed replacement for FC that we call System DC.** This version of the core language retains FC's explicit coercion proofs but replaces System F with a calculus with full-spectrum dependent types. The result is a calculus with a rich, decidable type system that can serve as a basis for the extension of Haskell with dependent types while still supporting existing Haskell programs.

The key idea that makes this work is the observation that we can replace FC in a backwards compatible way as long as the dependently-typed core language supports *irrelevant quantification* (Barras and Bernardo 2008; Miquel 2001; Pfenning 2001). Haskell is an efficient language because (among many other optimizations) GHC erases types during compilation. Even though we conflate types and terms in DC, we must retain the ability to perform this erasure. Therefore, DC disentangles the notion of "type" from that of "erasable component". Irrelevant quantification marks all terms (whether they are types or not) as erasable as long as they can be safely removed without changing the behavior of a program.

Our design of DC is strongly based on two recent dissertations that combine type equality coercions and irrelevant quantification in dependently-typed core calculi (Eisenberg 2016; Gundry 2013) as well as an extension of FC with kind equalities (Weirich et al. 2013). Although DC is inspired by this prior work, we make several improvements to these designs (see Section 8.1). The most important change is that we show that ***homogeneous* equality is compatible with explicit coercion proofs**. Prior work bases equality coercions on *heterogeneous* equality (McBride 2000), where terms are not required to have related types in an equality proposition. However, *homogeneous* equality is the most standard design: it is commonly used in dependent type systems that do not employ explicit coercions. By changing our treatment of equality, we are able to make simplifications both to the language semantics and the proofs of its metatheoretic properties (see Section 6.1). This change, however, makes the language no less expressive.

A second significant contribution of our work is that, in parallel with DC, we develop **System D, an *implicitly* typed version of DC**. D is a Curry-style language that does not support decidable type checking, similar to implicit System F[3]. However, D is otherwise equivalent to DC; we show that these languages type related programs and that those related programs have the same runtime behavior. In particular, any program in DC can be erased

[3]We use the words type checking and type inference interchangeably—they are equivalent in this setting and both problems are undecidable (Pfenning 1992; Wells 1999).

to a well-typed program in D, and for any typing derivation in D, there exists a well-typed program in DC that erases to it (see Section 5).

The design of D exchanges decidable type checking for a simpler specification, as we show in Sections 3 and 5. This calculus is simpler than DC because it does not need to do as much bookkeeping; only computationally relevant information appears in D terms. As a result, the proofs of key metatheoretic properties, such as the consistency of definitional equality, are also simpler in D. This has the benefit that we can avoid some complications in reasoning about DC by appealing to analogous results about D.

It is also the case that D is more canonical than DC. There are many ways to annotate terms in support of decidable type checking. There are accordingly many variants of DC that we can prove equivalent (through erasure) to D. We propose one such variant in this paper. We believe that we have chosen a design for DC that can be fully integrated into GHC. Some of our design choices are in support of that implementation. However, we discuss alternatives in Section 6.3.

Finally, D itself serves as an inspiration for type inference in the source language. Although type checking is undecidable, it serves as an "ideal" that clever inference algorithms can approximate. This process has already happened for the System FC-based core language: some GHC extensions augment Damas-Milner type inference (Damas and Milner 1982) with features of System F, such as first-class polymorphism (Peyton Jones et al. 2007; Vytiniotis et al. 2008) and visible type applications (Eisenberg et al. 2016).

Our final contribution is a **mechanization of all of the metatheory of this paper** using the Coq proof assistant (Coq development team 2004). This contribution is significant because these proofs require a careful analysis of the allowable interactions between dependent types, coercion abstraction, nontermination and irrelevance. This combination is hard to get right and at least two previous efforts have suffered from errors, as we describe in Section 7.2. Furthermore, many of our own initial designs of the two languages did not work, in intricate, hard-to-spot ways. Formalizing all the proofs in Coq provides a level of confidence about our results that we could not possibly achieve otherwise. Moreover, these results are available for further extension.

This paper concludes with a discussion that relates our work to the large field of research in this area (Section 8). In particular, we provide a close comparison of DC to prior extensions of FC with dependent types (Eisenberg 2016; Gundry 2013; Weirich et al. 2013) and with existing dependently-typed calculi and languages.

## 2 SYSTEM D, SYSTEM DC AND THE DESIGN OF DEPENDENT HASKELL

One purpose of GHC's core language is to give a semantics to Haskell programs in a manner that is independent of type inference. This division is important: it allows language designers to experiment with various type inference algorithms, while still preserving the semantics of their program. It also inspires Haskell source language extensions with features that do not admit effective type inference, through the use of type annotations.

Below, we give an example that illustrates the key features of DC and D, by showing how source-level Dependent Haskell expressions can be elaborated into an explicitly typed core. Note that the DC and D calculi that we define in this paper are designed to investigate the interaction between dependent types, coercion abstraction, irrelevant arguments and nontermination. The examples below demonstrate how these features interact in an implementation, like GHC, that includes primitive datatypes and pattern matching. For simplicity, DC and D do not include these as primitive, but can encode these examples using standard techniques.[4]

Consider the `zip` function, which combines two equal-length vectors into a vector of pairs, using the datatypes `Nat` and `Vec` from the introduction.

```
zip :: forall n a b. Vec a n -> Vec b n -> Vec (a,b) n
zip Nil       Nil       = Nil
zip (x :> xs) (y :> ys) = (x, y) :> zip xs ys
```

[4] Such as a Scott encoding (see page 504 of Curry et al. (1972)).

The type of `zip` is dependent because the first argument (a natural number) appears later in the type. For efficiency, we also do not want this argument around at runtime, so we mark it as erasable by using the "`forall n`" quantifier. The similarity between this code and current Haskell is intentional. In fact, this program already compiles with GHC 8.0. However, the meaning of this program is different here—remember that `n` is an invisible and irrelevant *term* argument in Dependent Haskell, not a promoted datatype.

The `zip` function type-checks because in the first branch `n` is equal to zero, so `Nil` has a type equal to `Vec a n`. In the second branch, when `n` is equal to `S m`, then the result of the recursive call has type `Vec a m`, so the result type of the branch is `Vec a (S m)`, also equal to `Vec a n`. This pattern matching is exhaustive because the two vectors have the same length; the two remaining patterns are not consistent with the annotated type.

In an explicitly-typed core language, such as DC, we use typing annotations to justify this reasoning. First, consider the elaborated version of the `Vec` datatype definition shown below. This definition explicitly binds the argument `m` to the `(:>)` constructor, using `forall` to note that the argument need not be stored at runtime. Furthermore, the type of each data constructor includes a context of equality constraints, describing the information gained during pattern matching.

```
data Vec (a :: Type) (n :: Nat) :: Type where
   Nil  :: (n ~ 0) => Vec a n
   (:>) :: forall (m :: Nat). (n ~ S m) => a -> Vec a m -> Vec a n
```

The core language version of `zip`, shown below, uses the binder `\-` to abstract irrelevant arguments and the binder `/\` to abstraction coercions. Each branch of a case quantifies over the corresponding arguments to the data constructor according to the types above, including the coercions `(n ~ 0)` and `(n ~ S m)`.

```
zip = \-n:Nat. \-a:Type. \-b:Type. \xs:Vec a n. \ys:Vec a n. case xs of
   Nil -> /\c1:(n ~ 0). case ys of
      Nil -> /\c2:(n ~ 0). Nil [a][n][c1]
      (:>) -> \-m:Nat. /\c2:(n ~ S m). \y:b. \ys:Vec b m. absurd [sym c1; c2]
   (:>) -> \m:Nat. /\c1:(n ~ S m). \x:a. \xs:Vec a m. case ys of
      Nil -> /\c2:(n ~ 0). absurd [sym c1; c2]
      (:>) -> \-m:Nat. /\c2:(n ~ S m). \y:b. \ys:Vec b m.
          (:>) [a][n][m][c1] ((,) [a][b] x y) (zip [m][a][b] xs ys)
```

The core language `zip` function must provide all arguments to data constructors and functions, even those that are inferred in the source language. Arguments that are not relevant to computation are marked with square brackets. These arguments include the datatype parameters (`n` and `a`) as well as explicit proofs for the equality constraints (`c1`). This example also shows how the elaborator compiles away nested pattern matching and requires cases for every data constructor. The impossible cases are marked with explicit proofs of contradiction, in this case that `(O ~ S m)`.

Although the explicit arguments and coercions simplify type checking, they obscure the meaning of terms like `zip`. Furthermore, there are many possible ways of annotating programs in support of decidable type checking—it would be good to know that the choice of annotation does not affect the meaning of a particular program. For example, the `Nil [a][n][c1]` case above could be replaced with `Nil [a][n][c2]` instead, because both `c1` and `c2` are proofs of the same equality. We would like to know that making this change does not affect the definition of `zip`.

To better understand `zip`, we can erase these annotations, as in System D.

```
zip = \-n. \-a. \-b. \xs. \ys. case xs of
   Nil -> /\c1. case ys of
      Nil -> /\c2. Nil [][][]
      (:>) -> \-m. /\c2. \y. \ys. absurd []
```

| | D | DC |
|---|---|---|
| *Typing* | $\Gamma \vDash a : A$ | $\Gamma \vdash a : A$ |
| *Proposition well-formedness* | $\Gamma \vDash \phi$ ok | $\Gamma \vdash \phi$ ok |
| *Definitional equality (terms)* | $\Gamma; \Delta \vDash a \equiv b : A$ | $\Gamma; \Delta \vdash \gamma : a \sim b$ |
| *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* | $\vDash a \rightsquigarrow b$ | $\Gamma \vdash a \rightsquigarrow b$ |

Fig. 1. Summary of judgment forms

| | | | |
|---|---|---|---|
| *terms, types* | $a, b, A, B$ | $::=$ | $\star \mid x \mid F \mid \lambda^{\rho} x.b \mid a\ b^{\rho} \mid \Box$ |
| | | $\mid$ | $\Pi^{\rho} x{:}A \to B \mid \Lambda c.a \mid a[\gamma] \mid \forall c{:}\phi.A$ |
| *propositions* | $\phi$ | $::=$ | $a \sim_A b$ |
| *relevance* | $\rho$ | $::=$ | $+ \mid -$ |
| *coercions* | $\gamma$ | $::=$ | $\bullet$ |
| | | | |
| *values* | $v$ | $::=$ | $\lambda^{+} x.a \mid \lambda^{-} x.v \mid \Lambda c.a \mid \star \mid \Pi^{\rho} x{:}A \to B \mid \forall c{:}\phi.A$ |
| | | | |
| *contexts* | $\Gamma$ | $::=$ | $\varnothing \mid \Gamma, x : A \mid \Gamma, c : \phi$ |
| *available set* | $\Delta$ | $::=$ | $\varnothing \mid \Delta, c$ |
| *signature* | $\Sigma$ | $::=$ | $\varnothing \mid \Sigma \cup \{F \sim a : A\}$ |

Fig. 2. Syntax of D

```
Cons -> \m. /\c1. \x. \xs. case ys of
   Nil -> /\c2. absurd []
   (:>) -> \-m. /\c2. \y. \ys.
       (:>) [][][][] ((,) [][] x y) (zip [][][] xs ys)
```

Besides being more similar to the source, this version captures exactly what this code does at runtime. It also justifies equating the two differently annotated versions in the type system. Because they erase to the same term, we know that the annotations chosen by the type inferencer do not affect the runtime behavior of the program.

## 3 SYSTEM D: A LANGUAGE WITH IMPLICIT EQUALITY PROOFS

We now make the definitions of the two languages of this paper precise. These two languages share parallel structure in their definitions. This is no coincidence. The annotated language DC is, in some sense, a *reification* of the implicit language derivations in D. To emphasize this connection, we reuse the same metavariables for analogous syntax in both languages.[5] The judgment forms for both languages are summarized in Figure 1.

The syntax of D, the implicit language, is shown in Figure 2. This language, inspired by pure type systems (Barendregt 1991), uses a shared syntax for terms and types. The language includes

[5]In fact, our Coq development uses the same syntax both languages, and relies on the judgment forms to identify the pertinent set of constructs.

- a single sort ($\star$) for classifying types,
- functions ($\lambda^+ x.a$) with dependent types ($\Pi^+ x\!:\!A \to B$), and their associated application forms ($a\ b^+$),
- functions with irrelevant arguments ($\lambda^- x.a$), their types ($\Pi^- x\!:\!A \to B$), and instantiations ($a\ \square^-$),
- coercion abstractions ($\Lambda c.a$), their types ($\forall c\!:\!\phi.B$), and instantiations ($a[\bullet]$),
- 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 $\rho$. We sometimes omit relevance annotations $\rho$ from applications $a\ b^\rho$ when they are clear from context. We also write nondependent types $\Pi^+ x\!:\!A \to B$ 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 $A$.

## 3.1 Evaluation

The call-by-name small-step evaluation rules for D are shown below. The first three rules are primitive reductions—if a term steps using one of these first three rules only, then we use the notation $\vDash a > b$. The primitive reductions include call-by-name $\beta$-reduction of abstractions, $\beta$-reduction of coercion abstractions, and unfolding of top-level definitions.

$$\textsc{E-AppAbs}\ \frac{}{\vDash (\lambda^\rho x.b)\ a^\rho \leadsto b\{a/x\}} \qquad \textsc{E-CAppCAbs}\ \frac{}{\vDash (\Lambda c.b)[\gamma] \leadsto b\{\gamma/c\}} \qquad \textsc{E-Axiom}\ \frac{F \sim a : A \in \Sigma_0}{\vDash F \leadsto a}$$

$$\textsc{E-AbsTerm}\ \frac{\vDash a \leadsto a'}{\vDash \lambda^- x.a \leadsto \lambda^- x.a'} \qquad \textsc{E-AppLeft}\ \frac{\vDash a \leadsto a'}{\vDash a\ b^\rho \leadsto a'\ b^\rho} \qquad \textsc{E-CAppLeft}\ \frac{\vDash a \leadsto a'}{\vDash a[\gamma] \leadsto a'[\gamma]}$$

The second three rules extend primitive reduction into a deterministic reduction relation, called *one-step reduction*, and written $\vDash a \leadsto b$. When iterated, this relation models the operational semantics of core Haskell by reducing expressions to their weak-head form.

The only unusual rule of this relation is rule E-AbsTerm that allows reduction to continue underneath an irrelevant abstraction. (Analogously, an implicit abstraction is a value *only* when its body is also a value.) This rule means that D models the behavior of source Haskell when it comes to polymorphism—type generalization via implicit abstraction does not delay computation and so has no computational effect. This rule compensates for the fact that we do not erase implicit generalizations and instantiations completely in D; although the arguments are not present, the locations are still marked in the term. We choose this design to simplify the metatheory of D, as we discuss further in Section 6.2.

## 3.2 Typing

The typing rules, shown in Figure 3, are based on a dependent type theory with $\star : \star$, as shown in the first rule in the figure (rule E-Star). Although this rule is known to violate logical consistency, it is not problematic in this context. Haskell already has unbound recursion (both at runtime and at compile-time), and thus is already logically inconsistent. Therefore, we avoid the complications that come from the stratified universe hierarchy needed to ensure termination in dependently-typed languages.

The next five rules describe relevant and irrelevant abstractions. D includes irrelevant abstractions to support *parametric* polymorphism—irrelevant arguments are not present in the term. Lambda expressions (and their

$$\boxed{\Gamma \vDash a : A}$$

$$\frac{\vDash \Gamma}{\Gamma \vDash \star : \star}\ \textsc{E-Star} \qquad \frac{\vDash \Gamma \qquad x : A \in \Gamma}{\Gamma \vDash x : A}\ \textsc{E-Var} \qquad \frac{\Gamma, x : A \vDash B : \star}{\Gamma \vDash \Pi^\rho x{:}A \to B : \star}\ \textsc{E-Pi} \qquad \frac{\Gamma, x : A \vDash a : B \qquad \rho \vee x \notin a}{\Gamma \vDash \lambda^\rho x.a : \Pi^\rho x{:}A \to B}\ \textsc{E-Abs}$$

$$\frac{\Gamma \vDash b : \Pi^+ x{:}A \to B \qquad \Gamma \vDash a : A}{\Gamma \vDash b\ a^+ : B\{a/x\}}\ \textsc{E-App} \qquad \frac{\Gamma \vDash b : \Pi^- x{:}A \to B \qquad \Gamma \vDash a : A}{\Gamma \vDash b\ \square^- : B\{a/x\}}\ \textsc{E-IApp} \qquad \frac{\Gamma \vDash a : A \qquad \Gamma; \widetilde{\Gamma} \vDash A \equiv B : \star}{\Gamma \vDash a : B}\ \textsc{E-Conv} \qquad \frac{\vDash \Gamma \qquad F \sim a : A \in \Sigma_0}{\Gamma \vDash F : A}\ \textsc{E-Fam}$$

$$\frac{\Gamma, c : \phi \vDash B : \star}{\Gamma \vDash \forall c{:}\phi.B : \star}\ \textsc{E-CPi} \qquad \frac{\Gamma, c : \phi \vDash a : B}{\Gamma \vDash \Lambda c.a : \forall c{:}\phi.B}\ \textsc{E-CAbs} \qquad \frac{\Gamma \vDash a_1 : \forall c{:}(a \sim_A b).B_1 \qquad \Gamma; \widetilde{\Gamma} \vDash a \equiv b : A}{\Gamma \vDash a_1[\bullet] : B_1\{\bullet/c\}}\ \textsc{E-CApp}$$

$$\boxed{\Gamma \vDash \phi\ \mathsf{ok}} \qquad \frac{\Gamma \vDash a : A \qquad \Gamma \vDash b : A}{\Gamma \vDash a \sim_A b\ \mathsf{ok}}\ \textsc{E-Wff}$$

$$\boxed{\rho \vee x \notin A} \qquad \frac{}{+ \vee x \notin A}\ \textsc{Rho-Rel} \qquad \frac{x \notin \mathsf{fv} A}{- \vee x \notin A}\ \textsc{Rho-IrrRel}$$

$$\boxed{\vDash \Sigma}$$

$$\frac{}{\vDash \varnothing}\ \textsc{Sig-Empty} \qquad \frac{\vDash \Sigma \qquad F \sim a : A \in \Sigma_0 \qquad \varnothing \vDash A : \star \qquad \varnothing \vDash a : A \qquad F \notin \mathsf{dom}\,\Sigma}{\vDash \Sigma \cup \{F \sim a : A\}}\ \textsc{Sig-ConsAx}$$

Fig. 3. D Type system

types) are marked by a relevance flag, $\rho$, indicating whether the type-or-term argument may be used in the body of the abstraction (+) or must be parametric (−). This usage is checked by the premise $\rho \vee x \notin a$. If the argument must be parametric, then it cannot appear anywhere in the body of the expression. This approach to irrelevant abstractions is directly inspired by ICC (Miquel 2001). Irrelevant applications mark missing arguments with $\square$. This is the only place that the typing rules allow the $\square$ term.

The next rule, rule E-Conv, is conversion. This type system assigns types up to definitional equality, defined by the judgment $\Gamma; \Delta \vDash a \equiv b : A$ shown in Figure 4. This judgment is indexed by $\Delta$, a set of *available variables*. For technical reasons that we return to in Section 4.2, we must restrict the coercion assumptions that are available in proofs of definitional equality to those in this set. When definitional equality is used in the typing judgment, as it is in rule E-Conv, all in-scope coercion variables are available. We use the notation $\widetilde{\Gamma}$ for the set of all coercion variables in the domain of $\Gamma$.

This language is parameterized by a set of *recursive definitions F*, specified by a toplevel signature. We assume that there exists some toplevel signature $\Sigma_0$ that is well-formed according to the rules presented at the bottom of Figure 3. In particular, note that rule Sig-ConsAx refers to $\Sigma_0$ in its second premise, not the smaller $\Sigma$. The other typing premises—like all other typing derivations—are implicitly parameterized with respect to this signature $\Sigma_0$, as long as it is well-formed.

For example, we can declare a standard, polymorphic recursive fixpoint operator **Fix** in the signature as

$$\vDash \mathbf{Fix} \sim \lambda^- x.\lambda^+ y.(y\,(\mathbf{Fix}\,\square\,y)) : \Pi^- x{:}\star \to (x \to x) \to x$$

and show that it is well-formed. Because D is a full-spectrum language, **Fix** can be used to define recursive functions and recursive datatypes. Alternatively, recursive definitions can be directly defined as part of the top-level signature. Because there is no inherent ordering on signatures, recursive definitions may be mutually defined and may be induction-recursive (Dybjer and Setzer 1999).

To support GADTs this language includes *coercion abstractions*, written $\Lambda c.a$. This term provides the ability for an expression $a$ to be parameterized over a type equality assumption $c$, which is evidence proving an equality proposition $\phi$. The assumed equality is stored in the context during type checking and is made available to definitional equality. In D, coercion assumptions are discharged in rule E-CApp by $\bullet$, a trivial proof that marks the provability of an assumed equality.

Propositions $\phi$, written $a \sim_A b$, are statements of equality between terms/types. The two terms $a$ and $b$ must have the same type $A$ for this statement to be well-formed, as shown in rule E-Wff. In other words, equality propositions are homogeneous. We cannot talk about equality between terms unless we know that their types are equal.

## 3.3 Definitional equality

The most delicate part in the design of a dependently-typed language is the definition of the equality used in the conversion rule. This relation, $\Gamma; \Delta \vDash a \equiv b : A$ defines when two terms $a$ and $b$ are indistinguishable. The rules in Figure 4 define this relation for D.

As in most dependently-typed languages, this definition of equality is an equivalence relation (see the first three rules of the figure) and a congruence relation (see all rules ending with Cong). Similarly, equality contains the reduction relation (rule E-Beta). Because evaluation may not terminate, this definition of equality is not a decidable relation.

Furthermore, this relation is (homogeneously) typed—two terms $a$ and $b$ are related at a particular type $A$ (and at all types equal to $A$, via rule E-EqConv). In other words, this system has the following property:

Lemma 3.1 (DefEq regularity). *If* $\Gamma; \Delta \vDash a \equiv b : A$ *then* $\Gamma \vDash a : A$ *and* $\Gamma \vDash b : A$.

So far, these rules are similar to most judgmental treatments of definitional equality in intensional type theory, such as that shown in Aspinall and Hoffman (2005). However, this definition differs from that used in most other dependently-typed languages through the inclusion of the rule E-Assn. This rule says that assumed propositions can be used directly, as long as they are in the available set.

The assumption rule strengthens this definition of equality considerably compared to intensional type theory. Indeed, it reflects the equality propositions into the definitional equality, as in extensional type theory (Martin-Löf 1984). However, D should not be considered an extensional type theory because our equality propositions are not the same as "propositional equality" found in other type theories—equality propositions are kept separate from types. Coercion abstraction is not the same as normal abstraction, and can only be justified by equality derivations, not by arbitrary terms. This means that all equality assumptions must eventually be justified by some derivation of definitional equality, not by computation. Because we cannot use a term to justify an assumed equality, this language remains type sound in the presence of nontermination.

## 3.4 Equality propositions are not types

Our languages firmly distinguish between types (which are all inhabited by terms) and equality propositions (which may or may not be provable using the rules in Figure 4). Propositions are checked for well-formedness with the judgment $\Gamma \vDash \phi$ ok (Figure 3). However, because propositions appear *in* types, we also need to define when two propositions are equal. We do so with the judgment $\Gamma; \Delta \vDash \phi_1 \equiv \phi_2$ at the bottom of (Figure 4) and call this relation *prop equality*.

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

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

$$\textsc{E-PiCong}\ \frac{\Gamma;\Delta \vDash A_1 \equiv A_2 : \star \quad \Gamma, x : A_1;\Delta \vDash B_1 \equiv B_2 : \star}{\Gamma;\Delta \vDash (\Pi^\rho x{:}A_1 \to B_1) \equiv (\Pi^\rho x{:}A_2 \to B_2) : \star}$$

$$\textsc{E-AbsCong}\ \frac{\Gamma, x : A_1;\Delta \vDash b_1 \equiv b_2 : B \quad \rho \vee x \notin b_1 \quad \rho \vee x \notin b_2}{\Gamma;\Delta \vDash (\lambda^\rho x.b_1) \equiv (\lambda^\rho x.b_2) : \Pi^\rho x{:}A_1 \to B}$$

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

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

$$\textsc{E-CPiCong}\ \frac{\Gamma;\Delta \vDash \phi_1 \equiv \phi_2 \quad \Gamma, c : \phi_1;\Delta \vDash A \equiv B : \star}{\Gamma;\Delta \vDash \forall c{:}\phi_1.A \equiv \forall c{:}\phi_2.B : \star}$$

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

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

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

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

$$\textsc{E-PiFst}\ \frac{\Gamma;\Delta \vDash \Pi^\rho x{:}A_1 \to B_1 \equiv \Pi^\rho x{:}A_2 \to B_2 : \star}{\Gamma;\Delta \vDash A_1 \equiv A_2 : \star}$$

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

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

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

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

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

$$\textsc{E-EqConv}\ \frac{\Gamma;\Delta \vDash a \equiv b : A \quad \Gamma;\widetilde{\Gamma} \vDash A \equiv B : \star}{\Gamma;\Delta \vDash a \equiv b : B}$$

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

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

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

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

Fig. 4. Definitional equality for implicit language

We use prop equality in two places in the definition of term/type equality. Prop equality is necessary in the congruence rule for coercion types (rule E-CPiCong). It also may be used to change the conclusion of the definitional equality judgment to an equivalent equality proposition (rule E-Cast).

Two propositions are equal when their corresponding terms are equal (rule E-PropCong) or when their corresponding types are equal (rule E-IsoConv). Furthermore, if two coercion abstraction types are equivalent then the injectivity of these types means that we can extract an equivalence of the propositions (rule E-CPiFst). Although the type system does not explicitly include rules for reflexivity, symmetry and transitivity, these operations are derivable from the analogous rules for definitional equality and rule E-CPiFst.

One difference between term/type and prop equality is that type forms are injective everywhere (see rules E-PiFst and E-CPiFst for example) but the constructor $\sim$ is injective in only the types of the equated terms, not in the two terms themselves. For example, if we have a prop equality $a_1 \sim_A a_2 \equiv b_1 \sim_B b_2$, we can derive $A \equiv B : \star$, using rule E-EqConv, but we cannot derive $a_1 \equiv b_1 : A$ or $a_2 \equiv b_2 : A$.

Prior work includes this sort of injectivity by default, but we introduce prop equality separate from type equality specifically so that we can leave it out of the language definition. The reason for this omission is twofold. First, unlike rule E-PiFst, for example, this injectivity not forced by the rest of the system. In contrast, the preservation theorem requires rule E-PiFst, as we describe below. Second, this omission leaves the system open for a more extensional definition of prop equality, which we hope to explore in future work (see Section 9).

## 4 TYPE SOUNDNESS FOR SYSTEM D

The previous section completely specifies the operational and static semantics of the D language. Next, we turn to its metatheoretic properties. In this section, we show that the language is type sound by proving the usual preservation and progress lemmas. Note that although we are working with a dependent type system, the structure of the proof below directly follows related results about FC (Breitner et al. 2014; Weirich et al. 2013). In particular, because this language (like $F_\omega$) has a nontrivial definitional equality, we must show that this equality is consistent before proving the progress lemma. We view the fact that current proof techniques extend to this full spectrum language as a positive feature of this design—the adoption of dependent types has not forced us to abandon existing methods for reasoning about the language. The contributions of this paper are in the design of the system itself, not in structure of the proof of its type soundness property. Therefore, we do not describe these arguments in great detail below.

### 4.1 Preservation and parallel reduction

We have defined three different reduction relations for the implicit language. Two we have seen already: primitive reduction and one-step reduction. The third relation, which subsumes the other two, is parallel reduction written $\vDash a \Rightarrow b$. We define this relation in support of our proof technique only; it is not part of the specification of D. For reasons of space, this relation appears only in Appendix C.3. This relation is a strongly confluent, but not necessarily terminating, rewrite relation on terms.

Theorem 4.1 (Confluence). *If $\vDash a \Rightarrow a_1$ and $\vDash a \Rightarrow a_2$ then there exists $b$, such that $\vDash a_1 \Rightarrow b$ and $\vDash a_2 \Rightarrow b$.*

We prove the preservation theorem for parallel reduction only, as it contains the other two reduction relations.

Theorem 4.2 (Preservation). *If $\Gamma \vDash a : A$ and $\vDash a \Rightarrow a'$ then $\Gamma \vDash a' : A$.*

Simultaneously, we must prove that parallel reduction is also contained in definitional equality.

Lemma 4.3 (ParDefEq). *If $\vDash a \Rightarrow a'$ and $\Gamma \vDash a : A$ then $\Gamma; \varnothing \vDash a \equiv a' : A$.*

The proofs of these theorems are straightforward, but do require several inversion lemmas for the typing relation. Because of implicit conversion (rule E-Conv), inversion of the typing judgment produces types that are definitionally equal but not syntactically equal to the given type. For example, the inversion rule for term abstractions reads

LEMMA 4.4 (INVERSION FOR ABSTRACTION). *If* $\Gamma \vDash \lambda^{\rho} x\!:\!A_0.b_0 : A$ *then there exists some* $A_1$ *and* $B_1$ *such that* $\Gamma; \widetilde{\Gamma} \vDash A \equiv \Pi^{\rho} x\!:\!A_1 \to B_1 : \star$ *and* $\Gamma, x : A_1 \vdash b_0 : B_1$ *and* $\Gamma, x : A_1 \vdash B_1 : \star$ *and* $\Gamma \vdash A_1 : \star$.

As a result of this inversion lemma, the case for rule E-BETA in the preservation proof requires injectivity for function types (rules E-PIFST and E-PISND) in definitional equality. Similarly, rule E-CBETA requires rules E-CPIFST and E-CPISND.

## 4.2 Progress and consistency

An important step for the proof of the progress lemma is to show the consistency of definitional equality. Consistency means that in certain contexts, the system cannot derive an equality between types that have different head forms. We write "**consistent** $A\,B$" when $A$ and $B$ are consistent—i.e. when it is not the case that they are types with conflicting heads.

We show consistency in two steps, using an auxiliary relation called *joinability*. Two types are joinable when they reduce to some common term using any number of steps of parallel reduction. Our consistency proof thus first shows that joinable types are consistent and then that definitionally equal types are joinable.

DEFINITION 4.5 (JOINABLE). *Two types are joinable, written* $\vdash a_1 \Leftrightarrow a_2$, *when there exists some* $b$ *such that* $\vdash a_1 \Rightarrow^* b$ *and* $\vdash a_2 \Rightarrow^* b$.

THEOREM 4.6 (JOINABILITY IMPLIES CONSISTENCY). *If* $\vdash A \Leftrightarrow B$ *then* **consistent** $A\,B$.

Only *some* definitionally equal types are joinable. Because our consistency proof is based on parallel reduction, and because parallel reduction ignores assumed equality propositions, we need to restrict the propositions that can be used in equality derivations. In particular, all available coercion assumptions must be between types that are already joinable. When a context and available set satisfy this restriction, we call it *Good*.

DEFINITION 4.7 (GOOD). *We write* **Good** $\Gamma\,\Delta$ *when* $c : a \sim_A b \in \Gamma$ *and* $c \in \Delta$ *implies* $\vdash a \Leftrightarrow b$.

THEOREM 4.8 (EQUALITY IMPLIES JOINABILITY). *If* **Good** $\Gamma\,\Delta$ *and* $\Gamma; \Delta \vDash a \equiv b : A$ *then* $\vdash a \Leftrightarrow b$

The **Good** $\Gamma\,\Delta$ property is required because the type system places no restrictions on propositions in coercion abstractions. It does not rule out a clearly bogus assumption, such as $\mathbf{Int} \sim_\star \mathbf{Bool}$. As a result, we cannot prove that only consistent types are definitionally equal in a context that includes such an assumption.

A consequence of our joinability-based proof of consistency is that there are some equalities that may be safe but we cannot allow the type system to derive. For example, we cannot allow the congruence rule for coercion abstraction types (rule E-CPICONG) to derive this equality.

$$\varnothing; \varnothing \vDash \forall c\!:\!(\mathbf{Int} \sim_\star \mathbf{Bool}).\mathbf{Int} \equiv \forall c\!:\!(\mathbf{Int} \sim_\star \mathbf{Bool}).\mathbf{Bool} : \star$$

The problem is that we don't know how to show that this equality consistent—these two terms are not joinable.

We prevent rule E-CPICONG from deriving this equality by *not* adding the assumption `c` to the available set $\Delta$ when showing the equality for `Int` and `Bool`. The rest of the rules preserve this restriction in the parts of the derivation that are necessary to show terms equivalent. Note that we can sometimes weaken the restriction in derivations: For example in rule E-CAPPCONG, the premise that shows $a \equiv b$ is to make sure that the terms $a_1[\bullet]$ and $b_1[\bullet]$ type check. It is not part of the equality proof, so we can use the full context at that point.

One may worry that with this restriction, our definitional equality might not admit the substitutivity property stated below.[6] This lemma states that in any context (i.e. a term with a free variable) we can lift an equality through that context.

[6]This lemma is called the "lifting lemma" in prior work (Sulzmann et al. 2007; Weirich et al. 2013).

$$\begin{array}{llcl}
\textit{terms, types} & a, b, A, B & ::= & \star \mid x \mid F \mid \lambda^{\rho} x\!:\!A.b \mid a\ b^{\rho} \mid \Pi^{\rho} x\!:\!A \to B \\
& & \mid & \Lambda c\!:\!\phi.a \mid a[\gamma] \mid \forall c\!:\!\phi.A \mid a \triangleright \gamma \\
\textit{coercions (excerpt)} & \gamma & ::= & c \mid \mathbf{refl}\ a \mid \mathbf{refl}\ (a \sim_{\gamma} b) \mid \mathbf{sym}\ \gamma \mid \gamma_1;\gamma_2 \mid \mathbf{red}\ a\ b \mid \Pi^{\rho} x\!:\!\gamma_1.\gamma_2 \mid \ldots
\end{array}$$

Fig. 5. Syntax of DC, the explicit language

Lemma 4.9 (Substitutivity). *If* $\Gamma_1, x : A, \Gamma_2 \vDash b : B$ *and* $\Gamma_1; \Delta \vDash a_1 \equiv a_2 : A$ *then* $\Gamma_1, (\Gamma_2\{a_1/x\}); \Delta \vDash b\{a_1/x\} \equiv b\{a_2/x\} : B\{a_1/x\}$.

Indeed, Eisenberg (2016) could not prove this property about his language because his treatment of coercion variables in the rule was too restrictive. However, this result is provable in our system because our restriction via available sets precisely characterizes what it means to "use" a coercion variable.

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

Lemma 4.10 (Progress). *If* $\Gamma \vDash a : A$ *and* $\mathbf{Good}\ \Gamma\ \widetilde{\Gamma}$ *and no term variable x in the domain of* $\Gamma$ *occurs free in a, then either a is a value or there exists some* $a'$ *such that* $\vDash a \leadsto a'$.

## 5 SYSTEM DC: AN EXPLICITLY-TYPED LANGUAGE

We now turn to the explicit language, DC, which adds syntactic forms for type annotations and explicit coercions to make type checking unique and decidable. The syntax of DC is shown in Figure 5. The new syntactic form $a \triangleright \gamma$ marks type coercions with explicit proofs $\gamma$. Furthermore, the syntax also includes the types of variables in term and coercion abstractions ($\lambda x\!:\!A.a$ and $\Lambda c\!:\!\phi.a$). To require explicit terms in instantiations, the term ($\square$) and the trivial coercion ($\bullet$) are missing from this syntax.

The main judgment forms of this language correspond exactly to the implicit language judgments, as shown in Figure 1.

We can connect DC terms to D terms through an erasure operation, written $|a|$, that translates annotated terms to their implicit counterparts. This definition is a structural recursion over the syntax, removing irrelevant information.

Definition 5.1 (Annotation erasure).

$$\begin{array}{ll}
|\star| = \star & |a\ b^{-}| = |a|\ \square^{-} \\
|x| = x & |\Pi^{\rho} x\!:\!A \to B| = \Pi^{\rho} x\!:\!|A| \to |B| \\
|F| = F & |\Lambda c\!:\!\phi.a| = \Lambda c.|a| \\
|\lambda^{\rho} x\!:\!A.a| = \lambda^{\rho} x.|a| & |a[\gamma]| = |a|[\bullet] \\
|a\ b^{+}| = |a|\ |b|^{+} & |\forall c\!:\!a_0 \sim_A a_1.b| = \forall c\!:\!|a_0| \sim_{|A|} |a_1|.|b| \\
& |a \triangleright \gamma| = a
\end{array}$$

We start our discussion by summarizing the properties that guide the design of DC and its connection to D. For brevity, we state these properties only about the typing judgment below, but analogues hold for the first six judgment forms shown in Figure 1.

First, typing is decidable in DC, and annotations nail down all sources of ambiguity in the typing relation, making type-checking fully syntax directed.

Lemma 5.2 (Decidable typing). *Given* $\Gamma$ *and a, it is decidable whether there exists some A such that* $\Gamma \vdash a : A$.

Lemma 5.3 (Uniqueness of typing). *If* $\Gamma \vdash a : A_1$ *and* $\Gamma \vdash a : A_2$ *then* $A_1 = A_2$.

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

$$\textsc{An-Star}\ \frac{\vdash \Gamma}{\Gamma \vdash \star : \star} \qquad \textsc{An-Var}\ \frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{An-Pi}\ \frac{\Gamma, x : A \vdash B : \star}{\Gamma \vdash \Pi^{\rho} x{:}A \to B : \star} \qquad \textsc{An-Abs}\ \frac{\Gamma, x : A \vdash a : B \qquad \rho \vee x \notin |a|}{\Gamma \vdash \lambda^{\rho} x{:}A.a : \Pi^{\rho} x{:}A \to B}$$

$$\textsc{An-App}\ \frac{\begin{array}{c}\Gamma \vdash b : \Pi^{\rho} x{:}A \to B \\ \Gamma \vdash a : A\end{array}}{\Gamma \vdash b\ a^{\rho} : B\{a/x\}} \qquad \textsc{An-Conv}\ \frac{\begin{array}{c}\Gamma \vdash a : A \\ \Gamma; \widetilde{\Gamma} \vdash \gamma : A \sim B \qquad \Gamma \vdash B : \star\end{array}}{\Gamma \vdash a \triangleright \gamma : B} \qquad \textsc{An-Fam}\ \frac{\vdash \Gamma \qquad F \sim a : A \in \Sigma_1}{\Gamma \vdash F : A} \qquad \textsc{An-CPi}\ \frac{\Gamma, c : \phi \vdash B : \star}{\Gamma \vdash \forall c{:}\phi.B : \star}$$

$$\textsc{An-CAbs}\ \frac{\Gamma, c : \phi \vdash a : B}{\Gamma \vdash \Lambda c{:}\phi.a : \forall c{:}\phi.B} \qquad \textsc{An-CApp}\ \frac{\Gamma \vdash a_1 : \forall c{:}a \sim_{A_1} b.B \qquad \Gamma; \widetilde{\Gamma} \vdash \gamma : a \sim b}{\Gamma \vdash a_1[\gamma] : B\{\gamma/c\}}$$

Fig. 6. Typing rules for DC

Next, the two languages are strongly related via this erasure operation, in the following way. We can always erase DC typing derivations to produce D derivations. Furthermore, given D derivations we can always produce annotated terms and derivations in DC that erase to them.

Lemma 5.4 (Erasure). *If* $\Gamma \vdash a : A$ *then* $|\Gamma| \vDash |a| : |A|$.

Lemma 5.5 (Annotation). *If* $\Gamma \vDash a : A$ *then, for all* $\Gamma_0$ *such that* $|\Gamma_0| = \Gamma$*, there exists some* $a_0$ *and* $A_0$*, such that* $\Gamma_0 \vdash a_0 : A_0$ *where* $|a_0| = a$ *and* $|A_0| = A$.

## 5.1 The design of DC

Designing a language that has decidable type checking, unique types, and corresponds exactly to D requires the addition of a number of annotations to the syntax of D, reifying the information contained in the typing derivation. In this section, we discuss some of the constraints on our designs and their effects on the rules for typing terms (Figure 6) and checking coercion proofs (Figures 7 and 8). Overall, the typing rules for DC are no more complex than their D counterparts. However, the rules for coercions require much more bookkeeping in DC than the corresponding rules in D. For example, compare rule E-CAbsCong with rule An-CAbsCong.

The most important change for the explicit language is the addition of explicit coercion proof terms for type conversion in rule An-Conv. Because the definitional equality relation is undecidable, we cannot ask the type checker to determine whether two types are equal in a conversion. Instead, this language includes an explicit proof $\gamma$ of the equality that the DC type checker is only required to verify. We describe this judgment in the next subsection.

Other rules of the type system also add annotations to make type checking syntax directed. For example, consider the typing rules for abstractions (rule An-Abs) and applications (rule An-App). We have two new annotations in these two rules. Abstractions include the types of bound variables and irrelevant applications use their actual arguments instead of using $\square$. As a positive result of this change, we need only one rule for typing applications, not two as applications must always include their arguments, even when those arguments are irrelevant. Furthermore, because terms now include irrelevant variables in these annotations, irrelevant abstractions check relevance against the body *after erasure*, following ICC* (Barras and Bernardo 2008).

Similarly, coercion abstraction and instantiation require annotations for the abstracted proposition and the evidence that the equality is satisfied. See rules An-CAbs and An-CApp. All other rules of the typing judgment are the same as D.

### 5.2 Explicit equality proofs

Figure 5 includes some of the syntax of the coercion proof terms that are available in DC. The syntax figure does not include all of the coercions because their syntax makes little sense out of the context of the rules that check them. Indeed, these proof terms merely record information found in the rules of the analogous D judgments for type and prop equality. In other words, $\gamma$ in the coercion judgment $\Gamma; \Delta \vdash \gamma : a \sim b$ records the information contained in a derivation of a type equality $\Gamma; \Delta \vDash a \equiv b : A$.

However, there is some flexibility in the design of the judgment $\Gamma; \Delta \vdash \gamma : a \sim b$. First, observe that the syntax of the judgment does not include a component that corresponds to $A$, the type of $a$ and $b$ in the implicit system. We do not include this type because it is unnecessary. In DC, $a$ and $b$ have unique types. If we ever need to know what their types are, we can always recover them directly from the context and the terms. (This choice mirrors the current implementation of GHC.)

Furthermore, we have flexibility in the relationship between the types of the terms in the coercion judgment. Suppose we have $\Gamma; \Delta \vdash \gamma : a \sim b$ and $\Gamma \vdash a : A$ and $\Gamma \vdash b : B$. Then there are three possible ways we could have designed the judgment with respect to the types $A$ and $B$. The judgment could require

(1) that $A = B$, i.e. that the types must be $\alpha$−equivalent, or
(2) that $|A| = |B|$, i.e. that the types must be equal up to erasure, or
(3) that there must exist some coercion $\Gamma; \Delta \vdash \gamma_0 : A \sim B$ that relates them.

There is also a fourth option—not enforcing any relationship between $A$ and $B$ to hold. While previous work (Eisenberg 2016; Gundry 2013; Weirich et al. 2013), allowed such heterogeneous equality, we cannot do so here and still connect to the homogeneous equality of D.

At first glance, the first option might seem the closest to D. After all, in that language, the two terms must type check with exactly the same type. However, given that D includes implicit coercion, that choice is overly restrictive—the two terms will also type check with definitionally equal types too. The second option relaxes that restriction, but the third is the closest to D.

Therefore, our system admits the following property of the coercion judgment.

Lemma 5.6 (Coercion regularity). *If* $\Gamma; \Delta \vdash \gamma : a \sim b$ *then there exists some* $A$, $B$ *and* $\gamma_0$*, such that* $\Gamma \vdash a : A$ *and* $\Gamma \vdash b : B$ *and* $\Gamma; \widetilde{\Gamma} \vdash \gamma_0 : A \sim B$.

Furthermore, allowing the two terms to have provably equal types leads to more compositional rules than the first two options. For example, consider the application congruence rule, rule An-AppCong. Due to dependency, the types of the two terms in the conclusion of this rule may not be $\alpha$-equivalent. If we had chosen the first option above, we would have to use the rule below instead, which includes a coercion around one of the terms to make their types line up. In DC, that coercion is unnecessary and the rule is symmetric.

AltAn-AppCongEq
$$\frac{\Gamma; \Delta \vdash \gamma_1 : a_1 \sim b_1 \qquad \Gamma; \Delta \vdash \gamma_2 : a_2 \sim b_2 \qquad \Gamma \vdash a_1\ {a_2}^\rho : A \qquad \Gamma \vdash b_1\ {b_2}^\rho : B \qquad \Gamma; \Delta \vdash \gamma : B \sim A}{\Gamma; \Delta \vdash (\gamma_1\ \gamma_2^\rho \triangleright \gamma) : (a_1\ {a_2}^\rho) \sim (b_1\ {b_2}^\rho \triangleright \gamma)}$$

On the other hand, we follow Eisenberg (2016) and allow *some* asymmetry in the congruence rules for syntactic forms with binders. For example, consider rule An-PiCong for showing two Π-types equal via congruence (this rule is analogous to rule E-PiCong of the implicit system). Note the asymmetry—the rule requires that the bodies of the Π-types be shown equivalent using a single variable $x$ of type $A_1$. However, in the conclusion, we would

$\boxed{\Gamma;\Delta \vdash \gamma : a \sim b}$ *(Type equality)*

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

$$\frac{\Gamma \vdash b : B \qquad \Gamma \vdash a : A \qquad \Gamma;\Delta \vdash \gamma : b \sim a}{\Gamma;\Delta \vdash \mathbf{sym}\,\gamma : a \sim b}\ \textsc{An-Sym}$$

$$\frac{\Gamma;\Delta \vdash \gamma_1 : a \sim a_1 \qquad \Gamma;\Delta \vdash \gamma_2 : a_1 \sim b}{\Gamma;\Delta \vdash (\gamma_1;\gamma_2) : a \sim b}\ \textsc{An-Trans}$$

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

$$\frac{\begin{array}{c}\Gamma;\Delta \vdash \gamma_1 : A_1 \sim A_2 \qquad \Gamma, x : A_1;\Delta \vdash \gamma_2 : b_1 \sim b_2\\ b_3 = b_2\{x \triangleright \mathbf{sym}\,\gamma_1/x\}\\ \Gamma \vdash A_2 : \star \qquad \rho \vee x \notin |b_1| \qquad \rho \vee x \notin |b_3|\end{array}}{\Gamma;\Delta \vdash (\lambda^\rho x\!:\!\gamma_1.\gamma_2) : (\lambda^\rho x\!:\!A_1.b_1) \sim (\lambda^\rho x\!:\!A_2.b_3)}\ \textsc{An-AbsCong}$$

$$\frac{\begin{array}{c}\Gamma;\Delta \vdash \gamma_1 : a_1 \sim b_1 \qquad \Gamma;\Delta \vdash \gamma_2 : a_2 \sim b_2\\ \Gamma \vdash a_1\ {a_2}^\rho : A \qquad \Gamma \vdash b_1\ {b_2}^\rho : B\end{array}}{\Gamma;\Delta \vdash \gamma_1\ \gamma_2^\rho : a_1\ {a_2}^\rho \sim b_1\ {b_2}^\rho}\ \textsc{An-AppCong}$$

$$\frac{\begin{array}{c}\Gamma;\Delta \vdash \gamma_1 : \phi_1 \sim \phi_2 \qquad \Gamma, c : \phi_1;\Delta \vdash \gamma_3 : B_1 \sim B_2\\ B_3 = B_2\{c \triangleright \mathbf{sym}\,\gamma_1/c\}\\ \Gamma \vdash \forall c\!:\!\phi_1.B_1 : \star \qquad \Gamma \vdash \forall c\!:\!\phi_2.B_3 : \star\end{array}}{\Gamma;\Delta \vdash (\forall c\!:\!\gamma_1.\gamma_3) : (\forall c\!:\!\phi_1.B_1) \sim (\forall c\!:\!\phi_2.B_3)}\ \textsc{An-CPiCong}$$

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

$$\frac{\begin{array}{c}\Gamma;\Delta \vdash \gamma_1 : a_1 \sim b_1\\ \Gamma;\widetilde{\Gamma} \vdash \gamma_2 : a_2 \sim b_2\\ \Gamma;\widetilde{\Gamma} \vdash \gamma_3 : a_3 \sim b_3\\ \Gamma \vdash a_1[\gamma_2] : A \qquad \Gamma \vdash b_1[\gamma_3] : B\end{array}}{\Gamma;\Delta \vdash \gamma_1(\gamma_2,\gamma_3) : a_1[\gamma_2] \sim b_1[\gamma_3]}\ \textsc{An-CAppCong}$$

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

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

$$\frac{\Gamma;\Delta \vdash \gamma : \Pi^\rho x\!:\!A_1 \to B_1 \sim \Pi^\rho x\!:\!A_2 \to B_2}{\Gamma;\Delta \vdash \mathbf{piFst}\,\gamma : A_1 \sim A_2}\ \textsc{An-PiFst}$$

$$\frac{\begin{array}{c}\Gamma;\Delta \vdash \gamma_1 : \Pi^\rho x\!:\!A_1 \to B_1 \sim \Pi^\rho x\!:\!A_2 \to B_2\\ \Gamma;\Delta \vdash \gamma_2 : a_1 \sim a_2 \qquad \Gamma \vdash a_1 : A_1 \qquad \Gamma \vdash a_2 : A_2\end{array}}{\Gamma;\Delta \vdash \gamma_1@\gamma_2 : B_1\{a_1/x\} \sim B_2\{a_2/x\}}\ \textsc{An-PiSnd}$$

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

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

$$\frac{\begin{array}{c}\Gamma;\Delta \vdash \gamma_1 : a \sim a'\\ \Gamma;\Delta \vdash \gamma_2 : a \sim_A a' \sim b \sim_B b'\end{array}}{\Gamma;\Delta \vdash \gamma_1 \triangleright \gamma_2 : b \sim b'}\ \textsc{An-Cast}$$

$$\frac{\Gamma;\Delta \vdash \gamma : (a \sim_A a') \sim (b \sim_B b')}{\Gamma;\Delta \vdash \mathbf{typeInj}\,\gamma : A \sim B}\ \textsc{An-IsoSnd}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B \qquad |a| = |b| \qquad \Gamma;\widetilde{\Gamma} \vdash \gamma : A \sim B}{\Gamma;\Delta \vdash \mathbf{refl}\,(a \sim_\gamma b) : a \sim b}\ \textsc{An-Refl2}$$

Fig. 7. Type equality for DC

$\boxed{\Gamma; \Delta \vdash \gamma : \phi_1 \sim \phi_2}$ *(Prop equality)*

An-PropCong
$$\frac{\Gamma; \Delta \vdash \gamma_1 : A_1 \sim A_2 \qquad \Gamma; \Delta \vdash \gamma_2 : B_1 \sim B_2 \qquad \Gamma \vdash A_1 \sim_A B_1 \ \mathsf{ok} \qquad \Gamma \vdash A_2 \sim_A B_2 \ \mathsf{ok}}{\Gamma; \Delta \vdash (\gamma_1 \sim_A \gamma_2) : (A_1 \sim_A B_1) \sim (A_2 \sim_A B_2)}$$

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

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

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

Fig. 8. Prop equality for DC

like to create a proof of equivalence for a Π-type where the bound variable has type $A_2$. Therefore, the resulting right-hand-side type must use a substitution to change the type of the bound variable.

Prior work (Gundry 2013; Weirich et al. 2013) included a symmetric rule instead of this one because of concern that this rule would be difficult to implement in GHC. Eisenberg (2016) reports that the opposite is true from his experience with GHC 8.0. The symmetric rule requires binding three variables instead of one, while the substitution in the asymmetric version proved no difficulty.

The congruence rule for coercion abstraction types, rule An-CPiCong is similarly asymmetric. This rule motivates the inclusion of rule An-IsoSym, a symmetry coercion between props. As in D, this rule (like reflexivity and transitivity) is derivable from the analogous rules for type equality. However, we need to refer to symmetry coercions in rules An-CPiCong and An-CAbsCong so it is convenient to have syntax for it available. Note that this rule is somewhat different from prior work because we lack injectivity for equated types in propositions. However, our version is more in line with rules rules An-PiCong and An-AbsCong.

There is also a subtle issue related to rule An-CAbsCong, the congruence rule for coercion abstractions, that we discovered in the process of proving the erasure theorem (5.4). In the case for this rule, the premise that the types of the two abstractions are equal is *not* implied by regularity (3.1). Instead, regularity gives us a coercion between $B_1$ and $B_2$ that could rely on $c$. However, the congruence rule for coercion abstraction types does not allow this dependence, so the rule requires an additional coercion $\gamma_4$ to be equivalent to rule E-CAbsCong.

Figure 4 includes a rule not found in the implicit system, rule An-Refl2. We call this rule a form of "reflexivity" because, according to D, the two terms $a$ and $b$ are the same, i.e. they erase to the same result. However, the coercions from these terms mean that they (probably) do not have $\alpha$-equivalent types in DC. However, as long as their types are coercible we should be able to equate them. (Remember that we do not want to equate terms such as $\lambda^+ x{:}\mathbf{Int}.x$ and $\lambda^+ x{:}\mathbf{Bool}.x$ that are erasure equivalent but do not have coercible types.)

Rule An-Refl2 provides *coherence* for our language—uses of coercion proofs (i.e. $a \triangleright \gamma$) do not interfere with equality. Prior work also include similar reasoning (Gundry 2013; Weirich et al. 2013), but only allowed one coercion proof to be eliminated at a time. In contrast, when viewed through the comparison with reflexivity in the implicit language, we can derive a much more efficient way of stating coherence. Proofs using this proof rule may be smaller than in the prior system.

## 5.3 Coercion props and coercion abstraction

DC uses the same syntax for equality props ($a \sim_A b$) as D, where $A$ is the type of the term $a$. The $A$ annotation is not actually needed for decidable type checking in DC as that type can easily be recovered from $a$. However, we include this annotation in the prop to simplify the definition of the erasure operation, shown in definition 5.1.

$\boxed{\Gamma \vdash a \rightsquigarrow b}$ *(DC reduction)*

$$\textsc{An-AppAbs}\quad \frac{}{\Gamma \vdash (\lambda^{\rho} x\!:\!A.b)\ a^{\rho} \rightsquigarrow b\{a/x\}}$$

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

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

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

$$\textsc{An-Combine}\quad \frac{}{\Gamma \vdash (v \triangleright \gamma_1) \triangleright \gamma_2 \rightsquigarrow v \triangleright (\gamma_1 ; \gamma_2)}$$

$$\textsc{An-Push}\quad \frac{\begin{array}{c}\Gamma; \widetilde{\Gamma} \vdash \gamma : \Pi^{\rho} x_1\!:\!A_1 \to B_1 \sim \Pi^{\rho} x_2\!:\!A_2 \to B_2 \\ a_1 = a\{x_2 \triangleright \mathbf{sym}\,(\mathbf{piFst}\,\gamma)/x_1\} \\ \gamma_2 = \gamma @ \mathbf{refl}\,(x_2 \triangleright \mathbf{sym}\,(\mathbf{piFst}\,\gamma) \sim_{\mathbf{piFst}\,\gamma} x_2)\end{array}}{\Gamma \vdash ((\lambda^{\rho} x_1\!:\!A_1.a) \triangleright \gamma)\ b^{\rho} \rightsquigarrow (\lambda^{\rho} x_2\!:\!A_2.(a_1 \triangleright \gamma_2))\ b^{\rho}}$$

$$\textsc{An-CPush}\quad \frac{\begin{array}{c}\Gamma; \widetilde{\Gamma} \vdash \gamma : \forall c_1\!:\!\phi_1.A_1 \sim \forall c_2\!:\!\phi_2.A_2 \\ a_1 = a\{c_2 \triangleright \mathbf{cpiFst}\,(\mathbf{sym}\,\gamma)/c_1\} \\ \gamma_2 = \gamma @ (c_2 \triangleright \mathbf{cpiFst}\,(\mathbf{sym}\,\gamma) \sim c_2)\end{array}}{\Gamma \vdash ((\Lambda c_1\!:\!\phi_1.a) \triangleright \gamma)[\gamma_1] \rightsquigarrow (\Lambda c_2\!:\!\phi_2.(a_1 \triangleright \gamma_2))[\gamma_1]}$$

Fig. 9. DC single-step reduction (excerpt)

When are props ($a \sim_A b$) well formed in DC? To make this question decidable, we place stronger restrictions on the types of $a$ and $b$ in equality propositions than we do in the coercion judgment. In the latter case, we know that a derivation of $\Gamma; \Delta \vdash \gamma : a \sim b$ means that there must exist some coercion between the types of the two terms. This coercion is not stored in the proof $\gamma$ itself, but it can be recovered via Lemma 5.6.

In contrast, to support decidable typechecking, the DC typing rules require two terms in a prop to have *erasure equivalent* types.

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

Given $a$, $b$ and $A$, we can easily access the type of $b$ and determine whether $A$ and $B$ are equal after erasure. However, we cannot necessarily determine whether there exists some coercion that equates $A$ and $B$, as definitional equality is undecidable.

This stronger restriction for props is not problematic. Even with this restriction we can annotate all valid derivations in D. In the case that the types are not erasure-equivalent in some proposition in a derivation, we can always use a cast to make the two terms have erasure-equivalent types. In other words, if we want to form a proposition that $a : A$ and $b : B$ are equal, where we have some $\gamma : A \sim_\star B$, we can use the proposition $(a \triangleright \gamma) \sim_B b$.

Alternatively, we could weaken this restriction by annotating $\gamma$ in DC equality propositions (in addition to $A$). However, we see no need for this modification.

## 5.4 Preservation and Progress for DC

The DC language also supports a preservation theorem for an annotated single-step reduction relation (some rules appear in Figure 9, the full relation is shown in Appendix C.4). This reduction uses a typing context ($\Gamma$) to propagate annotations during evaluation. However, these propagated annotations are irrelevant. The relation erases to the single-step relation for D which does not require type information.

Lemma 5.7 (DC reduction erasure). *If $\Gamma \vdash a \rightsquigarrow b$ and $\Gamma \vdash a : A$ then $\vDash |a| \rightsquigarrow |b|$ or $|a| = |b|$.*

The assumption in this lemma that the expression $a$ type checks is required because of irrelevant applications. In the $\beta$-reduction rule of DC, we always substitute the argument into the body of an abstraction. However, in D, the analogous rule does not have an argument to substitute. In the proof, we need to require that the term type checks to be sure that the argument truly is erasable.

We proved the preservation lemma for DC (shown below) directly. The lemma shown below is stronger than the one we can derive from composing the erasure and annotation theorems with the D preservation result. That version of the lemma does not preserve the type $A$ through reduction. Instead it produces a type $B$ that is erasure-equivalent to $A$. However, our evaluation rules always maintain $\alpha$-equivalent types.

Lemma 5.8 (Preservation for DC). *If* $\Gamma \vdash a : A$ *and* $\Gamma \vdash a \leadsto a'$ *then* $\Gamma \vdash a' : A$.

However, there are properties that we can lift from D through the annotation and erasure lemmas. For example, substitutivity and consistency directly carry over.

Lemma 5.9 (Substitutivity). *If* $\Gamma_1, x : A, \Gamma_2 \vdash b : B$ *and* $\Gamma_1 \vdash a_1 : A$ *and* $\Gamma_1 \vdash a_2 : A$ *and* $\Gamma_1; \Delta \vdash \gamma : a_1 \sim a_2$ *then there exists a* $\gamma'$ *such that* $\Gamma_1, (\Gamma_2\{a_1/x\}); \Delta \vdash \gamma' : b\{a_1/x\} \sim b\{a_2/x\}$.

Lemma 5.10 (Consistency). *If* $\Gamma; \Delta \vdash \gamma : a \sim b$ *and* $\mathbf{Good}\ |\Gamma|\ \Delta$ *then* $\mathbf{consistent}\ |a|\ |b|$.

In fact, this consistency result is also the key to the progress lemma for the annotated language. Before we can state that lemma, we must first define the analogue to values for the annotated language. Values allow explicit type coercions at top level and in the bodies of irrelevant abstractions.

Definition 5.11 (Coerced values and Annotated values).

$$\begin{array}{llll} \textit{coerced values} & w & ::= & v \mid v \triangleright \gamma \\ \textit{annotated values} & v & ::= & \lambda^{+}x{:}A.b \mid \lambda^{-}x{:}A.w \mid \Lambda c{:}\phi.a \\ & & \mid & \star \mid \Pi^{\rho}x{:}A \to B \mid \forall c{:}\phi.A \end{array}$$

Lemma 5.12 (Progress for DC). *If* $\Gamma \vdash a : A$ *and* $\mathbf{Good}\ |\Gamma|\ \Delta$*, then either a is a coerced value, or there exists some* $a'$ *such that* $\Gamma \vdash a \leadsto a'$.

## 6 DESIGN DISCUSSION

We have mentioned some of the factors underlying our designs of D and DC in the prior sections. Here, we discuss some of these design choices in more detail.

### 6.1 Heterogeneous vs. homogeneous equality

A *homogeneous* equality proposition is a four-place relation $a : A \sim b : B$, where the equated terms $a$ and $b$ are required to have definitionally equivalent types ($A$ and $B$) for this proposition to be well-formed. Because $A$ and $B$ are required to be equal, this relation is almost always written as a three-place relation. In contrast, a *heterogeneous* equality proposition is a four place relation $a : A \sim b : B$, where the types of the equated terms may be unrelated. A heterogeneous equality proposition is usually provable only when $A$ and $B$ are equal types (McBride 2000). However, some settings do allow terms with unequal types to be equated (Casinghino et al. 2014; Kimmel et al. 2013).

In the implicit language D, equality propositions are clearly homogeneous. But what about the annotated language? The only equality defined for this language is $\alpha$-equivalence. There is no conversion rule. As a result, it may seem like we have neither homogeneous equality nor heterogeneous equality, as we require the two types to be related, but not with the "definitional equality" of that language. However, we claim that because DC is an annotation of D, the semantics of the equality proposition in DC is the same as that in D. So we use the terminology "homogeneous equality" to refer to equality propositions in both languages.

Homogeneous equality is a natural fit for D. In this language we are required to include the type of the terms in the judgment so we can know at what type they should be compared. To have heterogeneous equality in that context, we would need two different types in equality propositions, i.e. they would be $(a : A \sim b : B)$. We would also need a rule that allows us to extract equalities between $A$ and $B$.

$$\textsc{EA-Kind}\quad \frac{c : (a : A \sim b : B) \in \Gamma}{\Gamma \vDash A : \star \equiv B : \star}$$

Once we had set up D with homogeneous equality, we were inspired to make it also compatible with DC.

In contrast, prior work uses heterogeneous equality (Eisenberg 2016; Gundry 2013; Weirich et al. 2013). As a result, these languages also include a "kind coercion" which extracts a proof of type equality from a proof of term equality.[7] In DC, such a coercion is unnecessary.

However, there is no drawback to using homogeneous equality in D and DC. In these languages, we can *define* a heterogeneous equality proposition by sequencing homogeneous equalities. For example, consider the following definition, where the proposition `a ~ b` is well-formed only because it is preceded by the proposition `k1 ~ k2`.

```
data Heq (a :: k1) (b :: k2) where
   HRefl :: (k1 ~ k2, a ~ b) => Heq a b
```

With this encoding, we do not need a the kind coercion, or any special rules or axioms. Pattern matching for this datatype makes the kind equality available.

One motivation for heterogeneous equality is to support programming with dependently-typed data structures in intensional type theories (McBride 2000). In fact, the Idris language includes heterogeneous equality primitively (Brady 2013). In this setting, heterogeneous equality is necessary to reason about equality between terms whose types are provably equivalent, but not definitionally equivalent. However, in D and DC, we reflect equality propositions into definitional equality so heterogeneous equality is not required for those examples.

Why did prior work use heterogeneous equality in the first place? Part of the reason was to design compositional rules for type coercions, such as rule An-AppCong (and the symmetric version of rule An-AbsCong). However, this work shows that we can have compositional congruence rules in the presence of homogeneous equality.

Another goal was to simplify the implementation of type inference in GHC 8.0 which must emit constraints between types and their kinds. A heterogeneous equality represents both of these at once, whereas two constraints are required with homogeneous equality. However, in terms of constraint generation in GHC, it turns out that generating two propositions was simpler anyways. GHC's solver uses a set of equalities on type variables as a substitution. For this induced substitution to be type-correct, each equality in the set must be homogeneous.

## 6.2 Can we erase more?

Irrelevant abstractions do not completely disappear from our implicit calculus. Even though evaluation continues under irrelevant abstractions, their *locations* are marked in D. In contrast, a Curry-style presentations of System F would allow generalization at any point.

We could imagine replacing our rules for irrelevant argument introduction and elimination with the following alternatives, as is the case in ICC (Miquel 2001).

$$\textsc{EA-IrrelAbs}\quad \frac{\Gamma \vDash A : \star \qquad \Gamma, x : A \vDash a : B}{\Gamma \vDash a : \Pi^- x{:}A \to B}$$

[7] This coercion is similar to Coq's `JMeq_eq` axiom that converts a heterogeneous proposition (`JMeq (a :: k) (b :: k)`) to a homogeneous proposition.

$$\frac{\begin{array}{l}\textsc{EA-IrrelApp}\\ \Gamma \vDash a : \Pi^{-} x{:}A \to B \qquad \Gamma \vDash b : A\end{array}}{\Gamma \vDash a : B\{b/x\}}$$

Adding these rules does not require us to change the annotated language DC. Instead, we would only need to modify the erasure operation to completely remove such abstractions and applications. Furthermore, this modification would strengthen the equational theory of the language.

However, this change complicates the metareasoning of D as $\Pi^{-}$ quantifiers can appear anywhere in a derivation. For example, the inversion lemma 4.4 would allow the type $A$ to be headed by any number of implicit binders before the explicit one. This seems possible, but intricate, so we decided to forgo this extension for a simpler system. We may revisit this decision in future work.

On the other hand, while we can contemplate this change for irrelevant quantification, we definitely cannot make an analogous change for coercion abstraction while preserving type safety. In particular, coercion abstractions can assume bogus equalities (like one between **Int** and **Bool**) and these equalities can be used to type check a stuck program. Precisely because of the possible of hypothetical bogus equalities, we must suspend computation at coercion abstractions.

Previous work by Cretin (2014) and Cretin and Rémy (2014) introduced a calculus built around *consistent* coercion abstraction. Their mechanism allows implicit abstraction over coercions, provided those coercions are shown instantiable. However, unlike the coercion abstraction used here, consistent coercion abstraction cannot be used to implement GADTs. Furthermore, GHC is careful during type inference to only introduce coercion abstraction at points where computation is already suspended, such as in the branches of case analysis.

### 6.3 Variations on the annotated language

The annotated language, DC, that we have developed in this paper is only *one* possible way of annotating D terms to form an equivalent decidable, syntax-directed system. We have already discussed some alternative designs in Section 5.2. However, there are two more variants that are worth further exploration.

First, consider a version of the annotated language that calculates unique types, but only up to erasure-equivalence. This version is equivalent to adding the following conversion rule to DC, which allows any type to be replaced by one that is erasure equivalent.

$$\frac{\begin{array}{l}\textsc{AltAn-Conv}\\ \Gamma \vdash a : A \qquad |A| = |B|\end{array}}{\Gamma \vdash a : B}$$

Because of this built-in treatment of coherence, this version of the language provides a more efficient implementation. In particular, the types of arguments do not necessarily need to be identical to the types that functions expect; they need only erase to the same result. Thus terms require fewer explicit coercions. Eisenberg reported that a related variant of this system was simpler to implement in GHC 8.0. He also explored a variant of this system in his dissertation (see Appendix F).

Second, note that we have made no efforts to compress the annotations required by DC. It is likely that there are versions of the language that can omit some of these annotations. In particular, *bidirectional type checking* (Pierce and Turner 2000) often requires fewer annotations for terms in normal form. Here, the balance is between code size and simplicity. GHC's optimizer must manipulate these typed terms; having simpler rules about where annotations are required makes this job easier. On the other hand, there are known situations where type annotations cause a significant blow up in code size, so it is worth exploring other options, such as rules proposed by Jay and Peyton Jones (2008).

Overall, even though DC may vary, none of these changes will affect D; indeed we should be able to prove analogous erasure and annotation theorems for each of these versions. The ability to contemplate these alternate versions is an argument in favor of the design of D; by rooting ourselves to the simpler language D, we can consider a variety of concrete implementable languages.

## 7 MECHANIZED METATHEORY

All of the definitions, lemmas and proofs in this paper are mechanized in the Coq proof assistant (Coq development team 2004), using tactics from the ssreflect library (Gonthier et al. 2016). We used the Ott tool (Sewell et al. 2010) to generate both the typeset rules in this paper and the Coq definitions that were the basis of our proofs. Our formalization uses a locally nameless representation of terms and variable binding. Some of the proofs regarding substitution and free variables were automatically generated from our language definition via the LNgen tool (Aydemir and Weirich 2010). Our total development is about 20,000 lines of code, plus another 13,000 lines generated by Ott and LNgen.

### 7.1 Decidability proof

Most of our Coq development follows standard practice of proofs by induction over derivations represented with Coq inductive datatypes. Our proof that typechecking DC is decidable required a different style of argument. In other words, we essentially implemented a type checker for DC as a dependently-typed functional program Coq; this function returns not only whether the input term type checks, but also a justification of its answer. If the term type checks, the checking function returns its type as well as an DC derivation for that type. Alternatively, if the term does not type check, then the checker returns a proof that there is no derivation, for any type. We used Coq's `Program` feature to separate the computation of the type checker from the proof that it returns the correct result (Sozeau 2008). We took advantage of notations so that the definition of type checker more closely resembles a functional program. This style of proof is convenient because the computation itself naturally drives the proof flow—in particular, all the branching is performed in the functions, and thus none of it has to be done during the proofs. Furthermore, many of the proof obligations could be discharged automatically.

The most difficult part of this definition was showing that the type checking function actually terminates. We separated this reasoning from the type checker itself by defining an inductive datatype representing the "fuel" required for type checking,and then showed that we could calculate that fuel from the size of the term.

Proving termination was complicated for two reasons. First, the style of defining the inference rules so that the typing context is checked in the leaves of the typing derivation (see, for example, rules An-Var and An-Star) means that termination metric is not a linear function of the size of the input term. Instead of following the rules exactly, we programmed the type checker to ensure the validity of the context whenever new assumptions were added. Second, some typing premises in the rules are merely to access the types of subterms that are already known to be correct. To simplify the termination argument, we replaced these recursive calls to the typechecker with calls to an auxiliary function that calculates the type of an annotated term, assuming that the term has already been checked.Interestingly, these changes not only made the type checker more efficient and the termination argument more straightforward, but they also occasionally simplified the correctness argument.

### 7.2 Why mechanize?

Producing this proof took significant effort, much more than if we had produced a paper description of the results. We undertook this effort partly because reasoning about dependently-typed languages in the presence of nontermination is dangerous. Indeed, including $\star : \star$ leads to inconsistent logics (Martin-Löf 1971), but not necessarily unsafe languages (Cardelli 1986).

In fact, the consistency proofs that appear in both Weirich et al. (2013) and Gundry (2013) are flawed as reported by Eisenberg (2016). Eisenberg shows how to repair the consistency proof and does so for his PiCo language. However, this repair is only relevant to languages with heterogeneous equality.

Furthermore, DC is an admittedly complex language, especially when it comes to the coercion rules (Figures 7 and 8). In the course of our development, we made many changes to our designs in our efforts to prove our desired results. These changes were, of course, motivated by failed proofs due to unexpected interactions in the subtleties of the system. We are not at all confident that we would have seen all of these issues with a purely paper proof.

At the same time, we found the effort in producing a mechanized proof to be more enjoyable than that of paper proofs. Mechanization turns the proof process into a software engineering effort: multiple authors may work together and always be aware of the status of each other's work. Furthermore, we expect that our artifact itself will be useful for future experimentation (perhaps with some of the design variations and extensions that we describe below). Certainly, we have found it useful for quickly ascertaining the impact of a design change throughout the development.

## 8 RELATED WORK

### 8.1 Prior versions of FC with dependency

The most closely related works to this paper are Weirich et al. (2013), Gundry's dissertation (2013) and Eisenberg's dissertation (2016). The current work is the only version to define an implicitly-typed language in addition to DC, a language whose design directly influenced the design of the more practical DC. Furthermore, as we have discussed previously, this variant of FC contains several technical distinctions from prior work, which we summarize here.

We have already discussed in detail the three main differences: that this language uses homogeneous equality instead of heterogeneous equality (Section 6.1), that this system is paired with an implicit language (Section 3), and that all of our proofs have been formalized in Coq (Section 7).

Other more minor technical improvements include:

- This system admits a substitutivity lemma (Lemma 4.9), which Eisenberg was unable to show. Substitutivity is not necessary for safety, though the computational content of this lemma is useful in GHC for optimization.
- This system uses an available set ($\Delta$) to restrict the use of coercion assumptions in rules E-CAbsCong and An-CAbsCong. Weirich et al. used an (invalid) check of how the coercion variable was used in the coercion, and Eisenberg repaired this check with the "almost devoid" relation. However, this approach is not available for D because it does not include explicit coercions. Instead, we use available sets in both languages, both simplifying the check and making it more generally applicable.
- This system includes a signature for general recursive definitions (Section 3.2), following Gundry. In contrast, Eisenberg only includes a fix term and Weirich et al. reuses coercion assumptions for recursive definitions. This latter approach causes difficulty in a full-spectrum calculus. For example, whether a term is a value depends on whether there is some recursive definition for it in the context. Similarly, our definition of parallel reduction automatically unfolds recursive definitions, but ignores all other coercion assumptions.
- This system includes a separate definition of equality for propositions (unlike all prior work). As a result, it includes injectivity only where needed (Section 3.4).
- This system includes an asymmetric rule for congruence rules with binders as opposed to the symmetric rule proposed in Weirich et al. and also used by Gundry (Section 5.2).

### 8.2 Other related calculi

Geuvers and Wiedijk (2004) and van Doorn et al. (2013) develop variants of pure type systems that replace implicit conversions with explicit convertibility proofs. Like this work, they show that the system with explicit equalities is equivalent to the system with implicit equalities, and include asymmetric rules for congruence with binders. However, there are several key differences. First, their work is based in intensional type theory and does not include coercion abstractions. Second, they also use heterogeneous equality instead of homogeneous equality. Finally, their work is based on Pure Type Systems, generalizing over sorts, rules and axioms; whereas we consider only a single instance here. However, given the context of GHC, this generality is not necessary.

The Trellys project developed novel languages for dependently-typed programming, such as $\text{Sep}^3$ (Kimmel et al. 2013) and Zombie (Casinghino et al. 2014; Sjöberg and Weirich 2015). As here, these languages include nontermination, full-spectrum dependent types and irrelevant arguments. Furthermore, the semantics are specified via paired annotated and erased languages. However, unlike this work, the Trellys project focused on call-by-value dependently-typed languages with heterogeneous equality, and on the interaction between terminating and nonterminating computation. In $\text{Sep}^3$ the terminating language is a separate language from the computation language, whereas in Zombie it is defined as a sublanguage of computation via the type system. Neither language includes a separate abstraction form for equality propositions.

Yang et al. (2016) also develop a full-spectrum dependently-typed calculus with type-in-type and general recursion. As in this work, they replace implicit conversion with explicit casts to produce a language with decidable type checking. However, their system is much less expressive: it lacks implicit quantification and any sort of propositional equality for first-class coercions.

## 8.3 Intensional type theory

The dependent type theory that we develop here is different in many ways from existing type theories, such as the ones that underlie other dependently-typed languages such as Epigram, Agda, Idris, or Coq. These languages are founded on *intensional type theory* (Coquand 1986; Martin-Löf 1975), a consistent foundation for mathematics. In contrast, Haskell is a nonterminating language, and thus inconsistent when viewed as a logic. Because Haskell programs do not always terminate, they cannot be used as proofs without running them first. As a result, our language has three major differences from existing type theories:

- *Type-in-type.* Terminating dependently-typed languages require polymorphism to be stratified into a hierarchy of levels, lest they permit an encoding of Girard's paradox (Girard 1972). This stratification motivates complexities in the design of the language, such as cumulativity (Martin-Löf 1984) or level polymorphism (Norell 2007). However, because Haskell does not require termination, there is no motivation for stratification. Programmers have a much simpler system when this hierarchy is collapsed into a single level with the addition of the $\star : \star$ axiom. But, although languages with type-in-type have been proposed before (Martin-Löf 1971) (and been proven type sound (Cardelli 1986)), there is significantly less research into their semantics than there is for intensional type theories.
- *Syntactic type theory.* Type theories are often extended through the use of axioms. For example, adding the law of the excluded middle produces a classical type theory, whereas adding the univalence axiom leads to homotopy type theory. We include axioms for type constructor injectivity, which is sometimes referred to as "syntactic" type theory. However, syntactic, classical and homotopy type theories are known to be mutually inconsistent: type theories used as logical foundations must choose only one of these extensions. Historically, syntactic type theories have not been as well studied as classical and homotopy type theories.
- *Separation between terms and coercions.* Because the term language may not terminate, DC coercions come from a separate, consistent language for reasoning about equality in DC. Propositional equalities are witnessed by coercions instead of computational proofs. This distinction means that coercions are not relevant at runtime and may be erased. Furthermore, DC's form of propositional equality has a flavor of extensional type theory (Martin-Löf 1984)—equality proofs, even assumed ones, can be used without an elimination form.

### 8.4 Other programming languages with dependent types

Our goal is to extend a mature, existing functional programming language with dependent types, in a way that is compatible with existing programs. However, instead of extending an existing language, other projects seek to design new dependently-typed languages from scratch.

The Cayenne Language (Augustsson 1998) was an early prototype in this area. This language was a full-spectrum dependently-typed language, inspired by functional programming. It was implemented as a new typechecker over an existing Haskell implementation, but unlike Dependent Haskell was not intended to be backwards compatible with Haskell. Furthermore, with this architecture, dependent types are only available at the source level—the implementation did not use a strongly typed core language for optimization. The type system of Cayenne was derived from intensional type theory, so differs from that of D and DC. In particular, in Cayenne the kind $\star$ is stratified into a universe hierarchy. This ensures (a) type-level computation terminates (necessary for soundness) and (b) that types can be erased prior to runtime. No other irrelevant arguments can be erased.

More recent languages, based on intensional type theory, include Epigram (McBride 2004), Agda (Norell 2007), and Idris (Brady 2013). Of these, Idris is the most advanced current language designed for practical dependently-typed programming. Because these languages are based on a different foundational type theory, their type systems differ from Dependent Haskell, as mentioned above. On the other hand, as practical tools for programming with dependent types, these tools do support erasure of irrelevant information.

## 9 CONCLUSIONS AND FUTURE WORK

This paper presents two strongly coupled versions of a full-spectrum core calculus for dependent types including nontermination, irrelevant arguments and first class equality coercions. Although these calculi were designed with GHC in mind, we find their approach exciting in its own right as a new approach to dependently-typed programming.

In future work, we plan to extend these calculi with more features of GHC, including recursive datatypes and pattern matching, and type system support for efficient compilation, such as roles (Breitner et al. 2014), and levity polymorphism (Eisenberg and Jones 2017). For the former, we may follow prior work and add datatypes as primitive constructs. However, we are also excited about adopting some of the technology in Cedille (Stump 2016), which would allow us to encode dependent pattern matching with minimal extension.

We also would like to extend the definition of type equality in this language. The more terms that are definitionally equal, the more programs that will type check. Some extensions we plan to consider include rules such as $\eta$-equivalence or additional injectivity rules, including those for type families (Stolarek et al. 2015). We also hope to extend prop equality with more semantic equivalences between propositions.

Finally, because our first-class equality is irrelevant we cannot extend this equality directly with ideas from cubical type theory (Angiuli et al. 2017; Bezem et al. 2014). However, we would also like to explore alternative treatment of coercions that are not erased, so that we can add higher-inductive types to GHC.

## REFERENCES


Carlo Angiuli, Robert Harper, and Todd Wilson. 2017. Computational higher-dimensional type theory. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 680–693.

David Aspinall and Martin Hoffman. 2005. *Dependent Types*. MIT Press, 45–86. http://www.cis.upenn.edu/~bcpierce/attapl/

Lennart Augustsson. 1998. Cayenne—a language with dependent types. In *Proc. ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. ACM, 239–250.

Brian Aydemir and Stephanie Weirich. 2010. *LNgen: Tool Support for Locally Nameless Representations*. Technical Report MS-CIS-10-24. Computer and Information Science, University of Pennsylvania.

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

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

Marc Bezem, Thierry Coquand, and Simon Huber. 2014. A model of type theory in cubical sets. In *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, Vol. 26. 107–128.

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Prog.* 23 (2013).

Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2014. Safe Zero-cost Coercions for Haskell. In *International Conference on Functional Programming (ICFP '14)*. ACM.

Luca Cardelli. 1986. *A Polymorphic Lambda Calculus with Type:Type*. Technical Report 10. Degital Equipment Corporation, SRC.

Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. 2014. Combining Proofs and Programs in a Dependently Typed Language. In *POPL 2014: 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, CA, USA, 33–45.

James Cheney and Ralf Hinze. 2003. *First-Class Phantom Types*. Technical Report. Cornell University.

Coq development team. 2004. *The Coq proof assistant reference manual*. LogiCal Project. http://coq.inria.fr Version 8.0.

Thierry Coquand. 1986. A Calculus of Constructions. (Nov. 1986). manuscript.

Julien Cretin. 2014. *Erasable coercions: a unified approach to type systems*. Ph.D. Dissertation. Université Paris Diderot (Paris 7).

Julien Cretin and Didier Rémy. 2014. System F with Coercion Constraints. In *Logics In Computer Science (LICS)*. ACM.

Haskell B. Curry, J. Roger Hindley, and J.P. Seldin (Eds.). 1972. *Combinatory logic: Volume II*. Amsterdam: North-Holland Pub. Co.

Luis Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*. ACM Press, New York, 207–12.

Peter Dybjer and Anton Setzer. 1999. A finite axiomatization of inductive-recursive definitions. In *Typed Lambda Calculi and Applications, volume 1581 of Lecture Notes in Computer Science*. Springer, 129–146.

Richard A. Eisenberg. 2016. *Dependent Types in Haskell: Theory and Practice*. Ph.D. Dissertation. University of Pennsylvania.

Richard A. Eisenberg and Simon Peyton Jones. 2017. Levity Polymorphism. In *PLDI '17*. To appear.

Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. 2016. Visible Type Application. In *European Symposium on Programming (ESOP)*. 229–254.

H. Geuvers and F. Wiedijk. 2004. A logical framework with explicit conversions. In *LFM'04, Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages, Cork, Ireland*, Carsten Schuermann (Ed.). 32–45.

Jean-Yves Girard. 1971. Une Extension De L'Interpretation De Gödel à L'Analyse, Et Son Application à L'Élimination Des Coupures Dans L'Analyse Et La Theorie Des Types. In *Proceedings of the Second Scandinavian Logic Symposium*, J.E. Fenstad (Ed.). Studies in Logic and the Foundations of Mathematics, Vol. 63. Elsevier, 63 – 92. http://www.sciencedirect.com/science/article/pii/S0049237X08708437

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph.D. Dissertation. Université Paris 7.

Georges Gonthier, Assia Mahboubi, and Enrico Tassi. 2016. *A Small Scale Reflection Extension for the Coq system*. Research Report RR-6455. Inria Saclay Ile de France. https://hal.inria.fr/inria-00258384

Adam Gundry. 2013. *Type Inference, Haskell and Dependent Types*. Ph.D. Dissertation. University of Strathclyde.

Barry Jay and Simon Peyton Jones. 2008. Scrap Your Type Applications. In *Proceedings of the 9th International Conference on Mathematics of Program Construction (MPC '08)*. Springer-Verlag, Berlin, Heidelberg, 2–27. DOI: http://dx.doi.org/10.1007/978-3-540-70594-9_2

Garrin Kimmel, Aaron Stump, Harley D. Eades, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjöberg, Nathin Collins, and Ki Yunh Anh. 2013. Equational reasoning about programs with general recursion and call-by-value semantics. *Progress in Informatics* 10 (March 2013), 19–48. http://www.nii.ac.jp/pi/

Per Martin-Löf. 1971. A Theory of Types. (1971). Unpublished manuscript.

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

Per Martin-Löf. 1984. *Intuitionistic type theory*. Studies in Proof Theory, Vol. 1. Bibliopolis. iv+91 pages.

Conor McBride. 2000. Elimination with a Motive. In *Types for Proofs and Programs, International Workshop, TYPES 2000, Durham, UK, December 8-12, 2000, Selected Papers*. 197–216. DOI:http://dx.doi.org/10.1007/3-540-45842-5_13

Conor McBride. 2004. Epigram. (2004). http://www.dur.ac.uk/CARG/epigram.

Alexandre Miquel. 2001. *The Implicit Calculus of Constructions Extending Pure Type Systems with an Intersection Type Binder and Subtyping*. Springer Berlin Heidelberg, Berlin, Heidelberg, 344–359. DOI:http://dx.doi.org/10.1007/3-540-45413-6_27

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of Functional Programming* 17, 1 (Jan. 2007), 1–82.

Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming (ICFP)*. Portland, OR, USA, 50–61.

Frank Pfenning. 1992. *On the Undecidability of Partial Polymorphic Type Reconstruction*. Technical Report. Pittsburgh, PA, USA.

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

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 1–44. DOI: http://dx.doi.org/10.1145/345099.345100

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium*, B. Robinet (Ed.). Lecture Notes in Computer Science, Vol. 19. Springer Berlin Heidelberg, 408–425. http://dx.doi.org/10.1007/3-540-06859-7_148

Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. 2008. Type checking with open type functions. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*. ACM, New York, NY, USA, 51–62.

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

Vilhelm Sjöberg and Stephanie Weirich. 2015. Programming up to Congruence. In *POPL 2015: 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Mumbai, India, 369–382.

Matthieu Sozeau. 2008. *Un environnement pour la programmation avec types dépendants*. Ph.D. Dissertation. Université Paris 11, Orsay, France.

Jan Stolarek, Simon Peyon Jones, and Richard A. Eisenberg. 2015. Injective Type Families for Haskell. In *Haskell Symposium (Haskell '15)*. ACM.

Aaron Stump. 2016. The Calculus of Dependent Lambda Eliminations. (Sept. 2016). http://homepage.cs.uiowa.edu/~astump/papers/cedille-draft.pdf Submitted for publication.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with type equality coercions. In *Types in languages design and implementation (TLDI '07)*. ACM.

Floris van Doorn, Herman Geuvers, and Freek Wiedijk. 2013. Explicit Convertibility Proofs in Pure Type Systems. In *Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks & Meta-languages: Theory & Practice (LFMTP '13)*. ACM, New York, NY, USA, 25–36.

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X) Modular type inference with local assumptions. *Journal of Functional Programming* 21 (2011), 333–412. Issue Special Issue 4-5. DOI:http://dx.doi.org/10.1017/S0956796811000098

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2008. FPH: First-class polymorphism for Haskell. In *ICFP 2008: The 13th ACM SIGPLAN International Conference on Functional Programming*. Victoria, BC, Canada, 295–306.

Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. 2013. System FC with Explicit Kind Equality. In *International Conference on Functional Programming (ICFP '13)*. ACM.

J.B. Wells. 1999. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic* 98, 1 (1999), 111 – 156.

Yanpeng Yang, Xuan Bi, and Bruno C. d. S. Oliveira. 2016. *Unified Syntax with Iso-types*. Springer International Publishing, Cham, 251–270. DOI:http://dx.doi.org/10.1007/978-3-319-47958-3_14

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a promotion. In *Types in Language Design and Implementation (TLDI '12)*. ACM.


## ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. 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.

# 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. These redundant hypotheses are marked via square brackets in the complete system below.

We 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.

Another source of redundancy comes from our use of the Coq proof assistant. Some of our proofs require the use of induction on judgments 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.

One other minor difference is that this specification also allows the toplevel signature to include type constants $T$, which must have kind $\star$. These type constants have little interact with the rest of the language.

# B TOPLEVEL SIGNATURES

Our results are proven with respect to the following toplevel signatures:

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

$$\Sigma_0 = |\Sigma_1|$$

However, our Coq proofs use these signature definitions opaquely. As a result, any pair of toplevel 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) $\Sigma_0 = |\Sigma_1|$

# C REDUCTION RELATIONS

## C.1 Primitive reduction

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

$$\textsc{Beta-AppAbs}\quad \frac{}{\vDash (\lambda^\rho x.a')\ b^\rho > a'\{b/x\}} \qquad \textsc{Beta-CAppCAbs}\quad \frac{}{\vDash (\Lambda c.a')[\bullet] > a'\{\bullet/c\}} \qquad \textsc{Beta-Axiom}\quad \frac{F \sim a : A \in \Sigma_0}{\vDash F > a}$$

### C.2 Implicit language one-step reduction

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

$$\textsc{E-AbsTerm}\ \frac{\vDash a \leadsto a'}{\vDash \lambda^{-} x.a \leadsto \lambda^{-} x.a'} \qquad \textsc{E-AppLeft}\ \frac{\vDash a \leadsto a'}{\vDash a\ b^{\rho} \leadsto a'\ b^{\rho}} \qquad \textsc{E-CAppLeft}\ \frac{\vDash a \leadsto a'}{\vDash a[\gamma] \leadsto a'[\gamma]} \qquad \textsc{E-AppAbs}\ \frac{}{\vDash (\lambda^{\rho} x.b)\ a^{\rho} \leadsto b\{a/x\}}$$

$$\textsc{E-CAppCAbs}\ \frac{}{\vDash (\Lambda c.b)[\gamma] \leadsto b\{\gamma/c\}} \qquad \textsc{E-Axiom}\ \frac{F \sim a : A \in \Sigma_0}{\vDash F \leadsto a}$$

### C.3 Parallel reduction

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

$$\textsc{Par-Refl}\ \frac{}{\vDash a \Rightarrow a} \qquad \textsc{Par-Beta}\ \frac{\vDash a \Rightarrow (\lambda^{\rho} x.a') \qquad \vDash b \Rightarrow b'}{\vDash a\ b^{\rho} \Rightarrow a'\{b'/x\}} \qquad \textsc{Par-App}\ \frac{\vDash a \Rightarrow a' \qquad \vDash b \Rightarrow b'}{\vDash a\ b^{\rho} \Rightarrow a'\ b'^{\rho}} \qquad \textsc{Par-CBeta}\ \frac{\vDash a \Rightarrow (\Lambda c.a')}{\vDash a[\bullet] \Rightarrow a'\{\bullet/c\}}$$

$$\textsc{Par-CApp}\ \frac{\vDash a \Rightarrow a'}{\vDash a[\bullet] \Rightarrow a'[\bullet]} \qquad \textsc{Par-Abs}\ \frac{\vDash a \Rightarrow a'}{\vDash \lambda^{\rho} x.a \Rightarrow \lambda^{\rho} x.a'} \qquad \textsc{Par-Pi}\ \frac{\vDash A \Rightarrow A' \qquad \vDash B \Rightarrow B'}{\vDash \Pi^{\rho} x\!:\!A \to B \Rightarrow \Pi^{\rho} x\!:\!A' \to B'} \qquad \textsc{Par-CAbs}\ \frac{\vDash a \Rightarrow a'}{\vDash \Lambda c.a \Rightarrow \Lambda c.a'}$$

$$\textsc{Par-CPi}\ \frac{\begin{array}{cc}\vDash A \Rightarrow A' & \vDash B \Rightarrow B' \\ \vDash a \Rightarrow a' & \vDash A_1 \Rightarrow A_1'\end{array}}{\vDash \forall c\!:\!A \sim_{A_1} B.a \Rightarrow \forall c\!:\!A' \sim_{A_1'} B'.a'} \qquad \textsc{Par-Axiom}\ \frac{F \sim a : A \in \Sigma_0}{\vDash F \Rightarrow a}$$

### C.4 Explicit language one-step reduction

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

$$\textsc{An-AppLeft}\ \frac{\Gamma \vdash a \leadsto a'}{\Gamma \vdash a\ b^{\rho} \leadsto a'\ b^{\rho}} \qquad \textsc{An-AppAbs}\ \frac{}{\Gamma \vdash (\lambda^{\rho} x\!:\!A.b)\ a^{\rho} \leadsto b\{a/x\}} \qquad \textsc{An-CAppLeft}\ \frac{\Gamma \vdash a \leadsto a'}{\Gamma \vdash a[\gamma] \leadsto a'[\gamma]} \qquad \textsc{An-CAppCAbs}\ \frac{}{\Gamma \vdash (\Lambda c\!:\!\phi.b)[\gamma] \leadsto b\{\gamma/c\}}$$

$$\textsc{An-AbsTerm}\ \frac{\Gamma \vdash A : \star \qquad \Gamma, x : A \vdash b \leadsto b'}{\Gamma \vdash (\lambda^{-} x\!:\!A.b) \leadsto (\lambda^{-} x\!:\!A.b')} \qquad \textsc{An-Axiom}\ \frac{F \sim a : A \in \Sigma_1}{\Gamma \vdash F \leadsto a} \qquad \textsc{An-CastTerm}\ \frac{\Gamma \vdash a \leadsto a'}{\Gamma \vdash a \triangleright \gamma \leadsto a' \triangleright \gamma} \qquad \textsc{An-Combine}\ \frac{[\mathbf{Value}\ v]}{\Gamma \vdash (v \triangleright \gamma_1) \triangleright \gamma_2 \leadsto v \triangleright (\gamma_1; \gamma_2)}$$

$$\textsc{An-Push}\ \frac{\begin{array}{c}\Gamma; \widetilde{\Gamma} \vdash \gamma : \Pi^{\rho} x_1\!:\!A_1 \to B_1 \sim \Pi^{\rho} x_2\!:\!A_2 \to B_2 \\ a_1 = a\{x_2 \triangleright \mathbf{sym}\,(\mathbf{piFst}\,\gamma)/x_1\} \\ \gamma_2 = \gamma@\mathbf{refl}\,(x_2 \triangleright \mathbf{sym}\,(\mathbf{piFst}\,\gamma) \sim_{\mathbf{piFst}\,\gamma} x_2)\end{array}}{\Gamma \vdash ((\lambda^{\rho} x_1\!:\!A_1.a) \triangleright \gamma)\ b^{\rho} \leadsto (\lambda^{\rho} x_2\!:\!A_2.(a_1 \triangleright \gamma_2))\ b^{\rho}}$$

$$\textsc{An-CPush}\ \frac{\begin{array}{c}\Gamma; \widetilde{\Gamma} \vdash \gamma : \forall c_1\!:\!\phi_1.A_1 \sim \forall c_2\!:\!\phi_2.A_2 \\ a_1 = a\{c_2 \triangleright \mathbf{cpiFst}\,(\mathbf{sym}\,\gamma)/c_1\} \\ \gamma_2 = \gamma@(c_2 \triangleright \mathbf{cpiFst}\,(\mathbf{sym}\,\gamma) \sim c_2)\end{array}}{\Gamma \vdash ((\Lambda c_1\!:\!\phi_1.a) \triangleright \gamma)[\gamma_1] \leadsto (\Lambda c_2\!:\!\phi_2.(a_1 \triangleright \gamma_2))[\gamma_1]}$$

## D FULL SYSTEM SPECIFICATION: IMPLICIT LANGUAGE TYPE SYSTEM

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

$$\textsc{E-Star}\quad \frac{\vDash \Gamma}{\Gamma \vDash \star : \star}$$

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

$$\textsc{E-Pi}\quad \frac{\Gamma, x : A \vDash B : \star \qquad [\Gamma \vDash A : \star]}{\Gamma \vDash \Pi^{\rho} x{:}A \to B : \star}$$

$$\textsc{E-Abs}\quad \frac{\Gamma, x : A \vDash a : B \qquad [\Gamma \vDash A : \star] \qquad \rho \lor x \notin a}{\Gamma \vDash \lambda^{\rho} x.a : \Pi^{\rho} x{:}A \to B}$$

$$\textsc{E-App}\quad \frac{\Gamma \vDash b : \Pi^{+} x{:}A \to B \qquad \Gamma \vDash a : A}{\Gamma \vDash b\ a^{+} : B\{a/x\}}$$

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

$$\textsc{E-Conv}\quad \frac{\Gamma \vDash a : A \qquad \Gamma; \widetilde{\Gamma} \vDash A \equiv B : \star \qquad [\Gamma \vDash B : \star]}{\Gamma \vDash a : B}$$

$$\textsc{E-CPi}\quad \frac{\Gamma, c : \phi \vDash B : \star \qquad [\Gamma \vDash \phi\ \text{ok}]}{\Gamma \vDash \forall c{:}\phi.B : \star}$$

$$\textsc{E-CAbs}\quad \frac{\Gamma, c : \phi \vDash a : B \qquad [\Gamma \vDash \phi\ \text{ok}]}{\Gamma \vDash \Lambda c.a : \forall c{:}\phi.B}$$

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

$$\textsc{E-Const}\quad \frac{\vDash \Gamma \qquad T : \star \in \Sigma_0}{\Gamma \vDash T : \star}$$

$$\textsc{E-Fam}\quad \frac{\vDash \Gamma \qquad F \sim a : A \in \Sigma_0 \qquad [\varnothing \vDash A : \star]}{\Gamma \vDash F : A}$$

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

$$\textsc{E-Wff}\quad \frac{\Gamma \vDash a : A \qquad \Gamma \vDash b : A \qquad [\Gamma \vDash A : \star]}{\Gamma \vDash a \sim_A b\ \text{ok}}$$

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

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

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

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

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

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

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

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

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

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

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

$$\textsc{E-AbsCong}\quad \frac{\begin{array}{c}\Gamma, x : A_1; \Delta \vDash b_1 \equiv b_2 : B \\ [\Gamma \vDash A_1 : \star] \\ \rho \lor x \notin b_1 \qquad \rho \lor x \notin b_2\end{array}}{\Gamma; \Delta \vDash (\lambda^\rho x.b_1) \equiv (\lambda^\rho x.b_2) : \Pi^\rho x{:}A_1 \to B}$$

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma;\Delta \vDash a \equiv b : A \quad \Gamma;\widetilde{\Gamma} \vDash A \equiv B : \star}{\Gamma;\Delta \vDash a \equiv b : B}\ \textsc{E-EqConv}$$

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

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

$$\frac{}{\vDash \varnothing}\ \textsc{E-Empty} \qquad \frac{\vDash \Gamma \quad \Gamma \vDash A : \star \quad x \notin \operatorname{dom}\Gamma}{\vDash \Gamma, x : A}\ \textsc{E-ConsTm} \qquad \frac{\vDash \Gamma \quad \Gamma \vDash \phi\ \mathsf{ok} \quad c \notin \operatorname{dom}\Gamma}{\vDash \Gamma, c : \phi}\ \textsc{E-ConsCo}$$

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

$$\frac{}{\vDash \varnothing}\ \textsc{Sig-Empty} \qquad \frac{\vDash \Sigma \quad T \notin \operatorname{dom}\Sigma}{\vDash \Sigma \cup \{T : \star\}}\ \textsc{Sig-ConsCs} \qquad \frac{\vDash \Sigma \quad F \sim a : A \in \Sigma_0 \quad \varnothing \vDash A : \star \quad \varnothing \vDash a : A \quad F \notin \operatorname{dom}\Sigma}{\vDash \Sigma \cup \{F \sim a : A\}}\ \textsc{Sig-ConsAx}$$

## E FULL SYSTEM SPECIFICATION: EXPLICIT LANGUAGE TYPE SYSTEM

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

$$\frac{\vdash \Gamma}{\Gamma \vdash \star : \star}\ \textsc{An-Star} \qquad \frac{\vdash \Gamma \quad x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsc{An-Var} \qquad \frac{\Gamma, x : A \vdash B : \star \quad [\Gamma \vdash A : \star]}{\Gamma \vdash \Pi^\rho x{:}A \to B : \star}\ \textsc{An-Pi}$$

$$\frac{[\Gamma \vdash A : \star] \quad \Gamma, x : A \vdash a : B \quad \rho \lor x \notin |a|}{\Gamma \vdash \lambda^\rho x{:}A.a : \Pi^\rho x{:}A \to B}\ \textsc{An-Abs}$$

$$\frac{\Gamma \vdash b : \Pi^\rho x{:}A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash b\ a^\rho : B\{a/x\}}\ \textsc{An-App} \qquad \frac{\Gamma \vdash a : A \quad \Gamma;\widetilde{\Gamma} \vdash \gamma : A \sim B \quad \Gamma \vdash B : \star}{\Gamma \vdash a \triangleright \gamma : B}\ \textsc{An-Conv}$$

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

$$\frac{\Gamma \vdash a_1 : \forall c{:}a \sim_{A_1} b.B \quad \Gamma;\widetilde{\Gamma} \vdash \gamma : a \sim b}{\Gamma \vdash a_1[\gamma] : B\{\gamma/c\}}\ \textsc{An-CApp} \qquad \frac{\vdash \Gamma \quad T : \star \in \Sigma_1}{\Gamma \vdash T : \star}\ \textsc{An-Const}$$

$$\frac{\vdash \Gamma \quad F \sim a : A \in \Sigma_1 \quad [\varnothing \vdash A : \star]}{\Gamma \vdash F : A}\ \textsc{An-Fam}$$

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

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

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

$$\textsc{An-Refl2}\quad \frac{\begin{array}{c}\Gamma \vdash a : A \qquad \Gamma \vdash b : B \\ |a| = |b| \qquad \Gamma; \widetilde{\Gamma} \vdash \gamma : A \sim B\end{array}}{\Gamma; \Delta \vdash \mathbf{refl}\,(a \sim_\gamma b) : a \sim b}$$

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

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

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

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

$$\textsc{An-AbsCong}\quad \frac{\begin{array}{c}\Gamma; \Delta \vdash \gamma_1 : A_1 \sim A_2 \\ \Gamma, x : A_1; \Delta \vdash \gamma_2 : b_1 \sim b_2 \\ b_3 = b_2\{x \triangleright \mathbf{sym}\,\gamma_1 / x\} \\ [\Gamma \vdash A_1 : \star] \qquad \Gamma \vdash A_2 : \star \\ \rho \lor x \notin |b_1| \qquad \rho \lor x \notin |b_3| \\ [\Gamma \vdash (\lambda^\rho x{:}A_1.b_2) : B]\end{array}}{\Gamma; \Delta \vdash (\lambda^\rho x{:}\gamma_1.\gamma_2) : (\lambda^\rho x{:}A_1.b_1) \sim (\lambda^\rho x{:}A_2.b_3)}$$

$$\textsc{An-AppCong}\quad \frac{\Gamma;\Delta \vdash \gamma_1 : a_1 \sim b_1 \quad \Gamma;\Delta \vdash \gamma_2 : a_2 \sim b_2 \quad \Gamma \vdash a_1\ {a_2}^{\rho} : A \quad \Gamma \vdash b_1\ {b_2}^{\rho} : B \quad [\Gamma;\widetilde{\Gamma} \vdash \gamma_3 : A \sim B]}{\Gamma;\Delta \vdash \gamma_1\ \gamma_2^{\rho} : a_1\ {a_2}^{\rho} \sim b_1\ {b_2}^{\rho}}$$

$$\textsc{An-PiFst}\quad \frac{\Gamma;\Delta \vdash \gamma : \Pi^{\rho} x\!:\!A_1 \to B_1 \sim \Pi^{\rho} x\!:\!A_2 \to B_2}{\Gamma;\Delta \vdash \mathbf{piFst}\,\gamma : A_1 \sim A_2}$$

$$\textsc{An-PiSnd}\quad \frac{\Gamma;\Delta \vdash \gamma_1 : \Pi^{\rho} x\!:\!A_1 \to B_1 \sim \Pi^{\rho} x\!:\!A_2 \to B_2 \quad \Gamma;\Delta \vdash \gamma_2 : a_1 \sim a_2 \quad \Gamma \vdash a_1 : A_1 \quad \Gamma \vdash a_2 : A_2}{\Gamma;\Delta \vdash \gamma_1 @ \gamma_2 : B_1\{a_1/x\} \sim B_2\{a_2/x\}}$$

$$\textsc{An-CPiCong}\quad \frac{\Gamma;\Delta \vdash \gamma_1 : \phi_1 \sim \phi_2 \quad \Gamma, c : \phi_1;\Delta \vdash \gamma_3 : B_1 \sim B_2 \quad B_3 = B_2\{c \triangleright \mathbf{sym}\,\gamma_1/c\} \quad \Gamma \vdash \forall c\!:\!\phi_1.B_1 : \star \quad \Gamma \vdash \forall c\!:\!\phi_2.B_3 : \star \quad [\Gamma \vdash \forall c\!:\!\phi_1.B_2 : \star]}{\Gamma;\Delta \vdash (\forall c\!:\!\gamma_1.\gamma_3) : (\forall c\!:\!\phi_1.B_1) \sim (\forall c\!:\!\phi_2.B_3)}$$

$$\textsc{An-CAbsCong}\quad \frac{\Gamma;\Delta \vdash \gamma_1 : \phi_1 \sim \phi_2 \quad \Gamma, c : \phi_1;\Delta \vdash \gamma_3 : a_1 \sim a_2 \quad a_3 = a_2\{c \triangleright \mathbf{sym}\,\gamma_1/c\} \quad \Gamma \vdash (\Lambda c\!:\!\phi_1.a_1) : \forall c\!:\!\phi_1.B_1 \quad \Gamma \vdash (\Lambda c\!:\!\phi_2.a_3) : \forall c\!:\!\phi_2.B_2 \quad [\Gamma \vdash (\Lambda c\!:\!\phi_1.a_2) : B] \quad \Gamma;\widetilde{\Gamma} \vdash \gamma_4 : \forall c\!:\!\phi_1.B_1 \sim \forall c\!:\!\phi_2.B_2}{\Gamma;\Delta \vdash (\lambda c\!:\!\gamma_1.\gamma_3 @ \gamma_4) : (\Lambda c\!:\!\phi_1.a_1) \sim (\Lambda c\!:\!\phi_2.a_3)}$$

$$\textsc{An-CAppCong}\quad \frac{\Gamma;\Delta \vdash \gamma_1 : a_1 \sim b_1 \quad \Gamma;\widetilde{\Gamma} \vdash \gamma_2 : a_2 \sim b_2 \quad \Gamma;\widetilde{\Gamma} \vdash \gamma_3 : a_3 \sim b_3 \quad \Gamma \vdash a_1[\gamma_2] : A \quad \Gamma \vdash b_1[\gamma_3] : B \quad [\Gamma;\widetilde{\Gamma} \vdash \gamma_4 : A \sim B]}{\Gamma;\Delta \vdash \gamma_1(\gamma_2, \gamma_3) : a_1[\gamma_2] \sim b_1[\gamma_3]}$$

$$\textsc{An-CPiSnd}\quad \frac{\Gamma;\Delta \vdash \gamma_1 : (\forall c_1\!:\!a \sim_A a'.B_1) \sim (\forall c_2\!:\!b \sim_B b'.B_2) \quad \Gamma;\widetilde{\Gamma} \vdash \gamma_2 : a \sim a' \quad \Gamma;\widetilde{\Gamma} \vdash \gamma_3 : b \sim b'}{\Gamma;\Delta \vdash \gamma_1 @ (\gamma_2 \sim \gamma_3) : B_1\{\gamma_2/c_1\} \sim B_2\{\gamma_3/c_2\}}$$

$$\textsc{An-Cast}\quad \frac{\Gamma;\Delta \vdash \gamma_1 : a \sim a' \quad \Gamma;\Delta \vdash \gamma_2 : a \sim_A a' \sim b \sim_B b'}{\Gamma;\Delta \vdash \gamma_1 \triangleright \gamma_2 : b \sim b'}$$

$$\textsc{An-IsoSnd}\quad \frac{\Gamma;\Delta \vdash \gamma : (a \sim_A a') \sim (b \sim_B b')}{\Gamma;\Delta \vdash \mathbf{typeInj}\,\gamma : A \sim B}$$

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

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

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

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

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

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

$$\textsc{An-Sig-ConsCs}\quad \frac{\vdash \Sigma \qquad T \notin \mathsf{dom}\,\Sigma}{\vdash \Sigma \cup \{T : \star\}}$$

$$\textsc{An-Sig-ConsAx}\quad \frac{\begin{array}{c}\vdash \Sigma \\ F \sim a : A \in \Sigma_1 \qquad \varnothing \vdash A : \star \\ \varnothing \vdash a : A \qquad F \notin \mathsf{dom}\,\Sigma\end{array}}{\vdash \Sigma \cup \{F \sim a : A\}}$$