A Formulation of Dependent ML with Explicit Equality Proofs

Daniel R. Licata   Robert Harper

December, 2005
CMU-CS-05-178

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We study a calculus that supports dependent programming in the style of Xi and Pfenning's Dependent ML. Xi and Pfenning's language determines equality of static data using a built-in decision procedure; ours permits explicit, programmer-written proofs of equality. In this report, we define our calculus' semantics and prove type safety and decidability of type checking; we have mechanized much of these proofs using the Twelf proof assistant. Additionally, we illustrate programming in our calculus through a series of examples. Finally, we present a detailed comparison with other dependently typed languages, including Dependent ML, Epigram, Cayenne, ATS, Ωmega, and RSP1.

This material is based on work supported in part by the National Science Foundation under grants CCR-0204248: Type Refinements and 0121633: ITR/SY+SI: Language Technology for Trustless Software Dissemination. Any opinions, findings, conclusions and recommendations in this publication are the authors' and do not reflect the views of this agency.

Keywords: type systems, dependent types, ML, phase distinction, explicit proofs

1 Introduction

1.1 Dependent Types

Consider the following signature for a module implementing lists of `string`s:

```
signature  STRING_LIST =
     sig
                 type  slist
                  val  nil : slist
                  val  cons : string × slist → slist
                  val  append : slist × slist → slist
                  val  nth : slist × nat → string
                  val  map2 : (string × string → string) × slist × slist → slist
     end.
```

While mostly self-explanatory, this signature leaves some questions unanswered. For example, `nth (lst, i)` is supposed to return the $\mathtt{i}^{th}$ element of `lst`, but what does it do when `i` is not smaller than the length of the list? The function `map2` should map the given function across the two lists, but what does it do when the lists are of different lengths? (Ignore the remaining items in the longer list? Raise an exception?) In a language such as Standard ML [38], these sorts of questions are usually answered in informal comments, and violations of the answers manifest themselves as run-time faults.

In a language with *dependent types* [33, 34, 35]—types that contain run-time programs—programs can be given precise enough types that these questions do not come up. Dependently typed languages generalize the usual function type from ML to a dependent function type, $\Pi\,\mathtt{x}{:}\mathtt{A}.\,\mathtt{B}$, where the argument to the function is allowed to appear in the result type. For example, the above signature can be revised to track the length of a list in its type:

```
signature  SLIST2 =
     sig
             type  slist (x : nat)
              val  nil : slist (0)
              val  cons : Π x:nat. string × slist (x) → slist (1 + x)
              val  append : Π x:nat. Π y:nat. slist (x) × slist (y) → slist (x + y)
              val  nth : Π x:nat. Π i:(nat|i < x). slist (x) → string
              val  map2 : Π x:nat. (string × string → string) × slist (x) × slist (x) → slist (x)
     end.
```

The first line means that the type `slist (E)` is well-formed when `E` is a term of type `nat`. We give precise types to `nil` and `cons`: `nil` is a list of length zero; the result of a `cons` has one more element than the input list. The type of `append` propagates information in the same manner—the length of the output is is the sum of the lengths of the input lists—and makes it more difficult for a buggy version to type check. The type of `map2` ensures that it is only called on lists of the same length, obviating our earlier questions. Similarly, the type of `nth` requires that the offset `i` be less than the length of the list; a primitive implementation could now return the data at the given offset without checking at run-time that the offset is in bounds.

As this example begins to suggest, dependent types can allow interesting properties to be checked in the type system, enable richer interfaces at module boundaries, serve as machine-checked documentation, and obviate some dynamic checks. Proving that a program possesses a more precise type can be harder, but in return the type tells more about the program's behavior. Pragmatically, the programmer can use dependency inasmuch as it seems worthwhile to capture such strong invariants.

1.2 Dependent Types and the Phase Distinction

For the types in the above example to be useful, equality of types should include some notion of equality for the programs embedded in them. For example, it is desirable that a term with type $\texttt{slist}\,(\texttt{1}+\texttt{1})$ also has type $\texttt{slist}\,(2)$. In a pure λ-calculus where program equality is decidable, this is not especially problematic. However, if non-terminating programs are allowed to appear in types, equality will be undecidable.[1] Additionally, it is unclear what it means to allow I/O effects or mutable state in types.

Proposals for dependently typed programming languages have taken various approaches to these problems. Some allow all programs to appear in types by excluding the problematic language features. For example, Epigram [37] insists on totality, disallowing effects and non-termination. Cayenne [4] allows non-termination (but no other effects) by sacrificing decidable type checking; program equality is sound but incomplete. Other proposals [62, 9, 48, 55] use a *phase distinction* [23] to isolate certain programs that can appear in types; the rest of the language can then be arbitrarily effectful.

In present work, we follow these latter proposals in insisting on the phase distinction, which maintains a clear separation between the compile-time (static) and run-time (dynamic) aspects of a program. Type checking is defined to rely only on the compile-time aspects of a program, which include the types of its run-time parts and, to support dependent types, the data that can appear in these types. Execution is free to rely on both the compile-time and the run-time aspects—languages with run-time type analysis [24, 15] compute with compile-time data at run-time, for example. This methodology ensures that the run-time part of the language can be chosen quite freely to have termination, or not, exceptions, or not, store effects, or not, without interfering with type checking. Moreover, standard technology [23, 29, 52, 17, 16] equips a language with the phase distinction with a higher-order module system that itself respects the phase distinction.

1.3 The Need For Proofs

The phase distinction ensures that some notion of program equality can be built into the type system, but existing languages differ in what notion of equality they include and whether they automate reasoning about other propositions. In traditional dependently typed languages such as Cayenne and Epigram, equality is often determined by computation (for example, $\beta\eta$-reduction for functions); additional equalities and other propositions are proven by the programmer using explicit proofs. In contrast, Xi and Pfenning's Dependent ML (DML) [62, 61, 56] is designed to permit fully automated reasoning about compile-time data. In DML, compile-time data, called *indices*, are drawn from a designated *index domain* that is chosen by the language designer. For example, `nil` would have type $\texttt{list}\,(\texttt{z})$, where `z` is a compile-time number in the index domain of natural numbers. Operations on indices (such as $+$) and propositions (such as equality and $<$) are also specified by the language designer. In order to provide fully-automated reasoning about indices, the language designer also fixes a particular constraint solver capable of deciding these propositions. For example, Xi and Pfenning's original implementation has integer indices with a constraint solver for linear integer inequalities [61].

While automation eases the burden on the programmer, a language that decides index propositions using only a constraint solver fixed by the language designer is restricted in several ways:

- For type checking to be decidable, all built-in index propositions must be decidable. For example, if decidable type checking is desired, the built-in index domain cannot even include all of arithmetic; the original DML implementation restricted index multiplication to stay in a decidable fragment.

- The language designer can include undecidable propositions at the cost of decidable type checking, but when the decision procedure loops while proving a proposition that is true, the programmer's

[1]This assumes a sufficiently rich notion of equality—for example, two programs are equal iff they reduce to the same value.

only recourse is to write a different program. In particular, even if the programmer knows why some proposition is true, he cannot convince the type checker.

- The language cannot allow the programmer to define new propositions about indices: the constraint solver will not be able to solve them.
- The language cannot allow the programmer to define new index domains with interesting operations on them. By the previous point, the programmer cannot define new propositions about these indices, limiting their utility. Moreover, decidable notions of equality that are general to all index domains (for example, computational principles) often do not include all desirable equalities; thus, the built-in equality of these new indices would likely be insufficient.

However, recent studies have shown the benefits of allowing a variety of indices, operations, and propositions. For example, static verification of array accesses [61] and many other data structure invariants [57] are possible using DML's integer index domain. Sometimes, this involves encoding other constraint domains as integers (e.g., $\{red, black\}$ as $\{0, 1\}$); using such encodings is less clear to programmers and creates opportunities for errors. Tracking matrix sizes requires going beyond the linear fragment of arithmetic supported by the original DML [9]. Interpreters and compiler transformations that verify object-language typing through the meta-language type system employ representations of object-language types and environments [8] as indices; these index domains are necessarily specific to the object language that is being implemented. Other interpreters use meta-language types themselves [41] as indices. XML documents can be represented typefully and taglessly using indices that describe their structure [64]. Finally, certified type checkers [47] can be written in a language with LF [22] terms and types as indices. The number and variety of these examples suggest that the above restrictions are undesirable.

To support undecidable propositions and programmer-defined index domains and propositions, some recent proposals for phase-respecting dependent types [9, 55] have shifted their focus away from constraint solvers, returning instead to the explicit proofs common in traditional dependently-typed languages. Propositions that do not admit decidable proof search often do admit decidable proof checking. Additionally, unlike a fixed constraint solver, explicit proofs easily extend to new propositions about programmer-defined indices.

1.4 Contributions

We are in the process of designing an ML-like language with programmer-defined index domains; in the previous sections, we have discussed some of the issues that set the context for our work. In particular, to support an ML-like language with unrestricted effects and decidable compile-time type checking, we take the phase distinction as fundamental. To support programmer-defined index domains and unrestricted propositions about them, we base our approach on explicit proofs rather than a constraint solver.

In this report, we lay a foundation by studying a language with the fixed index domain of natural numbers. We answer answer several questions about the design of this calculus:

1. How are indices and index operations represented as compile-time data?
2. How are indices used in the types of run-time data?
3. What notion of equality of compile-time data is built into the type system?
4. What does a programmer do when this notion of equality is insufficient? How can other propositions about indices (such as the $<$ in the type of `nth`) be stated and proven?
5. What does a programmer do when there is insufficient evidence for a proposition?

In our answers to these questions, we have attempted to cull the best features from the existing proposals for phase-respecting languages with programmer-defined index domains. For example, like Ωmega [48, 41], our calculus allows a programmer to define new functions on indices; like ATS [9], our calculus includes a consistent logic in which proofs of index propositions are given. Additionally, in answering these questions we have arrived at a need for some features not found in any existing proposal for phase-respecting dependent types. Most notably, we allow run-time computation with compile-time data such as indices and proofs. This enables some programming techniques familiar from traditional dependently typed languages—for example, it is sometimes useful to have run-time code dispatch on the structure of a proof.

In the remainder of this paper, we detail our calculus's answers to these questions. In Section 2, we describe our calculus's answers at a high level. In Section 3, we present the syntax of our calculus. In Section 4, we illustrate our calculus's answers by implementing the list module from this introduction. In Section 5, we present the semantics of our calculus and overview its meta-theory. We have formalized much of the meta-theory using Twelf [44]; the theorems are presented in Appendix B. In Section 6, we contrast our calculus's answers with those in related work. Finally, in Section 7, we discuss some possibilities for future work. The Twelf code implementing the examples and meta-theory is available on the Web [1].

2 Answers to the Design Questions

In this section, we discuss how our calculus answers the five design questions from Section 1.

2.1 Indices and Index Operations are Represented as Constructors

In a calculus like F_ω [20], compile-time data are called *(type) constructors* and classified by *kinds*; a particular kind TYPE classifies the types of run-time terms. We fit index domains into such a calculus following LX [15] and Ωmega [48, 41]: an index domain is a kind (other than TYPE) and indices are constructors of that kind. In these languages and ours, an index domain is often an inductively-defined kind. For example, the index domain of natural numbers could be defined by

$$\mathtt{kind}\,\mathtt{NAT} \;=\; \mathtt{z} \,|\, \mathtt{s}\,\mathtt{NAT}.$$

Like LX and Ωmega, we support index-level operations (such as the $+$ used in the type of append) as constructor-level functions. In our calculus, these functions can be written using the induction operators associated with the index domains. For example, the $+$ operator could be defined as follows:

$$\mathtt{plus} :: \mathtt{NAT} \to \mathtt{NAT} \to \mathtt{NAT} = \lambda_{\mathtt{c}}\,\mathtt{i}::\mathtt{NAT}.\,\lambda_{\mathtt{c}}\,\mathtt{j}::\mathtt{NAT}.\,\mathtt{NATrec}[\mathtt{u}.\mathtt{NAT}](\mathtt{i},\mathtt{j},\mathtt{i}'.\mathtt{r}.\mathtt{s}\,\mathtt{r}).$$

The NATrec construct allows induction over the kind of natural numbers; the equivalent definition in pattern-matching syntax would be

$$\begin{array}{rcl} \mathtt{plus}\,\mathtt{z}\,\mathtt{j} & = & \mathtt{j} \\ \mathtt{plus}\,(\mathtt{s}\,\mathtt{i}')\,\mathtt{j} & = & \mathtt{s}\,(\mathtt{plus}\,\mathtt{i}'\mathtt{j}). \end{array}$$

Languages such as ATS [9] and RSP1 [55] adopt a relational view of index-level operations; we discuss the trade-offs between this approach and ours in Section 6.

2.2 Indexed Types

Using indices in types, we revise the signature SLIST2 from Section 1 as follows:

```
signature  SLIST3  =
      sig
             type  slist (u :: NAT)
              val  nil : slist (z)
              val  cons : ∀ u::NAT. string × slist (u) → slist (s u)
              val  append : ∀ u::NAT. ∀ v::NAT. slist (u) × slist (v) → slist (plus u v)
              val  nth : ∀ u::NAT. ∀ v::(NAT|v < u). slist (u) → string
              val  map2 : ∀ u::NAT. (string × string → string) × slist (u) × slist (u) → slist (u)
      end.
```

The first line now says that the type $\texttt{slist}\,(\texttt{u})$ is well-formed when $\texttt{u}$ has *kind* $\texttt{NAT}$. Because our indices are constructors, the dependent function constructor Π has been replaced by the more familiar $\forall$ in the subsequent types.

This example allows us to illustrate some terminology. In this SLIST3 signature, $\texttt{slist}\,(\texttt{u} :: \texttt{NAT})$ is a family of types indexed by the constructors of a kind. In contrast, in the original SLIST2 signature, $\texttt{slist}\,(\texttt{x} : \texttt{nat})$ is a family of types indexed by the terms of a type. Because our calculus does not allow run-time terms to appear in types, types can *only* be indexed by a kind; consequently, by *indexed type* we always mean a type that is indexed by the constructors of a kind. Correspondingly, because all data that indexes types comes from the constructor level, we use "index" synonymously with "constructor". In contrast, when describing other languages, we use the phrase *dependent type* to refer to a type that is indexed by the terms of a type.[2] Note that, under this definition, a traditional polymorphic list type defined by

$$\texttt{type list}\,(\texttt{a} :: \texttt{TYPE}) \;=\; \texttt{nil}[\texttt{a} :: \texttt{TYPE}] \mid \texttt{cons}[\texttt{a} :: \texttt{TYPE}]\ \texttt{a}\ (\texttt{list a})$$

is also an indexed type, where the indices happen to be constructors of kind TYPE.

As a second bit of terminology, both $\texttt{slist}\,(\texttt{x} : \texttt{nat})$ and $\texttt{slist}\,(\texttt{u} :: \texttt{NAT})$ are *inductive families* [19].[3] Inductive families generalize ordinary ML-style datatypes in two ways; $\texttt{slist}\,(\texttt{u} :: \texttt{NAT})$ illustrates both. First, the data constructors for inductive families are allowed to target only a subset of the family's indices—for example, nil creates only an $\texttt{slist}\,(\texttt{z})$. Second, the data constructors for one subset of the indices can refer mutually and inductively to other subsets—for example, cons creates an inhabitant of $\texttt{slist}\,(\texttt{s i})$ from an inhabitant of $\texttt{slist}\,(\texttt{i})$. For ordinary polymorphic datatypes, it is possible but tedious to define each instance of an indexed datatype separately; this is not the case for inductive families. Dependent inductive families have been well-studied in type theory [31] and underlie Epigram [37]; indexed inductive families underlie DML's datatypes and GADTs [48].

2.3 Definitional Equality

Like F_ω, our calculus requires a coarser notion of type equality than syntactic equivalence. As mentioned above, it is desirable that the type $\texttt{list}\,(\texttt{plus}\,(\texttt{s z})\,(\texttt{s z}))$ be equal to the type $\texttt{list}\,(\texttt{s}\,(\texttt{s z}))$. To this end, our type system includes a notion of *definitional equality* of type constructors; our definitional equality relation includes β and η rules for constructor-level functions and β rules for constructor-level natural numbers.

[2]Note that this distinction between indexed and dependent types is not always correlated with the distinction between compile-time and run-time data. We believe that, for a programming language, type checking is fundamentally a compile-time activity; consequently, we view any data that can appear in types as compile-time data. For example, the dependent types in Epigram [37] and RSP1 [55] are indexed by terms that, in our view, must nonetheless be seen as compile-time data.

[3]More precisely, $\texttt{slist}\,(\texttt{u} :: \texttt{NAT})$ is a non-uniform mutually- and inductively-defined family of types indexed by constructors of kind NAT.

Under these rules, $\texttt{plus}\ (\texttt{s z})\ (\texttt{s z})$ is indeed equal to $\texttt{s}\ (\texttt{s z})$. The type system permits types and kinds to be silently interchanged with their definitional equals, so if a term has type $\texttt{list}\,(\texttt{plus}\ (\texttt{s z})\ (\texttt{s z}))$, it also has type $\texttt{list}\,(\texttt{s}\ (\texttt{s z}))$.

Unfortunately, there are some equal types that are not related by this notion of definitional equality. For example, consider a client of a SLIST3 module implementing a function

$$\texttt{map2App} : \forall\, \texttt{u}{::}\texttt{NAT}.\, \forall\, \texttt{v}{::}\texttt{NAT}.\, (\texttt{string} \times \texttt{string} \to \texttt{string}) \times \texttt{slist}\,(\texttt{u}) \times \texttt{slist}\,(\texttt{v}) \to \texttt{slist}\,(\texttt{plus u v})$$

that appends the first list with the second, appends the second list with the first, and then maps the given function over these two results. The natural implementation would be

$$\texttt{map2App} = \Lambda\, \texttt{i}{::}\texttt{NAT}.\, \Lambda\, \texttt{j}{::}\texttt{NAT}.\, \lambda\,(\texttt{f}, \texttt{l1}, \texttt{l2}).\, \texttt{map2}\,(\texttt{f}, \texttt{append l1 l2}, \texttt{append l2 l1})$$

but this program is not well typed. The call to map2 requires the two lists to have the same length; when $\texttt{l1} : \texttt{slist}\,(\texttt{i})$ and $\texttt{l2} : \texttt{slist}\,(\texttt{j})$, the type of append gives that the first argument to map2 has type $\texttt{list}\,(\texttt{plus i j})$ whereas the second has type $\texttt{list}\,(\texttt{plus j i})$. Doing the β-reduction resulting from the definition of plus gives

$$\begin{array}{rcl} \texttt{plus i j} & \equiv & \texttt{NATrec}[\_.\texttt{NAT}](\texttt{i}, \texttt{j}, \texttt{i}'.\texttt{r}.\texttt{s r}) \\ \texttt{plus j u} & \equiv & \texttt{NATrec}[\_.\texttt{NAT}](\texttt{j}, \texttt{i}, \texttt{i}'.\texttt{r}.\texttt{s r}). \end{array}$$

Unfortunately, these two constructors are not definitionally equal in our calculus (intuitively, there are no β-redices, and we only include β rules for NATrec). One might hope to enrich definitional equality to include facts like commutativity of addition, but enriching the general equality rules for inductive types to cover such equalities amounts to asking the type checker to search for inductive proofs; thus, it quickly becomes undecidable [26].

2.4 Propositions and Proofs

We address the limitations of definitional equality with a notion of *propositional equality* determined by explicit proofs; proofs of propositional equality can be used to influence the typing of a term. For example, we give a well-typed version of map2App using a proof that plus is commutative. In our calculus, propositional equality is represented as a kind; a proof is a constructor of that kind. More precisely, equality is represented as the inductive family of kinds $\texttt{EQ}_\texttt{N}(\texttt{I}, \texttt{J})$; an inhabitant of a particular member of this family is a witness to the equality of NATs I and J. Inductive families can be used to represent any proposition that is a relation among indices; for example, a kind $\texttt{Lt}_\texttt{N}(\texttt{I}, \texttt{J})$ could be used to represent the $\texttt{u} < \texttt{v}$ constraint in the type of nth. Proofs of such propositions are just constructor-level programs. The same properties that make definitional equality tractable—purity and termination—also make for a consistent logic, so it is reasonable to have the constructor level serve both purposes. This avoids duplication, and it would allow types to be indexed by proofs. However, there is nothing fundamental behind this decision: one could choose to have two syntactic classes for compile-time data, one for types and indices and one for proofs.

For proofs of propositional equality to be useful, they must be able to influence the typing of a term. For example, a proof that $\texttt{EQ}_\texttt{N}(\texttt{i}, \texttt{j})$ should imply that a term with type $\texttt{list}\,(\texttt{i})$ also, in some sense, has type $\texttt{list}\,(\texttt{j})$. This could be achieved by adapting the definitions of propositional equality that have been studied in intensional Martin-Löf type theory [40, 26].[4] In Martin-Löf type theory, propositional equality is defined to be the least relation containing reflexivity, and the elimination form for equality expresses the fact

[4]Intensional here refers to "intensional equality". In type theory with intensional equality, a proof must be explicitly used to retype a term; in type theory with extensional equality, the mere provability of a proposition induces a definitional equality, and therefore an implicit retyping. Because it relies on provability, extensional type theory is undecidable [26]. "Intensional" and "extensional" are used in this sense because many equalities of the extensions of terms are only true by virtue of an inductive proof. In most type theories, definitional equality only equates terms whose intensions are the same; in extensional type theory, definitional equality includes these extensional concepts.

that propositional equals are really definitionally equal. Using the elimination form, it would be possible to prove lemmas such as symmetry, transitivity, and congruence ($\mathtt{EQ_N}(\mathtt{I},\mathtt{J})$ implies $\mathtt{EQ_N}(\mathtt{s\ I},\mathtt{s\ J})$). To retype terms, we could add a run-time elimination form for proofs with the following typing rule:

$$\frac{\Delta,\mathtt{u} :: \mathtt{NAT} \vdash \mathtt{A} :: \mathtt{TYPE} \quad \Delta \vdash \mathtt{P} :: \mathtt{EQ_N}(\mathtt{I},\mathtt{J}) \quad \Delta\,;\,\Gamma \vdash \mathtt{E} : [\mathtt{I}/\mathtt{u}]\mathtt{A}}{\Delta\,;\,\Gamma \vdash \mathtt{subst}[\mathtt{u.A}](\mathtt{P},\mathtt{E}) : [\mathtt{J}/\mathtt{u}]\mathtt{A}}.$$

This construct could transition directly to E at run-time: because all proofs are ultimately reflexivity, the type given to $\mathtt{subst}[\mathtt{u.A}](\mathtt{P},\mathtt{E})$ would always be definitionally equal to the type of E; therefore, these semantics would satisfy type preservation.

This notion of propositional equality allows proofs of equality to be used to retype terms. However, it privileges propositional equality, defined as the identity relation, as the only proposition whose proofs can be eliminated at run-time. In Appendix A, we sketch a simple example that shows why one might want a run-time elimination form for other proofs. In this example, we track the units of measure of scientific quantities (as in Kennedy's languages [28] and Fortress [2]) using indices. Units—meters, seconds, the product of two units, the inverse of a unit, and scalar factors—are represented as constructors in an index domain U. An indexed type $\mathtt{ufloat}\,(\mathtt{u} :: \mathtt{U})$ represents floating-point numbers tagged with a unit; for example, $\mathtt{quantity}[\mathtt{met}]\ \mathtt{4.0}$ represents four meters and has type $\mathtt{ufloat}\,(\mathtt{met})$. Using these types, we define unit-respecting arithmetic operations: addition requires two quantities of the same unit; the unit of the multiplication of two quantities is the product of their units.

Scientific units obey certain algebraic laws; for example, $\mathtt{ufloat}\,(\mathtt{met}\cdot\mathtt{sec}^{-1})$ should be equal to $\mathtt{ufloat}\,(((\mathtt{met}\cdot\mathtt{sec}^{-1})\cdot\mathtt{sec}^{-1})\cdot\mathtt{sec}))$. These laws are not part of definitional equality for the index domain, so we axiomatize a notion of propositional equality that includes them. However, unlike the definition of equality as the least relation containing reflexivity, retyping based on these proofs of equality requires a run-time action: when u and v are propositionally equal units, $\mathtt{ufloat}\,(\mathtt{u})$ and $\mathtt{ufloat}\,(\mathtt{v})$ do not always classify the same terms. Though the retyping function is the identity on the underlying floats (interchanging algebraically equivalent units does not change the magnitude of the quantity), the coercion must package the number with the new unit.

Next, we extend the example by defining a proposition relating two units of the same dimension. Two units have the same dimension when they differ only by a factor of scale; for example, both meters and feet have dimension length. Retyping based on this proposition requires scaling the underlying float by the appropriate factor. To write this retyping function, it is necessary to compute with the proof that the units have the same dimension at run-time: we case-analyze the proof of equality, extract the factor of scale, and then do the appropriate multiplication. Run-time computation with indices and proofs is useful in other circumstances as well; for example, Brady [6] writes a structurally recursive quicksort by induction on the proofs of an accessibility relation.

To support examples like these, we have designed our calculus to allow run-time computation with all compile-time data. To study run-time elimination forms for proofs in our simple calculus, we axiomatize propositional equality for natural numbers inductively: eqn_zz proves that z is equal to z; $\mathtt{eqn\_ss}(\mathtt{I},\mathtt{J},\mathtt{P})$ proves that s I is equal to s J when P proves that I is equal to J. This definition is more like the propositions on units of measure than axiomatizing equality as reflexivity is: it is inductively defined; also, it has more than one constructor, so writing coercions will require a case-analysis with more than one branch.

2.5 Run-time Checks Produce Proofs

Sometimes, desired index relationships will not be evident. For example, a programmer might want to call map2 on two lists with potentially different lengths. One solution is to rewrite as much of the program as necessary to make it evident that the lists have the same length. However, propagating this information will sometimes be difficult or impossible—for example, the lists might be read from user input. In these cases,

Kinds

$$\mathtt{K} \quad ::= \quad \mathtt{TYPE} \mid \Pi_{\mathtt{k}}\, \mathtt{u}_1{::}\mathtt{K}_1.\, \mathtt{K}_2 \mid \mathtt{NAT} \mid \mathtt{EQ_N}(\mathtt{C}_1, \mathtt{C}_2)$$

Type Constructors

$$\begin{array}{lcl}
\mathtt{A,B,C,I,J,P} & ::= & \mathtt{C}_1 \rightarrow \mathtt{C}_2 \mid \mathtt{C}_1 \times \mathtt{C}_2 \mid \mathtt{C}_1 + \mathtt{C}_2 \mid \forall_{\mathtt{K}_2}\, \mathtt{C} \mid \exists_{\mathtt{K}_2}\, \mathtt{C} \mid \mathtt{unit} \mid \mathtt{void} \mid \mathtt{nat\ I} \mid \mathtt{list\ I} \\
 & & \mid \mathtt{u} \mid \lambda_{\mathtt{c}}\, \mathtt{u}{::}\mathtt{K}.\, \mathtt{C} \mid \mathtt{C}_1\ \mathtt{C}_2 \\
 & & \mid \mathtt{z} \mid \mathtt{s\ I} \mid \mathtt{NATrec[u.K](I, C_z, i'.r.C_s)} \\
 & & \mid \mathtt{eqn\_zz} \mid \mathtt{eqn\_ss(I, J, P)} \mid \mathtt{EQ_N rec[i.j.p.K](C, C_{zz}, i.j.p.r.C_{ss})}
\end{array}$$

Terms

$$\begin{array}{lcl}
\mathtt{E} & ::= & \mathtt{x} \mid \lambda\, \mathtt{x{:}A.\, E} \mid \mathtt{E}_1\ \mathtt{E}_2 \mid \mathtt{fix\, x{:}A.\, E} \\
 & & \mid (\mathtt{E}_1, \mathtt{E}_2) \mid \mathtt{fst\ E} \mid \mathtt{snd\ E} \\
 & & \mid \mathtt{inl[A]\ E} \mid \mathtt{inr[A]\ E} \mid \mathtt{case(E, x{:}A.E_1, y{:}B.E_2)} \\
 & & \mid \Lambda\, \mathtt{u{::}K.\, E} \mid \mathtt{E[C]} \mid \mathtt{pack[A](C, E)} \mid \mathtt{unpack[B](E_1, u{::}K.x{:}(A\ u).E_2)} \\
 & & \mid () \mid \mathtt{abort[A]\ E} \\
 & & \mid \mathtt{zero} \mid \mathtt{succ[I]\ E} \mid \mathtt{natcase[u.A](E, E_z, i'.n'.E_s)} \\
 & & \mid \mathtt{nil} \mid \mathtt{cons[I]\ E_1\ E_2} \mid \mathtt{listcase[u.A](E, E_n, hd.i'.tl.E_c)} \\
 & & \mid \mathtt{NATcase[u.A](I, E_z, i'.E_s)} \mid \mathtt{EQ_N case[i.j.p.A](C, E_{zz}, i.j.p.E_{ss})}
\end{array}$$

Contexts

$$\begin{array}{lcl}
\Delta & ::= & \cdot \mid \Delta, \mathtt{u} :: \mathtt{K} \\
\Gamma & ::= & \cdot \mid \Gamma, \mathtt{x} : \mathtt{A}
\end{array}$$

Figure 1: Syntax

the programmer should be able to write a run-time check that, if it is true, establishes the desired property. Then the programmer could check whether the two lists have the same length, call `map2` if they do, and handle the other case appropriately.[5] However, a standard boolean-valued check such as

$$\mathtt{sameLength} : \forall\, \mathtt{i{::}NAT}.\, \forall\, \mathtt{j{::}NAT}.\, \mathtt{list\,(i)} \times \mathtt{list\,(j)} \rightarrow \mathtt{bool}$$

does not serve this purpose: this function returning true has no connection with the truth of the proposition $\mathtt{EQ_N(i, j)}$. In our calculus, the truth of this proposition can be established by a check that returns a proof in the cases where it is true. For example, a programmer could write a function

$$\mathtt{sameLength} : \forall\, \mathtt{i{::}NAT}.\, \forall\, \mathtt{j{::}NAT}.\, \mathtt{list\,(i)} \times \mathtt{list\,(j)} \rightarrow (\exists\, \_{::}\mathtt{EQ_N(i, j)}.\, \mathtt{unit}) + \mathtt{unit}$$

that, instead of just returning `true`, creates a proof that the indices are equal. This proof could then be used to retype the $\mathtt{list\,(j)}$ to a $\mathtt{list\,(i)}$ before calling `map2`.

In general, the programmer can write and use arbitrary run-time functions to check the truth of propositions. In Section 4, we use a proof-producing implementation of `lessThan` to write a version of `nth` that works with an offset that is not necessarily in bounds, calling the statically checked version of `nth` in the case where the offset is in the correct range.

3 Syntax

We present the full syntax of our calculus in Figure 1. We use $[\mathtt{C}_2/\mathtt{u}]\mathtt{C}$, $[\mathtt{C}_2/\mathtt{u}]\mathtt{K}$, $[\mathtt{C}_2/\mathtt{u}]\mathtt{E}$, $[\mathtt{C}_2/\mathtt{u}]\Delta$, $[\mathtt{C}_2/\mathtt{u}]\Gamma$,

[5]An alternative would be to rewrite `map2` with a less strict type, building in a case for lists of different lengths; then the truth of the proposition is irrelevant. However, this leads to unnecessary code duplication—the original `map2` does what the programmer wanted when the lists do have the same length.

and $[\mathtt{E}_2/\mathtt{x}]\mathtt{E}$ for the meta-operations of capture-avoiding substitution. The innermost substitution applies first, so $[\mathtt{C}_2/\mathtt{u}][\mathtt{C}_1/\mathtt{v}]\mathtt{C}$ is $[\mathtt{C}_2/\mathtt{u}]([\mathtt{C}_1/\mathtt{v}]\mathtt{C})$.

Much of the kind and constructor level has been summarized above: it includes types, constructor-level functions, the index domain of natural numbers, and the kind $\mathtt{EQ}_\mathtt{N}(\mathtt{C}_1, \mathtt{C}_2)$ of proofs of equality of constructors of kind NAT. However, because the proofs of equality introduce dependencies of kinds on constructors, the usual function kind of F_ω is replaced with a dependent function kind. We often abbreviate TYPE as T, NAT as N, and $\Pi_\mathtt{k}\,\mathtt{u}{::}\mathtt{K}_2.\,\mathtt{K}$ as $\mathtt{K}_2 \rightarrow_\mathtt{k} \mathtt{K}$ when u is not free in K. We abbreviate $\forall_\mathtt{K}\ (\lambda_\mathtt{c}\,\mathtt{u}{::}\mathtt{K}.\,\mathtt{C})$ and $\exists_\mathtt{K}\ (\lambda_\mathtt{c}\,\mathtt{u}{::}\mathtt{K}.\,\mathtt{C})$ as the more familiar $\forall\,\mathtt{u}{::}\mathtt{K}.\,\mathtt{C}$ and $\exists\,\mathtt{u}{::}\mathtt{K}.\,\mathtt{C}$. But for a few constructs, the term level is a standard polymorphic λ-calculus. Because numbers and lists are given indexed types, there are constructors embedded in the syntax for succ and cons. Additionally, NATcase and $\mathtt{EQ}_\mathtt{N}\mathtt{case}$ are term-level elimination forms for the constructor-level natural numbers and proofs of their equality.

We defer presentation of the typing rules until after the examples—they are mostly familiar. One subtlety is that the case-like elimination forms for NAT and $\mathtt{EQ}_\mathtt{N}(\mathtt{I}, \mathtt{J})$ (both at the constructor level and at the term level) treat these kinds as inductive families, so information about the scrutinized constructor is propagated into the case branches using substitution [19, 31, 37, 24, 15]. In these rules, the scrutinized constructor is allowed to appear free in the result kind/type of each rule, and in each branch the appropriate constructor is substituted. Consider the rule for the constructor-level NATrec:

$$\frac{\Delta, \mathtt{i} :: \mathtt{NAT} \vdash \mathtt{K}\ \mathtt{kind} \quad \Delta \vdash \mathtt{I} :: \mathtt{NAT} \quad \Delta \vdash \mathtt{C}_1 :: [\mathtt{z}/\mathtt{i}]\mathtt{K} \quad \Delta, \mathtt{i}' :: \mathtt{NAT}, \mathtt{r} :: [\mathtt{i}'/\mathtt{i}]\mathtt{K} \vdash \mathtt{C}_2 :: [\mathtt{s}\ \mathtt{i}'/\mathtt{i}]\mathtt{K}}{\Delta \vdash \mathtt{NATrec}[\mathtt{i}.\mathtt{K}](\mathtt{I}, \mathtt{C}_1, \mathtt{i}'.\mathtt{r}.\mathtt{C}_2) :: [\mathtt{I}/\mathtt{i}]\mathtt{K}}\ .$$

In the z branch, the result constructor must only have kind $[\mathtt{z}/\mathtt{u}]\mathtt{K}$. The same device is used in the $\mathtt{EQ}_\mathtt{N}\mathtt{rec}$ rule, where I, J, and the proof itself can appear free in the result. We have also applied this device to term-level cases, where the situation is a bit different: for example, in listcase, the list itself cannot appear in the result type (since terms cannot appear in types); however, its indices can. These rules will be crucial in the examples below.

4 Examples

Our answers to the first and second design questions in Sections 2.1 and 2.2 include some examples of representing indices as type constructors and using indices in types. We illustrate our answers to the remaining three questions in this section. We have implemented all of the following examples using Twelf to run the LF encoding of the semantics as a type checker and an interpreter; the code is available on the Web [1].

We implement the signature from the introduction, tweaked slightly to reflect the fact that there are no strings in our calculus:

```
signature  NLIST  =
      sig
            type  bnat = ∃ u::NAT. nat (u)
            type  nlist (u :: NAT)
             val  nil : nlist (z)
             val  cons : ∀ u::NAT. bnat × nlist (u) → nlist (s u)
             val  append : ∀ u::NAT. ∀ v::NAT. nlist (u) × nlist (v) → nlist (plus u v)
             val  nth : ∀ u::NAT. ∀ v::NAT. ∀ _::Lt_N(v, u). nlist (u) → bnat
             val  map2 : ∀ u::NAT. (bnat × bnat → bnat) × nlist (u) × nlist (u) → nlist (u)
             val  map2App : ∀ u, v::NAT. (bnat × bnat → bnat) × nlist (u) × nlist (v) → nlist (plus u v)
      end.
```

We instead work with lists whose elements are $\exists\,\mathtt{u}{::}\mathtt{NAT}.\,\mathtt{nat}\,(\mathtt{u})$—“blurred” natural numbers whose sizes are statically unknown. Just as $\forall$ in our calculus acts as a dependent function type, $\exists$ acts as a (weak) dependent pair type.[6]

4.1 Using Definitional Equality

Implementing `nlist` as the `list` type of our calculus, `nil` is built-in, and `cons` is just a λ-abstraction over the built-in `cons`. For `append`, recall that the constructor `plus` of kind $\mathtt{NAT} \to_{\mathtt{k}} \mathtt{NAT} \to_{\mathtt{k}} \mathtt{NAT}$ is defined to be $\lambda_{\mathtt{c}}\, \mathtt{i}{::}\mathtt{NAT}.\, \lambda_{\mathtt{c}}\, \mathtt{j}{::}\mathtt{NAT}.\, \mathtt{NATrec}[\mathtt{u}.\mathtt{NAT}](\mathtt{i}, \mathtt{j}, \mathtt{i}'.\mathtt{r}.\mathtt{s}\ \mathtt{r})$. We use `plus` to give a precise type for `append` as follows:

$$\begin{array}{l}\mathtt{append} : \forall\, \mathtt{i}{::}\mathtt{NAT}.\, \forall\, \mathtt{j}{::}\mathtt{NAT}.\, \mathtt{list}\,(\mathtt{i}) \times \mathtt{list}\,(\mathtt{j}) \to \mathtt{list}\,(\mathtt{plus}\ \mathtt{i}\ \mathtt{j}) \ = \\ \mathtt{fix}\, \mathtt{r}{:}\forall\, \mathtt{i}{::}\mathtt{NAT}.\, \forall\, \mathtt{j}{::}\mathtt{NAT}.\, \mathtt{list}\,(\mathtt{i}) \times \mathtt{list}\,(\mathtt{j}) \to \mathtt{list}\,(\mathtt{plus}\ \mathtt{i}\ \mathtt{j}). \\ \ \Lambda\, \mathtt{i}, \mathtt{j}{::}\mathtt{N}.\, \lambda\, \mathtt{ls}{:}\mathtt{list}\,(\mathtt{i}) \times \mathtt{list}\,(\mathtt{j}). \\ \ \ \mathtt{listcase}[\mathtt{i}'.\mathtt{list}\,(\mathtt{plus}\ \mathtt{i}'\ \mathtt{j})](\mathtt{fst}\ \mathtt{ls}, \mathtt{snd}\ \mathtt{ls}, \mathtt{hd}.\mathtt{i}'.\mathtt{tl}.\mathtt{cons}[(\mathtt{plus}\ \mathtt{i}'\ \mathtt{j})]\ \mathtt{hd}\ (\mathtt{r}[\mathtt{i}'][\mathtt{j}]\ (\mathtt{tl}, \mathtt{snd}\ \mathtt{ls}))).\end{array}$$

Typing this term uses both definitional equality and the inductive-family typing rule for `listcase`. For example, the branch of the `listcase` for an empty first list must have type $\mathtt{list}\,(\mathtt{plus}\ \mathtt{z}\ \mathtt{j})$, but the result of the branch has type $\mathtt{list}\,(\mathtt{j})$; fortunately, these types are definitionally equal. Similarly, in the `cons` branch, the result clearly has type $\mathtt{s}\,(\mathtt{plus}\ \mathtt{i}'\ \mathtt{j})$, and definitional equality shows that it has the desired type, $\mathtt{plus}\ (\mathtt{s}\ \mathtt{i}')\ \mathtt{j}$.

Implementing `map2` is similar. One way to implement it is to case on each of the two lists and go into an infinite loop (i.e., raise an exception) in the mismatched cases: because the function can only be called on lists of the same length, these cases will never occur. It is also possible to implement the function in a manifestly total manner, for example by casing on the first list and then use manifestly total `head` and `tail` on the other. We take this approach here, as writing `head` and `tail` is also illustrative. A first attempt at `tail` falls flat:

$$\begin{array}{l}\mathtt{tail} : \forall\, \mathtt{i}{::}\mathtt{N}.\, \mathtt{list}\,(\mathtt{s}\ \mathtt{i}) \to \mathtt{list}\ \mathtt{i} \ = \\ \Lambda\, \mathtt{i}{::}\mathtt{N}.\, \lambda\, \mathtt{l}{:}\mathtt{list}\,(\mathtt{s}\ \mathtt{i}).\, \mathtt{listcase}[\mathtt{i}'.\mathtt{list}\,(\mathtt{i})](\mathtt{l}, ???, \mathtt{hd}.\mathtt{i}'.\mathtt{tl}.\mathtt{tl}).\end{array}$$

First, we have no $\mathtt{list}\,(\mathtt{i})$ to return in the `nil` branch; second, in the `cons` branch, we have not established that $\mathtt{i}'$, the size of the `tl` list exposed by pattern matching, is the same as `i`. One way around these problems is to define the truncated predecessor function for indices,

$$\mathtt{tpred} :: \mathtt{NAT} \to_{\mathtt{k}} \mathtt{NAT} \ = \ \lambda_{\mathtt{c}}\, \mathtt{i}{::}\mathtt{N}.\, \mathtt{NATrec}[\_.\mathtt{NAT}](\mathtt{i}, \mathtt{z}, \mathtt{i}'.\_.\mathtt{i}')$$

and then write

$$\begin{array}{l}\mathtt{tail}' : \forall\, \mathtt{i}{::}\mathtt{N}.\, \mathtt{list}\,(\mathtt{i}) \to \mathtt{list}\,(\mathtt{tpred}\ \mathtt{i}) \ = \\ \Lambda\, \mathtt{i}{::}\mathtt{N}.\, \lambda\, \mathtt{l}{:}\mathtt{list}\,(\mathtt{i}).\, \mathtt{listcase}[\mathtt{i}'.\mathtt{list}\,(\mathtt{tpred}\ \mathtt{i}')](\mathtt{l}, \mathtt{nil}, \mathtt{hd}.\mathtt{i}'.\mathtt{tl}.\mathtt{tl}).\end{array}$$

Then it is simple to write `tail`:

$$\mathtt{tail} : \forall\, \mathtt{i}{::}\mathtt{N}.\, \mathtt{list}\,(\mathtt{s}\ \mathtt{i}) \to \mathtt{list}\,(\mathtt{i}) \ = \ \Lambda\, \mathtt{i}{::}\mathtt{N}.\, \mathtt{tail}'[\mathtt{s}\ \mathtt{i}].$$

To write `tail`, we computed an index in the result type based on an input index. This device does not work for $\mathtt{head} : \forall\, \mathtt{i}{::}\mathtt{N}.\, \mathtt{list}\,(\mathtt{s}\ \mathtt{i}) \to \mathtt{bnat}$, as the result type of this function does not even mention the

[6]By "weak", we mean that the existential has a closed-scope elimination form rather than projections. This is a simple way to maintain the phase distinction: the first projection of an existential projects a constructor from a term; permitting this introduces complications that we wish to avoid here.

index i, so we cannot vary the index in it. However, we can instead define a type (and not just the indices in it) by case analysis on a index. For example,

$$\begin{aligned}&\mathtt{hdtp} \;=\; \lambda_c\, \mathtt{i}{::}\mathtt{NAT}.\, \mathtt{NATrec}[\_.\mathtt{TYPE}](\mathtt{i}, \mathtt{unit}, \_.\_.\mathtt{bnat})\\ &\mathtt{head}' : \forall\, \mathtt{i}{::}\mathtt{N}.\, \mathtt{list}\,(\mathtt{i}) \to \mathtt{hdtp}\ \mathtt{i} \;=\; \Lambda\, \mathtt{i}{::}\mathtt{N}.\, \lambda\, \mathtt{l}{:}\mathtt{list}\,(\mathtt{i}).\, \mathtt{listcase}[\mathtt{i}'.\mathtt{hdtp}\ \mathtt{i}'](\mathtt{l}, (), \mathtt{hd}.\_.\_.\mathtt{hd})\\ &\mathtt{head} : \forall\, \mathtt{i}{::}\mathtt{N}.\, \mathtt{list}\,(\mathtt{s}\ \mathtt{i}) \to \mathtt{bnat} = \Lambda\, \mathtt{i}{::}\mathtt{N}.\, \mathtt{head}'[\mathtt{s}\ \mathtt{i}].\end{aligned}$$

Again, note the uses of definitional equality: when applied to a list (s I), β-reduction shows that head′ has the desired type.

We can now define map2 as follows: [7]

```
map2 : all i::N. (bnat * bnat -> bnat) * list(i) * list(i) -> list(i) =
fix r : all i::N. (bnat * bnat -> bnat) * list(i) * list(i) -> list(i).
  Fn i::N. fn x: (bnat * bnat -> bnat) * list(i) * list(i).
       (listcase[i'.list(i') -> list(i')]
             (fst (snd x),
              fn lst:list(z). lst,
              hd.i'.tl.
                 fn l2:list(s i').
                   cons((fst x) (hd, head[i'] l2),
                        i',
                        r[i'](fst x, (tl, tail[i'] l2)))))
       (snd (snd x)).
```

Aside from illustrating uses of definitional equality and the inductive-family typing rules, the examples in this section (head and tail) show a technique for writing, in a manifestly total form, functions that are only defined for some of the elements of an inductive family. The technique is this: write an auxiliary function with a type that is defined by case analysis (or, more generally, induction) on the indices of the type family; in the irrelevant cases, define the type to be something trivial; then, define the original function to be the restriction of this auxiliary function to the desired indices. At the term level, one could instead fill in the irrelevant cases with an infinite loop. However, the technique described here will also be applicable at the constructor level, where one does not have the luxury of general recursion.

4.2 Using Propositions and Proofs

4.2.1 Proving Simple Theorems

Our kind and constructor level is a first-order intuitionistic logic: dependent functions allow quantification over individuals such as NAT; because we have included propositions in the same syntactic category as individuals, implication is definable using quantification. This mechanism can be used to establish some properties of indices. For example, a simple induction over natural numbers shows that equality is reflexive:

$$\mathtt{eqn\_refl} :: \Pi_k\, \mathtt{i}{::}\mathtt{NAT}.\, \mathtt{EQ_N}(\mathtt{i},\mathtt{i}) \;=\; \lambda_c\, \mathtt{i}{::}\mathtt{NAT}.\, \mathtt{NATrec}[\mathtt{u}.\mathtt{EQ_N}(\mathtt{u},\mathtt{u})](\mathtt{i}, \mathtt{eqn\_zz}, \mathtt{i}'.\mathtt{r}.\mathtt{eqn\_ss}(\mathtt{i}',\mathtt{i}',\mathtt{r})).$$

The following proof of symmetry is an example of induction over proofs:

$$\begin{aligned}&\mathtt{eqn\_sym} :: \Pi_k\, \mathtt{i}{::}\mathtt{NAT}.\, \Pi_k\, \mathtt{j}{::}\mathtt{NAT}.\, \mathtt{EQ_N}(\mathtt{i},\mathtt{j}) \to_k \mathtt{EQ_N}(\mathtt{j},\mathtt{i}) \;=\\ &\lambda_c\, \mathtt{i}{::}\mathtt{NAT}.\, \lambda_c\, \mathtt{j}{::}\mathtt{NAT}.\, \lambda_c\, \mathtt{p}{::}\mathtt{EQ_N}(\mathtt{i},\mathtt{j}).\, \mathtt{EQ_N}\mathtt{rec}[\mathtt{i}'.\mathtt{j}'.\_.\mathtt{EQ_N}(\mathtt{j}',\mathtt{i}')](\mathtt{p}, \mathtt{eqn\_zz}, \mathtt{i}'.\mathtt{j}'.\_.\mathtt{r}.\mathtt{eqn\_ss}(\mathtt{j}',\mathtt{i}',\mathtt{r})).\end{aligned}$$

When inducting over the proof, there is no need to contradict the “off-diagonal” cases as one would have to do in a proof by induction over the two numbers.

[7] In the example code, we sometimes use fn/c for λ_c, pi for Π_k, fn for λ, Fn for Λ, all and exists for $\forall$ and $\exists$, and * for $\times$. Additionally, we use the shorthand i,j::K2 for iterated binding forms, so pi i,j::K2.K is pi i::K2.pi j::K2.K.

Transitivity is a little trickier. One way to do it is as follows:

$$\begin{array}{l}
\mathtt{eqn\_trans} :: \Pi_k\, \mathtt{i}::\mathtt{NAT}.\, \Pi_k\, \mathtt{j}::\mathtt{NAT}.\, \mathtt{EQ_N(i,j)} \to_k \Pi_k\, \mathtt{k}::\mathtt{NAT}.\, \mathtt{EQ_N(j,k)} \to_k \mathtt{EQ_N(i,k)} \; = \\
\lambda_c\, \mathtt{i,j}::\mathtt{N}.\, \lambda_c\, \mathtt{p12}::\mathtt{EQ_N(i,j)}. \\
\mathtt{EQ_Nrec}[\mathtt{i'.j'.\_.}\Pi_k\, \mathtt{k}::\mathtt{N}.\, \mathtt{EQ_N(j',k)} \to_k \mathtt{EQ_N(i',k)}] \\
\qquad (\mathtt{p12}, \\
\qquad\; \lambda_c\, \mathtt{k}::\mathtt{N}.\, \lambda_c\, \mathtt{p23}::\mathtt{EQ_N(i,j)}.\, \mathtt{p23}, \\
\qquad\; \mathtt{i'.j'.p'.r.} \\
\qquad\qquad \lambda_c\, \mathtt{k}::\mathtt{N}.\, \mathtt{NATrec}[\mathtt{k'.EQ_N(s\ j',k')} \to_k \mathtt{EQ_N(s\ i',k')}] \\
\qquad\qquad\qquad\qquad\qquad (\mathtt{k}, \\
\qquad\qquad\qquad\qquad\qquad\; \lambda_c\, \mathtt{p23}::\mathtt{EQ_N(s\ j',z)}.\, \mathtt{eqn\_trans\_contra\ i'\ j'\ p23}, \\
\qquad\qquad\qquad\qquad\qquad\; \mathtt{k'.\_.}\lambda_c\, \mathtt{p23}::\mathtt{EQ_N(s\ j',s\ k')}.\, \mathtt{eqn\_ss(i',k',r\ k'\ (eqn\_pp\ j'\ k'\ p23))))}.
\end{array}$$

By induction on the proof of $\mathtt{EQ_N(i,j)}$, we create a proof that all $\mathtt{k}$ equal to $\mathtt{j'}$ are equal to $\mathtt{i'}$. In the $\mathtt{eqn\_zz}$ case this is easy, since $\mathtt{i'}$ and $\mathtt{j'}$ are both $\mathtt{z}$. In the inductive case, we case analyze $\mathtt{k}$, producing in each case a proof that if $\mathtt{k}$ is equal to $\mathtt{s\ j'}$ then it is equal to $\mathtt{s\ i'}$. When $\mathtt{k}$ is $\mathtt{z}$, the assumption is contradictory (zero and successor are never equal). When $\mathtt{k}$ is $\mathtt{s\ k'}$, we can use the outer inductive hypothesis $\mathtt{r}$ on a proof of $\mathtt{EQ_N(j',k')}$ extracted using the lemma $\mathtt{eqn\_pp}$, and then $\mathtt{eqn\_ss}$ gives the result. The lemmas $\mathtt{eqn\_pp}$ and $\mathtt{eqn\_trans\_contra}$ are defined below. This proof requires more sophisticated uses of the induction principles than the previous lemmas. For example, abstracting over $\mathtt{k}$ in each branch of the $\mathtt{EQ_Nrec}$ ensures that a strong enough inductive hypothesis is available: we appeal to $\mathtt{r}$ on the $\mathtt{k'}$ bound in the $\mathtt{NATrec}$, so assuming $\mathtt{EQ_N(j',k)} \to_k \mathtt{EQ_N(i',k)}$ for a fixed $\mathtt{k}$ bound outside the loop is insufficient. Binding $\mathtt{p23}$ in each branch of the $\mathtt{NATrec}$ propagates index information: in the $\mathtt{z}$ branch, we give it type $\mathtt{EQ_N(s\ j',z)}$, whereas in the successor branch we give it type $\mathtt{EQ_N(s\ j',s\ k')}$. This is a well-known technique [24, 15].

To discharge our first lemma, we need to prove

$$\mathtt{eqn\_pp} :: \Pi_k\, \mathtt{i,j}::\mathtt{N}.\, \mathtt{EQ_N(s\ i,s\ j)} \to_k \mathtt{EQ_N(i,j)}.$$

The kind of this constructor is similar to the type of `tail`; we use the same device:

$$\begin{array}{l}
\mathtt{eqn\_pp'} :: \Pi_k\, \mathtt{i,j}::\mathtt{N}.\, \mathtt{EQ_N(i,j)} \to_k \mathtt{EQ_N(tpred\ i,tpred\ j)} \; = \\
\lambda_c\, \mathtt{i,j}::\mathtt{N}.\, \lambda_c\, \mathtt{p}::\mathtt{EQ_N(i,j)}.\, \mathtt{EQ_Nrec}[\mathtt{i'.j'.p'.EQ_N(tpred\ i',tpred\ j')}](\mathtt{p,eqn\_zz,i'.j'.p'.\_.p'}) \\
\mathtt{eqn\_pp} :: \Pi_k\, \mathtt{i,j}::\mathtt{N}.\, \mathtt{EQ_N(s\ i,s\ j)} \to_k \mathtt{EQ_N(i,j)} \; = \lambda_c\, \mathtt{i,j}::\mathtt{N}.\, \mathtt{eqn\_pp'\ (s\ i)\ (s\ j)}.
\end{array}$$

Now, we must discharge the other assumption by writing

$$\mathtt{eqn\_trans\_contra} :: \Pi_k\, \mathtt{j,i}::\mathtt{N}.\, \mathtt{EQ_N(s\ j,z)} \to_k \mathtt{EQ_N(s\ i,z)}.$$

The hypothesis, that zero is equal to the successor of some number, certainly seems contradictory, but how can we exploit this contradiction within the language? If we had a kind `VOID` with the usual false elim $\mathtt{abort_c[K]\ C}$, we could first demonstrate the contradiction and then use $\mathtt{abort_c}$ to derive this particular consequence. Would it be possible to write this function? Its type would be

$$\mathtt{eqn\_trans\_contra'} :: \Pi_k\, \mathtt{j}::\mathtt{N}.\, \mathtt{EQ_N(s\ j,z)} \to_k \mathtt{VOID}.$$

To implement it, we would need to define a kind by cases on indices (this is similar to the type of $\mathtt{head'}$, which was also defined by cases on indices):

$$\begin{array}{l}
\mathtt{K(z,z)} = \mathtt{UNIT} \\
\mathtt{K(s\_,s\_)} = \mathtt{UNIT} \\
\mathtt{K(z,s\_)} = \mathtt{VOID} \\
\mathtt{K(s\_,z)} = \mathtt{VOID}.
\end{array}$$

Then, $\lambda_c\, \mathtt{j}::\mathtt{N}.\, \lambda_c\, \mathtt{p}::\mathtt{EQ_N}(\mathtt{s\ j}, \mathtt{z}).\, \mathtt{EQ_N rec}[().\mathtt{i}'.\mathtt{j}'.()](, \mathtt{p}, ().\mathtt{i}'.\mathtt{j}'.\mathtt{p}'.\_)\mathtt{K}(\mathtt{i}', \mathtt{j}')$ proves the result, since the result kind is UNIT in all the cases we must consider. Unfortunately, our language does not have the operators needed to define this K at the kind level (kind-level λ and NATrec); we may add them in future work. However, we can still salvage the idea by defining the *indices* of the result kind by cases on the input. While not as general (this trick does not allow the outer "shape" of the kind to vary), it suffices for this lemma:

$$\begin{aligned}&\mathtt{eqn\_trans\_contra} :: \Pi_k\, \mathtt{j}, \mathtt{i}::\mathtt{N}.\, \mathtt{EQ_N}(\mathtt{s\ j}, \mathtt{z}) \to_k \mathtt{EQ_N}(\mathtt{s\ i}, \mathtt{z}) \;=\\ &\lambda_c\, \mathtt{j}, \mathtt{i}::\mathtt{N}.\, \lambda_c\, \mathtt{p}::\mathtt{EQ_N}(\mathtt{s\ j}, \mathtt{z}).\, \mathtt{EQ_N rec}[\mathtt{i}'.\mathtt{j}'.\_.\mathtt{EQ_N}(\mathtt{f\ i\ i}'\ \mathtt{j}', \mathtt{z})](\mathtt{p}, \mathtt{eqn\_zz}, \mathtt{i}'.\mathtt{j}'.\_.\_.\mathtt{eqn\_zz})\end{aligned}$$

where f is

$$\lambda_c\, \mathtt{i}, \mathtt{u}, \mathtt{v}::\mathtt{N}.\, \mathtt{NATrec}[\_.\mathtt{NAT}](\mathtt{u}, \mathtt{z}, \_.\_.\mathtt{NATrec}[\_.\mathtt{NAT}](\mathtt{v}, \mathtt{s\ i}, \_.\_.\mathtt{z})).$$

That is, when the second two arguments match, the value of f is z, so we are proving $\mathtt{EQ_N}(\mathtt{z}, \mathtt{z})$ in each branch; when we substitute the indices of p, it yields what we needed.

4.2.2 Retyping Based on Equality Proofs

Now, we return to the map2App example. Our purported solution was

$$\begin{aligned}&\mathtt{map2App} : \forall\, \mathtt{u}::\mathtt{NAT}.\, \forall\, \mathtt{v}::\mathtt{NAT}.\, (\mathtt{bnat} \times \mathtt{bnat} \to \mathtt{bnat}) \times \mathtt{list}\,(\mathtt{u}) \times \mathtt{list}\,(\mathtt{v}) \to \mathtt{bnat}\,(\mathtt{plus\ u\ v}) \;=\\ &\Lambda\, \mathtt{i}::\mathtt{NAT}.\, \Lambda\, \mathtt{j}::\mathtt{NAT}.\, \lambda\,(\mathtt{f}, \mathtt{l1}, \mathtt{l2}).\, \mathtt{map2}\ (\mathtt{f}, \mathtt{append\ l1\ l2}, \mathtt{append\ l2\ l1}).\end{aligned}$$

The necessary index equalities are

$$\begin{aligned}\mathtt{plus\ i\ j} &\equiv \mathtt{NATrec}[\_.\mathtt{NAT}](\mathtt{i}, \mathtt{j}, \mathtt{i}'.\mathtt{r}.\mathtt{s\ r})\\ \mathtt{plus\ j\ u} &\equiv \mathtt{NATrec}[\_.\mathtt{NAT}](\mathtt{j}, \mathtt{i}, \mathtt{i}'.\mathtt{r}.\mathtt{s\ r}).\end{aligned}$$

We observed that these two constructors are not definitionally equal; however, using the above machinery, it is easy to prove that these two terms are equal:

```
plus_rhz :: pi i::N. EQN(plus i z, i) =
fn/c i::N.
  NATrec [u.EQN(plus u z, u)]
         (i, eqn_zz, i'.r.eqn_ss(plus i' z, i', r))

plus_rhs :: pi i,j::N. EQN(plus i (s j), s (plus i j)) =
fn/c i,j::N.
  NATrec[u.EQN(plus u (s j), s (plus u j))]
        (i, eqn-refl (s j), i'.r.eqn_ss(plus i' (s j), s (plus i' j), r))

plus_commutes :: pi i,j::N. EQN(plus i j, plus j i) =
fn/c i,j::N.
  NATrec[u.EQN(plus u j, plus j u)]
        (u,
         eqn_sym (plus j z) (plus z j) (plus-rhz j),
         i'.r.
           eqn_trans (s (plus i' j))
                     (s (plus j i'))
                     (eqn_ss (plus i' j, plus j' i, r))
                     (plus j (s i'))
                     (eqn_sym (plus j (s i'))
                              (s (plus j i'))
                              (s (plus_rhs j i')))).
```

To finish off map2App, we must be able to exploit a $\mathtt{EQ}_{\mathtt{N}}(\mathtt{plus\ i\ j}, \mathtt{plus\ j\ i})$ to retype a list $(\mathtt{plus\ i\ j})$ to a list $(\mathtt{plus\ j\ i})$. Such a retyping mechanism can be defined using the term-level elimination forms for proofs. For example,

```
retype/list : all i,j::N. all _::EQN(i, j). list(i) -> list(j) =
fix r::all i,j::N. all _::EQN(i, j). list(i) -> list(j).
  Fn i,j::N. Fn p::EQN(i,j).
     EQNcase[i'.j'._. list(i') -> list(j')]
            (p,
             fn x:list(z). x,
             i'.j'.p'.
               fn lst:list(s i').
                  cons[j']
                      (head[i'] lst)
                      (r[i'][j'][p'] (tail[i'] lst))).
```

Then, we can use this retyping function as follows:

```
map2App : all i,j::N. (bnat * bnat -> bnat)
                      * list(i) * list(j)
                      -> list(plus i j)  =
Fn i,j::N. fn x: (bnat * bnat -> bnat) * list(i) * list(j).
   map2[plus i j]
       (fst x,
        (append[i][j] (fst (snd x), snd (snd x)),
         retype/list[plus j i][plus i j]
                    [plus_commutes j i]
                    (append[j][i](snd (snd x), fst (snd x))))).
```

More generally, we can write a retyping function that works for any type indexed by a natural number:

```
retype/NAT : all i,j::N.all _::EQN(i,j).all c::N->T.(c i) -> (c j) =
fix r : all i,j::N.all _::EQN(i,j).all c::NAT->TYPE.(c i) -> (c j).
   Fn i,j::N. Fn _::EQN(i,j).
     EQNcase[i'.j'._. all c::NAT->TYPE.(c i') -> (c j')]
            (p,
             Fn c::N->T. fn x:(c z). x,
             Fn c::N->T. fn x:(c (s i')).
               r[i'][j'][p'][fn/c n::N. c (s n)] x).
```

This is possible because the inductive definition of equality that we have given ultimately amounts to reflexivity.

4.2.3 Restricting the Domain of a Function

As another example, we write nth as a total function that always returns an element of the list (not an option, as in SML). To do so, nth requires a proof that the offset into the list is in bounds. If the equivalent operation were included as primitive, it could be implemented without a run-time bounds check [61].

In the type of nth given in the signature above, the constraint $\mathtt{v} < \mathtt{u}$ is represented by requiring a proof of the proposition $\mathtt{Lt}_{\mathtt{N}}(\mathtt{v}, \mathtt{u})$. This proposition could be treated analogously to $\mathtt{EQ}_{\mathtt{N}}(\mathtt{I}, \mathtt{J})$, with inhabitants lt_zs I and lt_ss I J P and elimination forms giving induction. However, rather than assuming a built-in proposition $\mathtt{Lt}_{\mathtt{N}}(\mathtt{I}, \mathtt{J})$, we define less-than notationally as $\mathtt{EQ}_{\mathtt{N}}(\mathtt{J}, \mathtt{plus\ I\ (s\ K)})$ for some K. If our calculus were extended with Σ-kinds, we could do this properly; here, we imitate it by having the term nth parametrized separately by K and the proof of equality:

```
nth : all u,v,w::N. all _::EQN(u, plus v (s w)).list(u) -> bnat =
fix r : all u,v,w::N. all _::EQN(u, plus v (s w)).list(u) -> bnat.
   Fn u,v,w::N. Fn p::EQN(u, plus v (s w)).
      fn lst:list(u).
         (NATcase[v'. all _::EQN(u, plus v' (s w)). bnat]
                 (v,
                  Fn p'::EQN(u, (s w)).
                     head[w] (retype/list[u][(s w)][p] lst),
                  v'. Fn p'::EQN(u, plus (s v') (s w)).
                       r[plus v' (s w)][v'][w][(eqn-refl (plus v' (s w)))]
                        tail[plus v' (s w)]
                            (retype/list[u][plus (s v') (s w)][p] lst)))
         [p].
```

In this example, polymorphism over proof kinds plays the same role as the subset sorts [62] (and, in later presentations, guards and asserts [9]) in DML. Additionally, this version of `nth` recursively analyzes the constructor-level number `v` at run-time, illustrating run-time computation over indices. Other calculi with indexed types [62, 9, 48] require passing `nth` a term-level `nat` (`v`) for case-analysis.

4.3 Using Run-time Checks To Produce Proofs

In some cases, the size of a list will not be known statically (for example, if the number is the result of run-time input). In these cases, run-time checks can be used to generate proofs. For example, we can write `lessThan` as follows:

```
lessThan : all v,u::N. (exists w::N. exists _::EQN(u, plus v (s w)).unit) + unit =
fix r.
Fn v,u::N.
  NATcase[v'. (exists w::N. exists _::EQN(u, plus v' (s w)).unit) + unit]
    (v,
     NATcase[u'. (exists w::N. exists _::EQN(u',s w).unit) + unit]
       (u,
        inr[exists w::N. exists _::EQN(z, s w).unit] (),
        u'. inl[unit]
               (pack[fn/c w::N. exists _::EQN(s u', s w)]
                    (u',
                     pack[fn/c _::EQN(s u', s u'). unit]
                         (eqn-refl (s u'), ()))))),
     v'.
      NATcase[u'. (exists w::N.
                     exists _::EQN(u', plus (s v') (s w)).unit)
                  + unit]
        (u,
         inr[(exists w::N.
                exists _::EQN(z, plus (s v') (s w)).unit)]
            (),
         case(r[v'][u'],
              ex1 : (exists w::N. exists _::EQN(u', plus v' (s w)).unit).
                inl[unit]
                  (unpack[(exists w::N.
                             exists _::EQN(s u', plus (s v') (s w)).unit)]
                         (ex1,
                          w::N.
                          ex2:(fn/c w::N.exists _::EQN(u', plus v' (s w)).unit)
                              u.
                            unpack[(exists w::N.
                                      exists _::EQN(s u', plus (s v') (s w)).
```

```
                                           unit)]
                                   (ex2,
                                    p::EQN(u’, plus v’ (s w)).
                                    _: (fn/c _::EQN(u’, plus v’ (s w)). unit)
                                       p.
                                      pack[fn/c w::N.
                                             exists _::EQN(s u’, plus (s v’) (s w)).
                                             unit]
                                          (w,
                                           pack[fn/c _::EQN(s u’, plus (s v’) (s w)).unit]
                                               (eqn_ss(u’,
                                                      plus v’ (s w),
                                                      p),
                                               ()))))),
              _:unit.
                inr[(exists w::N.
                              exists _::EQN(s u’, plus (s v’) (s w)).unit)] ())))).
```

If our calculus had `Σ` and sum kinds, it would be possible to instead write this check as a static function whose value is case-analyzed at runtime (using the analogue of `NATcase` for sum kinds).

Using `lessThan`, it is easy to write a version of `nth` that works for any offset:

```
nth/dyn-check : all u::N. bnat * list(u) -> (bnat + unit) =
Fn u::N. fn x : bnat * list(u).
   unpack[bnat+unit]
         (fst x,
          v::NAT. _::(fn/c v::N.nat(v)) v.
              case(lessThan[v][u],
                   y : (exists w::N. exists _::EQN(u, plus v (s w)). unit).
                    inl[unit]
                      (unpack[bnat]
                             (y,
                              w::N.
                               e:(fn/c w::N.
                                     (exists _::EQN(u, plus v (s w)). unit))
                                 w.
                                unpack[bnat]
                                      (e,
                                       p::EQN(u, plus v (s w)).
                                        _: (fn/c _::EQN(u, plus v (s w)). unit)
                                           p.
                                        nth[u][v][w][p] (snd x)))),
                   z:unit. inr[bnat] ())).
```

4.4 Discussion

We now take stock of these examples. The basic approach seems reasonable, in that the code was mostly easy to write. Sometimes, we wrote (up to type annotations) the same code that we would have written without the more precise types (`append`, `map2`). It is interesting to note that these cases were also ones that made good use of definitional equality—this supports our hypothesis that including basic computation in definitional equality is worthwhile. That said, the examples suggest various avenues for improvement:

- In some examples (e.g, `map2App`), it was necessary to write proofs and retyping functions to establish index equalities that were beyond definitional equality; these incurred run-time time and space costs.

There are two opportunities for improvement here: first, one could hope to alleviate the run-time costs of proofs; second, one could hope to reduce the extent to which the programmer has to write proofs.

Along the first line, Brady describes techniques [6] for the compilation of Epigram that reduce the time and space costs of dependent programming. For example, one of his techniques identifies duplicate data in inductive families: if the same index appears more than once, only one copy need be passed at run-time. Another identifies redundant data tags—for `list` $(\mathtt{n})$, either knowing that the index is `z` or `s` or knowing that the list is `nil` or `cons` is sufficient. A third technique identifies inductive families whose indices completely determine their inhabitants—our $\mathtt{EQ_N}(\mathtt{I}, \mathtt{J})$ is an example—and prevents constructing and passing such families at run-time. These techniques seem applicable to our language: like Epigram, our constructor level is a total language with inductive families; some of the techniques do not depend on totality and could consequently be applied to our term level as well. Alternatively, we could potentially use proof irrelevance [43] to collapse kinds that are not used at run-time. Ideally, we would like to support fully general indexed types without ruining the asymptotic time and space complexity of programs.

The second opportunity, reducing the need for writing proofs, seems more ambitious. One approach might be to reintroduce constraint solvers as entities definable in the language.

- In the presentation above, we have used a module syntax to structure the examples. In a language like ours, we anticipate that the module system will be used not only to structure run-time code, but also to develop libraries of index domains and the operations on and proofs about them. The recent techniques for advanced module systems [23, 29, 52, 17, 16] presume that the phase distinction is realized with constructors as compile-time data and terms as run-time data. Because our calculus meets this requirement, it should be relatively straightforward to extend our calculus with such a module system.
- In some examples (e.g., `eqn_trans`), it was necessary to be clever in handling case branches where the indices are contradictory. Epigram's pattern matching notation [36] addresses this problem by generating refutations of contradictory cases automatically in many situations. Because pattern matching is elaborated to elimination rules like those in this paper, it seems likely that we will be able to adapt their techniques to our setting. However, employing their techniques may require us to add kind-level operators and polymorphism.
- As these examples illustrate, the syntax of our language requires many type annotations. It would be desirable to ease this burden as part of building a practical external language on top of our calculus. It may be possible to make some progress using established techniques such as bidirectional type checking as in Pierce and Turner [45], type and term inference as in Twelf [44] and Epigram [37], or type inference for GADTs [42, 51, 46].

5 Semantics

In this section, we present the static and dynamic semantics of our calculus and discuss its meta-theory.

5.1 Static Semantics

Our static semantics comprises the following judgements, which are defined by the rules below.

$\Delta \vdash \mathtt{K}\,\mathtt{kind}$	Kind formation
$\Delta \vdash \mathtt{K} \equiv \mathtt{K}'\,\mathtt{kind}$	Definitional equality of kinds
$\Delta \vdash \mathtt{C} :: \mathtt{K}$	Kinding of constructors
$\Delta \vdash \mathtt{C} \equiv \mathtt{C}' :: \mathtt{K}$	Definitional equality of constructors
$\Delta ; \Gamma \vdash \mathtt{E} : \mathtt{A}$	Typing

Both definitional equality judgements are congruent equivalence relations. Definitional equality of kinds is simply an extension of the definitional equality of the constructors embedded in them. Definitional equality of constructors includes β and extensionality rules for $\Pi_{\mathtt{k}}\,\mathtt{u}{::}\mathtt{K}_2.\,\mathtt{K}$ and β rules for $\mathtt{NAT}$ and $\mathtt{EQ}_{\mathtt{N}}(\mathtt{I}, \mathtt{J})$. Since reflexivity of constructor equality is left as an admissible rule, assumptions $\mathtt{u} :: \mathtt{K}$ should be thought of as shorthand for both $\mathtt{u} :: \mathtt{K}$ and $\mathtt{u} \equiv \mathtt{u} :: \mathtt{K}$ (see the rule `deq-cn-var` below).

In the rules, we assume and maintain the invariant that all types and kinds in the context are well-formed and all variables in the context are distinct. In particular, there is an implicit side condition on binding forms that the bound variable is neither bound in the context nor free in any kind or type in it (we can α-rename it if it is).

$\boxed{\Delta \vdash \mathtt{K}\,\mathtt{kind}}$

$$\frac{}{\Delta \vdash \mathtt{TYPE}\,\mathtt{kind}}\ \texttt{wf-kd-type} \qquad \frac{\Delta \vdash \mathtt{K}_1\,\mathtt{kind} \quad \Delta, \mathtt{u} :: \mathtt{K}_1 \vdash \mathtt{K}_2\,\mathtt{kind}}{\Delta \vdash \Pi_{\mathtt{k}}\,\mathtt{u}{::}\mathtt{K}_1.\,\mathtt{K}_2\,\mathtt{kind}}\ \texttt{wf-kd-pi}$$

$$\frac{}{\Delta \vdash \mathtt{NAT}\,\mathtt{kind}}\ \texttt{wf-kd-nat} \qquad \frac{\Delta \vdash \mathtt{I} :: \mathtt{N} \quad \Delta \vdash \mathtt{J} :: \mathtt{N}}{\Delta \vdash \mathtt{EQ}_{\mathtt{N}}(\mathtt{I}, \mathtt{J})\,\mathtt{kind}}\ \texttt{wf-kd-eqn}$$

$\boxed{\Delta \vdash \mathtt{K}_1 \equiv \mathtt{K}_2\,\mathtt{kind}}$

$$\frac{\Delta \vdash \mathtt{K}_2 \equiv \mathtt{K}_1\,\mathtt{kind}}{\Delta \vdash \mathtt{K}_1 \equiv \mathtt{K}_2\,\mathtt{kind}}\ \texttt{deq-kd-sym} \qquad \frac{\Delta \vdash \mathtt{K}_1 \equiv \mathtt{K}_2\,\mathtt{kind} \quad \Delta \vdash \mathtt{K}_2 \equiv \mathtt{K}_3\,\mathtt{kind}}{\Delta \vdash \mathtt{K}_1 \equiv \mathtt{K}_3\,\mathtt{kind}}\ \texttt{deq-kd-trans}$$

$$\frac{}{\Delta \vdash \mathtt{TYPE} \equiv \mathtt{TYPE}\,\mathtt{kind}}\ \texttt{deq-kd-type} \qquad \frac{\Delta \vdash \mathtt{K}_1 \equiv \mathtt{K}_1'\,\mathtt{kind} \quad \Delta, \mathtt{u} :: \mathtt{K}_1 \vdash \mathtt{K}_2 \equiv \mathtt{K}_2'\,\mathtt{kind}}{\Delta \vdash \Pi_{\mathtt{k}}\,\mathtt{u}{::}\mathtt{K}_1.\,\mathtt{K}_2 \equiv \Pi_{\mathtt{k}}\,\mathtt{u}{::}\mathtt{K}_1'.\,\mathtt{K}_2'\,\mathtt{kind}}\ \texttt{deq-kd-pi}$$

$$\frac{}{\Delta \vdash \mathtt{NAT} \equiv \mathtt{NAT}\,\mathtt{kind}}\ \texttt{deq-kd-nat} \qquad \frac{\Delta \vdash \mathtt{I} \equiv \mathtt{I}' :: \mathtt{N} \quad \Delta \vdash \mathtt{J} \equiv \mathtt{J}' :: \mathtt{N}}{\Delta \vdash \mathtt{EQ}_{\mathtt{N}}(\mathtt{I}, \mathtt{J}) \equiv \mathtt{EQ}_{\mathtt{N}}(\mathtt{I}', \mathtt{J}')\,\mathtt{kind}}\ \texttt{deq-kd-eqn}$$

$\boxed{\Delta \vdash \mathtt{C} :: \mathtt{K}}$

$$\frac{\Delta \vdash \mathtt{C} :: \mathtt{K} \quad \Delta \vdash \mathtt{K} \equiv \mathtt{K}'\,\mathtt{kind}}{\Delta \vdash \mathtt{C} :: \mathtt{K}'}\ \texttt{ofkd-deq} \qquad \frac{}{\Delta, \mathtt{u} :: \mathtt{K}, \Delta' \vdash \mathtt{u} :: \mathtt{K}}\ \texttt{ofkd-var}$$

$$\frac{\Delta \vdash \mathtt{C}_1 :: \mathtt{TYPE} \quad \Delta \vdash \mathtt{C}_2 :: \mathtt{TYPE}}{\Delta \vdash \mathtt{C}_1 \rightarrow \mathtt{C}_2 :: \mathtt{TYPE}}\ \texttt{ofkd-arrow} \qquad \frac{\Delta \vdash \mathtt{C}_1 :: \mathtt{TYPE} \quad \Delta \vdash \mathtt{C}_2 :: \mathtt{TYPE}}{\Delta \vdash \mathtt{C}_1 \times \mathtt{C}_2 :: \mathtt{TYPE}}\ \texttt{ofkd-prod}$$

$$\frac{\Delta \vdash \mathtt{C}_1 :: \mathtt{TYPE} \quad \Delta \vdash \mathtt{C}_2 :: \mathtt{TYPE}}{\Delta \vdash \mathtt{C}_1 + \mathtt{C}_2 :: \mathtt{TYPE}}\ \texttt{ofkd-sum}$$

$$\frac{\Delta \vdash \mathtt{K2}\,\mathtt{kind} \quad \Delta \vdash \mathtt{C} :: \mathtt{K2} \rightarrow_{\mathtt{k}} \mathtt{TYPE}}{\Delta \vdash \forall_{\mathtt{K2}}\,\mathtt{C} :: \mathtt{TYPE}}\ \texttt{ofkd-all} \qquad \frac{\Delta \vdash \mathtt{K2}\,\mathtt{kind} \quad \Delta \vdash \mathtt{C} :: \mathtt{K2} \rightarrow_{\mathtt{k}} \mathtt{TYPE}}{\Delta \vdash \exists_{\mathtt{K2}}\,\mathtt{C} :: \mathtt{TYPE}}\ \texttt{ofkd-exists}$$

$$\frac{}{\Delta \vdash \mathtt{unit}_{\mathtt{c}} :: \mathtt{TYPE}}\ \texttt{ofkd-unit} \qquad \frac{}{\Delta \vdash \mathtt{void}_{\mathtt{c}} :: \mathtt{TYPE}}\ \texttt{ofkd-void}$$

$$\frac{\Delta \vdash \mathtt{I} :: \mathtt{NAT}}{\Delta \vdash \mathtt{nat}\ \mathtt{I} :: \mathtt{TYPE}}\ \texttt{ofkd-nat} \qquad \frac{\Delta \vdash \mathtt{I} :: \mathtt{NAT}}{\Delta \vdash \mathtt{list}\ \mathtt{I} :: \mathtt{TYPE}}\ \texttt{ofkd-list}$$

$$\frac{\Delta \vdash \mathtt{K}_2\ \mathtt{kind} \quad \Delta, \mathtt{u} :: \mathtt{K}_2 \vdash \mathtt{C} :: \mathtt{K}}{\Delta \vdash \lambda_{\mathtt{c}}\ \mathtt{u}{::}\mathtt{K}_2.\,\mathtt{C} :: \Pi_{\mathtt{k}}\ \mathtt{u}{::}\mathtt{K}_2.\,\mathtt{K}}\ \texttt{ofkd-fn} \qquad \frac{\Delta \vdash \mathtt{C}_1 :: \Pi_{\mathtt{k}}\ \mathtt{u}{::}\mathtt{K}_2.\,\mathtt{K} \quad \Delta \vdash \mathtt{C}_2 :: \mathtt{K}_2}{\Delta \vdash \mathtt{C}_1\ \mathtt{C}_2 :: [\mathtt{C}_2/\mathtt{u}]\mathtt{K}}\ \texttt{ofkd-app}$$

$$\frac{}{\Delta \vdash \mathtt{z} :: \mathtt{NAT}}\ \texttt{ofkd-z} \qquad \frac{\Delta \vdash \mathtt{I} :: \mathtt{NAT}}{\Delta \vdash \mathtt{s}\ \mathtt{I} :: \mathtt{NAT}}\ \texttt{ofkd-s}$$

$$\frac{\Delta, \mathtt{i} :: \mathtt{NAT} \vdash \mathtt{K}\ \mathtt{kind} \quad \Delta \vdash \mathtt{I} :: \mathtt{NAT} \quad \Delta \vdash \mathtt{C}_1 :: [\mathtt{z}/\mathtt{i}]\mathtt{K} \quad \Delta, \mathtt{i}' :: \mathtt{NAT}, \mathtt{r} :: [\mathtt{i}'/\mathtt{i}]\mathtt{K} \vdash \mathtt{C}_2 :: [\mathtt{s}\ \mathtt{i}'/\mathtt{i}]\mathtt{K}}{\Delta \vdash \mathtt{NATrec}[\mathtt{i}.\mathtt{K}](\mathtt{I}, \mathtt{C}_1, \mathtt{i}'.\mathtt{r}.\mathtt{C}_2) :: [\mathtt{I}/\mathtt{i}]\mathtt{K}}\ \texttt{ofkd-natrec}$$

$$\frac{}{\Delta \vdash \mathtt{eqn\_zz} :: \mathtt{EQ}_{\mathtt{N}}(\mathtt{z}, \mathtt{z})}\ \texttt{ofkd-eqn-zz} \qquad \frac{\Delta \vdash \mathtt{I} :: \mathtt{NAT} \quad \Delta \vdash \mathtt{J} :: \mathtt{NAT} \quad \Delta \vdash \mathtt{C} :: \mathtt{EQ}_{\mathtt{N}}(\mathtt{I}, \mathtt{J})}{\Delta \vdash \mathtt{eqn\_ss}(\mathtt{I}, \mathtt{J}, \mathtt{P}) :: \mathtt{EQ}_{\mathtt{N}}(\mathtt{s}\ \mathtt{I}, \mathtt{s}\ \mathtt{J})}\ \texttt{ofkd-eqn-ss}$$

$$\frac{\begin{array}{c}\Delta, \mathtt{i} :: \mathtt{N}, \mathtt{j} :: \mathtt{N}, \mathtt{p} :: \mathtt{EQ}_{\mathtt{N}}(\mathtt{i}, \mathtt{j}) \vdash \mathtt{K}\ \mathtt{kind} \\ \Delta \vdash \mathtt{C} :: \mathtt{EQ}_{\mathtt{N}}(\mathtt{I}, \mathtt{J}) \\ \Delta \vdash \mathtt{C}_1 :: [\mathtt{eqn\_zz}/\mathtt{p}][\mathtt{z}/\mathtt{j}][\mathtt{z}/\mathtt{i}]\mathtt{K} \\ \Delta, \mathtt{i} :: \mathtt{N}, \mathtt{j} :: \mathtt{N}, \mathtt{p} :: \mathtt{EQ}_{\mathtt{N}}(\mathtt{i}, \mathtt{j}), \mathtt{r} :: \mathtt{K} \vdash \mathtt{C}_2 :: [\mathtt{eqn\_ss}(\mathtt{i}, \mathtt{j}, \mathtt{p})/\mathtt{p}][\mathtt{s}\ \mathtt{j}/\mathtt{j}][\mathtt{s}\ \mathtt{i}/\mathtt{i}]\mathtt{K}\end{array}}{\Delta \vdash \mathtt{EQ}_{\mathtt{N}}\mathtt{rec}[\mathtt{i}.\mathtt{j}.\mathtt{p}.\mathtt{K}](\mathtt{C}, \mathtt{C}_1, \mathtt{i}.\mathtt{j}.\mathtt{p}.\mathtt{r}.\mathtt{C}_2) :: [\mathtt{C}/\mathtt{p}][\mathtt{J}/\mathtt{j}][\mathtt{I}/\mathtt{i}]\mathtt{K}}\ \texttt{ofkd-eqnrec}$$

$\boxed{\Delta \vdash \mathtt{C}_1 \equiv \mathtt{C}_2 :: \mathtt{K}}$

$$\frac{\Delta \vdash \mathtt{C}_2 \equiv \mathtt{C}_1 :: \mathtt{K}}{\Delta \vdash \mathtt{C}_1 \equiv \mathtt{C}_2 :: \mathtt{K}}\ \texttt{deq-cn-sym} \qquad \frac{\Delta \vdash \mathtt{C}_1 \equiv \mathtt{C}_2 :: \mathtt{K} \quad \Delta \vdash \mathtt{C}_2 \equiv \mathtt{C}_3 :: \mathtt{K}}{\Delta \vdash \mathtt{C}_1 \equiv \mathtt{C}_3 :: \mathtt{K}}\ \texttt{deq-kd-trans}$$

$$\frac{\Delta \vdash \mathtt{C} \equiv \mathtt{C}' :: \mathtt{K} \quad \Delta \vdash \mathtt{K} \equiv \mathtt{K}'\ \mathtt{kind}}{\Delta \vdash \mathtt{C} \equiv \mathtt{C}' :: \mathtt{K}'}\ \texttt{deq-cn-deq-kd} \qquad \frac{}{\Delta, \mathtt{u} :: \mathtt{K}, \Delta' \vdash \mathtt{u} \equiv \mathtt{u} :: \mathtt{K}}\ \texttt{deq-cn-var}$$

$$\frac{\Delta \vdash \mathtt{C}_1 \equiv \mathtt{C}_1' :: \mathtt{TYPE} \quad \Delta \vdash \mathtt{C}_2 \equiv \mathtt{C}_2' :: \mathtt{TYPE}}{\Delta \vdash \mathtt{C}_1 \to \mathtt{C}_2 \equiv \mathtt{C}_1' \to \mathtt{C}_2' :: \mathtt{TYPE}}\ \texttt{deq-cn-arrow}$$

$$\frac{\Delta \vdash \mathtt{C}_1 \equiv \mathtt{C}_1' :: \mathtt{TYPE} \quad \Delta \vdash \mathtt{C}_2 \equiv \mathtt{C}_2' :: \mathtt{TYPE}}{\Delta \vdash \mathtt{C}_1 \times \mathtt{C}_2 \equiv \mathtt{C}_1' \times \mathtt{C}_2' :: \mathtt{TYPE}}\ \texttt{deq-cn-prod}$$

$$\frac{\Delta \vdash \mathtt{C}_1 \equiv \mathtt{C}_1' :: \mathtt{TYPE} \quad \Delta \vdash \mathtt{C}_2 \equiv \mathtt{C}_2' :: \mathtt{TYPE}}{\Delta \vdash \mathtt{C}_1 + \mathtt{C}_2 \equiv \mathtt{C}_1' + \mathtt{C}_2' :: \mathtt{TYPE}}\ \texttt{deq-cn-sum}$$

$$\frac{\Delta \vdash \mathtt{K2} \equiv \mathtt{K2}'\ \mathtt{kind} \quad \Delta \vdash \mathtt{C} \equiv \mathtt{C}' :: \mathtt{K2} \to_{\mathtt{k}} \mathtt{TYPE}}{\Delta \vdash \forall_{\mathtt{K2}}\ \mathtt{C} \equiv \forall_{\mathtt{K2}'}\ \mathtt{C}' :: \mathtt{TYPE}}\ \texttt{deq-cn-all}$$

$$\frac{\Delta \vdash \mathtt{K2} \equiv \mathtt{K2}'\ \mathtt{kind} \quad \Delta \vdash \mathtt{C} \equiv \mathtt{C}' :: \mathtt{K2} \to_{\mathtt{k}} \mathtt{TYPE}}{\Delta \vdash \exists_{\mathtt{K2}}\ \mathtt{C} \equiv \exists_{\mathtt{K2}'}\ \mathtt{C}' :: \mathtt{TYPE}}\ \texttt{deq-cn-exists}$$

$$\frac{}{\Delta \vdash \mathtt{unit_c} \equiv \mathtt{unit_c} :: \mathtt{TYPE}}\ \texttt{deq-cn-unit} \qquad \frac{}{\Delta \vdash \mathtt{void_c} \equiv \mathtt{void_c} :: \mathtt{TYPE}}\ \texttt{deq-cn-void}$$

$$\frac{\Delta \vdash I \equiv I' :: \mathtt{NAT}}{\Delta \vdash \mathtt{nat}\ I \equiv \mathtt{nat}\ I' :: \mathtt{TYPE}}\ \texttt{deq-cn-nat} \qquad \frac{\Delta \vdash I \equiv I' :: \mathtt{NAT}}{\Delta \vdash \mathtt{list}\ I \equiv \mathtt{list}\ I' :: \mathtt{TYPE}}\ \texttt{deq-cn-list}$$

$$\frac{\Delta \vdash K_2 \equiv K_2'\ \mathtt{kind} \quad \Delta, u :: K_2 \vdash C \equiv C' :: K}{\Delta \vdash \lambda_c\, u{::}K_2.\, C \equiv \lambda_c\, u{::}K_2'.\, C' :: \Pi_k\, u{::}K_2.\, K}\ \texttt{deq-cn-fn}$$

$$\frac{\Delta \vdash C_1 \equiv C_1' :: \Pi_k\, u{::}K_2.\, K \quad \Delta \vdash C_2 \equiv C_2' :: K_2}{\Delta \vdash C_1\ C_2 \equiv C_1'\ C_2' :: [C_2/u]K}\ \texttt{deq-cn-app}$$

$$\frac{\Delta, u :: K_2 \vdash C_1 \equiv C_1' :: K \quad \Delta \vdash C2 \equiv C2' :: K_2}{\Delta \vdash (\lambda_c\, u{::}K_2.\, C_1)\ C_2 \equiv [C_2'/u]C_1' :: [C_2/u]K}\ \texttt{deq-cn-app-beta}$$

$$\frac{\Delta \vdash K_2\ \mathtt{kind} \quad \Delta \vdash C :: \Pi_k\, u{::}K_2.\, K \quad \Delta \vdash C' :: \Pi_k\, u{::}K_2.\, K \quad \Delta, u :: K_2 \vdash C\ u \equiv C'\ u :: K}{\Delta \vdash C \equiv C' :: \Pi_k\, u{::}K_2.\, K}\ \texttt{deq-cn-fn-ext}$$

$$\frac{}{\Delta \vdash \mathtt{z} \equiv \mathtt{z} :: \mathtt{NAT}}\ \texttt{deq-cn-z} \qquad \frac{\Delta \vdash I \equiv I' :: \mathtt{NAT}}{\Delta \vdash \mathtt{s}\ I \equiv \mathtt{s}\ I' :: \mathtt{NAT}}\ \texttt{deq-cn-s}$$

$$\frac{\begin{array}{c}\Delta, u :: \mathtt{NAT} \vdash K \equiv K'\ \mathtt{kind} \\ \Delta \vdash I \equiv I' :: \mathtt{NAT} \\ \Delta \vdash C_z \equiv C_z' :: [\mathtt{z}/u]K \\ \Delta, i' :: \mathtt{N}, r :: [i'/u]K \vdash C_s \equiv C_s' :: [\mathtt{s}\ I'/u]K\end{array}}{\Delta \vdash \mathtt{NATrec}[u.K](I, C_z, i'.r.C_s) \equiv \mathtt{NATrec}[u.K'](I', C_z', i'.r.C_s') :: [I/u]K}\ \texttt{deq-cn-natrec}$$

$$\frac{\Delta, u :: \mathtt{N} \vdash K\ \mathtt{kind} \quad \Delta \vdash C_z \equiv C_z' :: [\mathtt{z}/u]K \quad \Delta, i' :: \mathtt{N}, r :: [i'/u]K \vdash C_s :: [\mathtt{s}\ I'/u]K}{\Delta \vdash \mathtt{NATrec}[u.K](\mathtt{z}, C_z, i'.r.C_s) \equiv C_z' :: [\mathtt{z}/u]K}\ \texttt{deq-cn-natrec-beta-z}$$

$$\frac{\begin{array}{c}\Delta, u :: \mathtt{N} \vdash K \equiv K'\ \mathtt{kind} \\ \Delta \vdash I \equiv I' :: \mathtt{NAT} \\ \Delta \vdash C_z \equiv C_z' :: [\mathtt{z}/u]K \\ \Delta, i' :: \mathtt{N}, r :: [i'/u]K \vdash C_s \equiv C_s' :: [\mathtt{s}\ i'/u]K\end{array}}{\Delta \vdash \mathtt{NATrec}[u.K](\mathtt{s}\ I, C_z, i'.r.C_s) \equiv [\mathtt{NATrec}[u.K'](I', C_z', i'.r.C_s')/r][I'/i']C_s' :: [\mathtt{s}\ I/u]K}\ \texttt{deq-cn-natrec-beta-s}$$

$$\frac{}{\Delta \vdash \mathsf{eqn\_zz} \equiv \mathsf{eqn\_zz} :: \mathtt{EQ_N}(\mathtt{z}, \mathtt{z})}\ \texttt{deq-cn-eq-zz}$$

$$\frac{\Delta \vdash I \equiv I' :: \mathtt{NAT} \quad \Delta \vdash J \equiv J' :: \mathtt{NAT} \quad \Delta \vdash P \equiv P' :: \mathtt{EQ_N}(I, J)}{\Delta \vdash \mathsf{eqn\_ss}(I, J, P) \equiv \mathsf{eqn\_ss}(I', J', P') :: \mathtt{EQ_N}(\mathtt{s}\ I, \mathtt{s}\ J)}\ \texttt{deq-cn-eq-ss}$$

$$\frac{\begin{array}{c}\Delta, i :: \mathtt{N}, j :: \mathtt{N}, p :: \mathtt{EQ_N}(i, j) \vdash K \equiv K'\ \mathtt{kind} \\ \Delta \vdash C \equiv C' :: \mathtt{EQ_N}(I, J) \\ \Delta \vdash C_{zz} \equiv C_{zz}' :: [\mathsf{eqn\_zz}/p][\mathtt{z}/j][\mathtt{z}/i]K \\ \Delta, i :: \mathtt{N}, j :: \mathtt{N}, p :: \mathtt{EQ_N}(i, j), r :: K \vdash C_{ss} \equiv C_{ss}' :: [\mathsf{eqn\_ss}(i, j, p)/p][\mathtt{s}\ j/j][\mathtt{s}\ i/i]K\end{array}}{\Delta \vdash \mathtt{EQ_N rec}[i.j.p.K](C, C_{zz}, i.j.p.r.C_{ss}) \equiv \mathtt{EQ_N rec}[i.j.p.K'](C', C_{zz}', i.j.p.r.C_{ss}') :: [C/p][J/j][I/i]K}\ \texttt{deq-cn-eqnrec}$$

$$\frac{\begin{array}{c}\Delta, i :: N, j :: N, p :: EQ_N(i, j) \vdash K\ \mathtt{kind} \\ \Delta \vdash C_{zz} \equiv C'_{zz} :: [eqn\_zz/p][z/j][z/i]K \\ \Delta, i :: N, j :: N, p :: EQ_N(i, j), r :: K \vdash C_{ss} :: [eqn\_ss(i, j, p)/p][s\ j/j][s\ i/i]K\end{array}}{\Delta \vdash EQ_N rec[i.j.p.K](C, C_{zz}, i.j.p.r.C_{ss}) \equiv C'_{zz} :: [eqn\_zz/p][z/j][z/i]K}\ \texttt{deq-cn-eqnrec-beta-zz}$$

deq-cn-eqnrec-beta-ss:

$$\frac{\begin{array}{c}\Delta, i :: N, j :: N, p :: EQ_N(i, j) \vdash K \equiv K'\ \mathtt{kind} \\ \Delta \vdash C_{zz} \equiv C'_{zz} :: [eqn\_zz/p][z/j][z/i]K \\ \Delta, i :: N, j :: N, p :: EQ_N(i, j), r :: K \vdash C_{ss} \equiv C'_{ss} :: [eqn\_ss(i, j, p)/p][s\ j/j][s\ i/i]K \\ \Delta \vdash P \equiv P' :: EQ_N(I, J) \\ \Delta \vdash J \equiv J' :: NAT \\ \Delta \vdash I \equiv I' :: NAT\end{array}}{\Delta \vdash EQ_N rec[i.j.p.K](eqn\_ss(I, J, P), C_{zz}, i.j.p.r.C_{ss}) \equiv [EQ_N rec[i.j.p.K'](P, C'_{zz}, i.j.p.r.C'_{ss})/r][P'/p][J'/j][I'/i]C_{ss} :: [C/p][J/j][I/i]K}$$

$$\boxed{\Delta\,;\Gamma \vdash E : A}$$

$$\frac{\Delta\,;\Gamma \vdash E : A \quad \Delta \vdash A \equiv A' :: TYPE}{\Delta\,;\Gamma \vdash E : A'}\ \texttt{oftp-deq} \qquad \frac{}{\Delta\,;\Gamma, x : A, \Gamma' \vdash x : A}\ \texttt{oftp-var}$$

$$\frac{\Delta \vdash A_2 :: TYPE \quad \Delta\,;\Gamma, x : A_2 \vdash E : A}{\Delta\,;\Gamma \vdash \lambda\, x{:}A_2.\, E : A_2 \to A}\ \texttt{oftp-fn} \qquad \frac{\Delta\,;\Gamma \vdash E_1 : A_2 \to A \quad \Delta\,;\Gamma \vdash E_2 : A_2}{\Delta\,;\Gamma \vdash E_1\ E_2 : A}\ \texttt{oftp-app}$$

$$\frac{\Delta \vdash A :: TYPE \quad \Delta\,;\Gamma, x : A \vdash E : A}{\Delta\,;\Gamma \vdash \mathtt{fix}\, x{:}A.\, E : A}\ \texttt{oftp-fix} \qquad \frac{\Delta\,;\Gamma \vdash E_1 : A_1 \quad \Delta\,;\Gamma \vdash E_2 : A_2}{\Delta\,;\Gamma \vdash (E_1, E_2) : A_1 \times A_2}\ \texttt{oftp-pair}$$

$$\frac{\Delta\,;\Gamma \vdash E : A_1 \times A_2}{\Delta\,;\Gamma \vdash \mathtt{fst}\, E : A_1}\ \texttt{oftp-fst} \qquad \frac{\Delta\,;\Gamma \vdash E : A_1 \times A_2}{\Delta\,;\Gamma \vdash \mathtt{snd}\, E : A_2}\ \texttt{oftp-snd}$$

$$\frac{\Delta \vdash A_2 :: TYPE \quad \Delta\,;\Gamma \vdash E : A_1}{\Delta\,;\Gamma \vdash \mathtt{inl}[A_2]\, E : A_1 + A_2}\ \texttt{oftp-inl} \qquad \frac{\Delta \vdash A_1 :: TYPE \quad \Delta\,;\Gamma \vdash E : A_2}{\Delta\,;\Gamma \vdash \mathtt{inr}[A_1]\, E : A_1 + A_2}\ \texttt{oftp-inr}$$

$$\frac{\Delta\,;\Gamma \vdash E : A_1 + A_2 \quad \Delta\,;\Gamma, x_1 : A_1 \vdash E_1 : A \quad \Delta\,;\Gamma, x_2 : A_2 \vdash E_2 : A}{\Delta\,;\Gamma \vdash \mathtt{case}(E, x_1{:}A_1.E_1, x_2{:}A_2.E_2) : A}\ \texttt{oftp-case}$$

$$\frac{\Delta \vdash \forall_K\, A :: TYPE \quad \Delta, u :: K\,;\Gamma \vdash E : A}{\Delta\,;\Gamma \vdash \Lambda\, u{::}K.\, E : \forall_K\, (\lambda_c\, u{::}K.\, A)}\ \texttt{oftp-Fn} \qquad \frac{\Delta\,;\Gamma \vdash E : \forall_K\, B \quad \Delta \vdash C :: K}{\Delta\,;\Gamma \vdash E[C] : B\ C}\ \texttt{oftp-App}$$

$$\frac{\Delta \vdash C :: K \quad \Delta\,;\Gamma \vdash E : A\ C \quad \Delta \vdash A :: K \to_k TYPE}{\Delta\,;\Gamma \vdash \mathtt{pack}[A](C, E) : \exists_K\, A}\ \texttt{oftp-pack}$$

$$\frac{\Delta\,;\Gamma \vdash E_1 : \exists u{::}K.\, A \quad \Delta, u :: K\,;\Gamma, x : A\ u \vdash E_2 : B \quad \Delta \vdash B :: TYPE}{\Delta\,;\Gamma \vdash \mathtt{unpack}[B](E_1, u{::}K.x{:}(A\ u).E_2) : B}\ \texttt{oftp-unpack}$$

$$\frac{}{\Delta\,;\Gamma \vdash () : \mathtt{unit}}\ \texttt{oftp-empty-tuple} \qquad \frac{\Delta\,;\Gamma \vdash E : \mathtt{void}}{\Delta\,;\Gamma \vdash \mathtt{abort}[A]\, E : A}\ \texttt{oftp-abort}$$

$$\frac{}{\Delta\,;\Gamma \vdash \mathtt{zero} : \mathtt{nat}\,(z)}\ \texttt{oftp-zero} \qquad \frac{\Delta\,;\Gamma \vdash E : \mathtt{nat}\,(I)}{\Delta\,;\Gamma \vdash \mathtt{succ}[I]\, E : \mathtt{nat}\,(s\ I)}\ \texttt{oftp-succ}$$

$$\frac{\begin{array}{c}\Delta, \mathtt{i} :: \mathtt{NAT} \vdash \mathtt{A}\ \mathtt{type} \quad \Delta\,;\,\Gamma \vdash \mathtt{E} : \mathtt{nat}\,(\mathtt{I}) \\ \Delta\,;\,\Gamma \vdash \mathtt{E}_1 : [\mathtt{z}/\mathtt{i}]\mathtt{A} \\ \Delta, \mathtt{i}' :: \mathtt{NAT}\,;\,\Gamma, \mathtt{n}' : \mathtt{nat}\,(\mathtt{i}') \vdash \mathtt{E}_2 : [\mathtt{s}\ \mathtt{i}'/\mathtt{i}]\mathtt{A}\end{array}}{\Delta \vdash \mathtt{natcase}[\mathtt{i.A}](\mathtt{E}, \mathtt{E}_1, \mathtt{i}'.\mathtt{n}'.\mathtt{E}_2) :: [\mathtt{I}/\mathtt{i}]\mathtt{A}}\ \texttt{oftp-natcase}$$

$$\frac{}{\Delta\,;\,\Gamma \vdash \mathtt{nil} : \mathtt{list}\,(\mathtt{z})}\ \texttt{oftp-nil} \qquad \frac{\Delta\,;\,\Gamma \vdash \mathtt{E}_1 : \exists \mathtt{n}{::}\mathtt{N}.\,\mathtt{nat}\,(\mathtt{n}) \quad \Delta\,;\,\Gamma \vdash \mathtt{E}_2 : \mathtt{list}\,(\mathtt{I})}{\Delta\,;\,\Gamma \vdash \mathtt{cons}[\mathtt{I}]\ \mathtt{E}_1\ \mathtt{E}_2 : \mathtt{list}\,(\mathtt{s}\ \mathtt{I})}\ \texttt{oftp-cons}$$

$$\frac{\begin{array}{c}\Delta, \mathtt{i} :: \mathtt{NAT} \vdash \mathtt{A}\ \mathtt{type} \quad \Delta\,;\,\Gamma \vdash \mathtt{E} : \mathtt{list}\,(\mathtt{I}) \quad \Delta\,;\,\Gamma \vdash \mathtt{E}_1 : [\mathtt{z}/\mathtt{i}]\mathtt{A} \\ \Delta, \mathtt{i}' :: \mathtt{NAT}\,;\,\Gamma, \mathtt{hd} : \exists \mathtt{u}{::}\mathtt{N}.\,\mathtt{nat}\,(\mathtt{u}), \mathtt{tl} : \mathtt{nat}\,(\mathtt{i}') \vdash \mathtt{E}_2 : [\mathtt{s}\ \mathtt{i}'/\mathtt{i}]\mathtt{A}\end{array}}{\Delta\,;\,\Gamma \vdash \mathtt{listcase}[\mathtt{i.A}](\mathtt{E}, \mathtt{E}_1, \mathtt{hd.tl.i}'.\mathtt{E}_2) : [\mathtt{I}/\mathtt{i}]\mathtt{A}}\ \texttt{oftp-listcase}$$

$$\frac{\Delta, \mathtt{i} :: \mathtt{NAT} \vdash \mathtt{A}\ \mathtt{type} \quad \Delta \vdash \mathtt{I} :: \mathtt{NAT} \quad \Delta\,;\,\Gamma \vdash \mathtt{E}_1 : [\mathtt{z}/\mathtt{i}]\mathtt{A} \quad \Delta, \mathtt{i}' :: \mathtt{NAT}\,;\,\Gamma \vdash \mathtt{E}_2 : [\mathtt{s}\ \mathtt{i}'/\mathtt{i}]\mathtt{A}}{\Delta\,;\,\Gamma \vdash \mathtt{NATcase}[\mathtt{i.A}](\mathtt{I}, \mathtt{E}_1, \mathtt{i}'.\mathtt{E}_2) : [\mathtt{I}/\mathtt{i}]\mathtt{A}}\ \texttt{oftp-NATcase}$$

$$\frac{\begin{array}{c}\Delta, \mathtt{i} :: \mathtt{N}, \mathtt{j} :: \mathtt{N}, \mathtt{p} :: \mathtt{EQ}_\mathtt{N}(\mathtt{i}, \mathtt{j}) \vdash \mathtt{A} :: \mathtt{TYPE} \\ \Delta \vdash \mathtt{C} :: \mathtt{EQ}_\mathtt{N}(\mathtt{I}, \mathtt{J}) \\ \Delta\,;\,\Gamma \vdash \mathtt{E}_1 : [\mathtt{eqn\_zz}/\mathtt{p}][\mathtt{z}/\mathtt{j}][\mathtt{z}/\mathtt{i}]\mathtt{A} \\ \Delta, \mathtt{i} :: \mathtt{N}, \mathtt{j} :: \mathtt{N}, \mathtt{p} :: \mathtt{EQ}_\mathtt{N}(\mathtt{i}, \mathtt{j})\,;\,\Gamma \vdash \mathtt{E}_2 : [\mathtt{eqn\_ss}(\mathtt{i}, \mathtt{j}, \mathtt{p})/\mathtt{p}][\mathtt{s}\ \mathtt{j}/\mathtt{j}][\mathtt{s}\ \mathtt{i}/\mathtt{i}]\mathtt{A}\end{array}}{\Delta \vdash \mathtt{EQ}_\mathtt{N}\mathtt{case}[\mathtt{i.j.p.A}](\mathtt{C}, \mathtt{E}_1, \mathtt{i.j.p.E}_2) :: [\mathtt{C}/\mathtt{p}][\mathtt{J}/\mathtt{j}][\mathtt{I}/\mathtt{i}]\mathtt{A}}\ \texttt{oftp-EQNcase}$$

5.2 Dynamic Semantics

The dynamic semantics are mostly standard. The elimination forms for constructors rely on a notion of weak head reduction to reduce the scrutinized constructor to an introduction form.

$$\boxed{\mathtt{C} \xrightarrow{\text{whr}} \mathtt{C}'}$$

$$\frac{\mathtt{C}_1 \xrightarrow{\text{whr}} \mathtt{C}_1'}{\mathtt{C}_1\ \mathtt{C}_2 \xrightarrow{\text{whr}} \mathtt{C}_1'\ \mathtt{C}_2}\ \texttt{whr-app-1} \qquad \frac{}{(\lambda_\mathtt{c}\ \mathtt{u}{::}\mathtt{K2}.\ \mathtt{C})\ \mathtt{C2} \xrightarrow{\text{whr}} [\mathtt{C2}/\mathtt{u}]\mathtt{C}}\ \texttt{whr-app-beta}$$

$$\frac{\mathtt{I} \xrightarrow{\text{whr}} \mathtt{I}'}{\mathtt{NATrec}[\mathtt{u.K}](\mathtt{I}, \mathtt{C}_\mathtt{z}, \mathtt{i}'.\mathtt{r}.\mathtt{C}_\mathtt{s}) \xrightarrow{\text{whr}} \mathtt{NATrec}[\mathtt{u.K}](\mathtt{I}', \mathtt{C}_\mathtt{z}, \mathtt{i}'.\mathtt{r}.\mathtt{C}_\mathtt{s})}\ \texttt{whr-natrec-num}$$

$$\frac{}{\mathtt{NATrec}[\mathtt{u.K}](\mathtt{z}, \mathtt{C}_\mathtt{z}, \mathtt{i}'.\mathtt{r}.\mathtt{C}_\mathtt{s}) \xrightarrow{\text{whr}} \mathtt{C}_\mathtt{z}}\ \texttt{whr-natrec-beta-z}$$

$$\frac{}{\mathtt{NATrec}[\mathtt{u.K}](\mathtt{s}\ \mathtt{I}, \mathtt{C}_\mathtt{z}, \mathtt{i}'.\mathtt{r}.\mathtt{C}_\mathtt{s}) \xrightarrow{\text{whr}} [\mathtt{NATrec}[\mathtt{u.K}](\mathtt{I}, \mathtt{C}_\mathtt{z}, \mathtt{i}'.\mathtt{r}.\mathtt{C}_\mathtt{s})/\mathtt{r}][\mathtt{I}/\mathtt{i}']\mathtt{C}_\mathtt{s}}\ \texttt{whr-natrec-beta-s}$$

$$\frac{\mathtt{P} \xrightarrow{\text{whr}} \mathtt{P}'}{\mathtt{EQ}_\mathtt{N}\mathtt{rec}[\mathtt{i.j.p.K}](\mathtt{P}, \mathtt{C}_\mathtt{zz}, \mathtt{i.j.p.r.C}_\mathtt{ss}) \xrightarrow{\text{whr}} \mathtt{EQ}_\mathtt{N}\mathtt{rec}[\mathtt{i.j.p.K}](\mathtt{P}', \mathtt{C}_\mathtt{zz}, \mathtt{i.j.p.r.C}_\mathtt{ss})}\ \texttt{whr-eqnrec-proof}$$

$$\frac{}{\mathtt{EQ_N rec[i.j.p.K](eqn\_zz, C_{zz}, i.j.p.r.C_{ss})} \xrightarrow{\text{whr}} \mathtt{C_{zz}}}\ \texttt{whr-eqnrec-beta-zz}$$

$$\frac{}{\mathtt{EQ_N rec[i.j.p.K](eqn\_ss(I, J, P), C_{zz}, i.j.p.r.C_{ss})} \xrightarrow{\text{whr}} \mathtt{[EQ_N rec[i.j.p.K](P, C_{zz}, i.j.p.r.C_{ss})/r][P/p][J/j][I/i]C_{ss}}}\ \texttt{whr-eqnrec-beta-ss}$$

$\boxed{\mathtt{E\ value}}$

The value judgement is defined by a subsyntax (that is, `E value` is derivable if `E` is also produced by the following grammar). In the grammar, the metavariable `E` still refers to arbitrary terms.

$$\begin{array}{lcl} \mathtt{V} & ::= & \lambda\,\mathtt{x{:}A.\,E} \mid \mid (\mathtt{V_1, V_2}) \mid \mathtt{inl[A]\ V} \mid \mathtt{inr[A]\ V} \mid \Lambda\,\mathtt{u{::}K.\,E} \mid \mathtt{pack[A](C, V)} \mid () \\ & & \mid \mathtt{zero} \mid \mathtt{succ[I]\ V} \mid \mathtt{nil} \mid \mathtt{cons[I]\ V_1\ V_2} \end{array}$$

$\boxed{\mathtt{E} \mapsto \mathtt{E'}}$

Reference to a term produced by `V` is shorthand for an extra premise of `V value`.

$$\frac{\mathtt{E_1} \mapsto \mathtt{E_1'}}{\mathtt{E_1\ E_2} \mapsto \mathtt{E_1'\ E_2}}\ \texttt{step-app-1} \qquad \frac{\mathtt{E_2} \mapsto \mathtt{E_2'}}{\mathtt{V_1\ E_2} \mapsto \mathtt{V_1\ E_2'}}\ \texttt{step-app-2}$$

$$\frac{}{(\lambda\,\mathtt{x{:}A.\,E})\ \mathtt{V_2} \mapsto \mathtt{[V_2/x]E}}\ \texttt{step-app-beta} \qquad \frac{}{\mathtt{fix\,x{:}A.\,E} \mapsto \mathtt{[fix\,x{:}A.\,E/x]E}}\ \texttt{step-fix}$$

$$\frac{\mathtt{E_1} \mapsto \mathtt{E_1'}}{(\mathtt{E_1, E_2}) \mapsto (\mathtt{E_1', E_2})}\ \texttt{step-pair-1} \qquad \frac{\mathtt{E_2} \mapsto \mathtt{E_2'}}{(\mathtt{V_1, E_2}) \mapsto (\mathtt{V_1, E_2'})}\ \texttt{step-pair-2}$$

$$\frac{\mathtt{E} \mapsto \mathtt{E'}}{\mathtt{fst\ E} \mapsto \mathtt{fst\ E'}}\ \texttt{step-fst} \qquad \frac{}{\mathtt{fst}\ (\mathtt{V_1, V_2}) \mapsto \mathtt{V_1}}\ \texttt{step-fst-beta}$$

$$\frac{\mathtt{E} \mapsto \mathtt{E'}}{\mathtt{snd\ E} \mapsto \mathtt{snd\ E'}}\ \texttt{step-snd} \qquad \frac{}{\mathtt{snd}\ (\mathtt{V_1, V_2}) \mapsto \mathtt{V_2}}\ \texttt{step-snd-beta}$$

$$\frac{\mathtt{E} \mapsto \mathtt{E'}}{\mathtt{inl[A]\ E} \mapsto \mathtt{inl[A]\ E'}}\ \texttt{step-inl} \qquad \frac{\mathtt{E} \mapsto \mathtt{E'}}{\mathtt{inr[A]\ E} \mapsto \mathtt{inr[A]\ E'}}\ \texttt{step-inr}$$

$$\frac{\mathtt{E} \mapsto \mathtt{E'}}{\mathtt{case(E, x{:}A.E_l, y{:}B.E_r)} \mapsto \mathtt{case(E', x{:}A.E_l, y{:}B.E_r)}}\ \texttt{step-case}$$

$$\frac{}{\mathtt{case(inl[V]\ , x{:}A.E_l, y{:}B.E_r)} \mapsto \mathtt{[V/x]E_l}}\ \texttt{step-case-beta-l}$$

$$\frac{}{\mathtt{case(inr[V]\ , x{:}A.E_l, y{:}B.E_r)} \mapsto \mathtt{[V/y]E_r}}\ \texttt{step-case-beta-r}$$

$$\frac{\mathtt{E_1} \mapsto \mathtt{E_1'}}{\mathtt{E_1[C]} \mapsto \mathtt{E_1'[C]}}\ \texttt{step-st-app} \qquad \frac{}{(\Lambda\,\mathtt{u{::}K.\,E})\ \mathtt{C} \mapsto \mathtt{[C/u]E}}\ \texttt{step-st-app-beta}$$

$$\frac{\mathtt{E} \mapsto \mathtt{E'}}{\mathtt{pack[A](C, E)} \mapsto \mathtt{pack[A](C, E')}}\ \texttt{step-pack}$$

$$\frac{\mathtt{E_1} \mapsto \mathtt{E_1'}}{\mathtt{unpack[B](E_1, u{::}K.x{:}(A\ u).E_2)} \mapsto \mathtt{unpack[B](E_1', u{::}K.x{:}(A\ u).E_2)}}\ \texttt{step-unpack}$$

$$\frac{}{\mathtt{unpack[B](pack[A_1](C,V),u{::}K.x{:}(A_2\ u).E_2)} \mapsto \mathtt{[V/x][C/u]E_2}}\ \texttt{step-unpack-beta}$$

$$\frac{\mathtt{E} \mapsto \mathtt{E'}}{\mathtt{abort[A]\ E} \mapsto \mathtt{abort[A]\ E'}}\ \texttt{step-abort} \qquad \frac{\mathtt{E} \mapsto \mathtt{E'}}{\mathtt{succ[I]\ E} \mapsto \mathtt{succ[I]\ E'}}\ \texttt{step-succ}$$

$$\frac{\mathtt{E} \mapsto \mathtt{E'}}{\mathtt{natcase[i.A](E,E_1,i'.n'.E_2)} \mapsto \mathtt{natcase[i.A](E',E_1,i'.n'.E_2)}}\ \texttt{step-natcase}$$

$$\frac{}{\mathtt{natcase[i.A](zero,E_1,i'.n'.E_2)} \mapsto \mathtt{E_1}}\ \texttt{step-natcase-beta-z}$$

$$\frac{}{\mathtt{natcase[i.A](succ[I]\ V,E_1,i'.n'.E_2)} \mapsto \mathtt{[V/n][I/i']E_2}}\ \texttt{step-natcase-beta-s}$$

$$\frac{\mathtt{E_1} \mapsto \mathtt{E_1'}}{\mathtt{cons[I]\ E_1\ E_2} \mapsto \mathtt{cons[I]\ E_1'\ E_2}}\ \texttt{step-cons-1} \qquad \frac{\mathtt{E_2} \mapsto \mathtt{E_2'}}{\mathtt{cons[I]\ V_1\ E_2} \mapsto \mathtt{cons[I]\ V_1\ E_2'}}\ \texttt{step-cons-2}$$

$$\frac{\mathtt{E} \mapsto \mathtt{E'}}{\mathtt{listcase[i.A](E,E_1,h.i.tl.E_2)} \mapsto \mathtt{listcase[i.A](E,E_1,h.i.tl.E_2)}}\ \texttt{step-listcase}$$

$$\frac{}{\mathtt{listcase[i.A](nil,E_1,h.i.tl.E_2)} \mapsto \mathtt{E_1}}\ \texttt{step-listcase-beta-nil}$$

$$\frac{}{\mathtt{listcase[i.A](cons[I]\ V_1\ V_2,E_1,h.i.t.E_2)} \mapsto \mathtt{[V_2/t][I/i][V_1/h]E_2}}\ \texttt{step-listcase-beta-cons}$$

$$\frac{\mathtt{C} \xrightarrow{\text{whr}} \mathtt{C'}}{\mathtt{NATcase[i.A](C,E_1,i'.E_2)} \mapsto \mathtt{NATcase[i.A](C',E_1,i'.E_2)}}\ \texttt{step-NATcase}$$

$$\frac{}{\mathtt{NATcase[i.A](z,E_1,i'.E_2)} \mapsto \mathtt{E_1}}\ \texttt{step-NATcase-beta-z}$$

$$\frac{}{\mathtt{NATcase[i.A](s\ I,E_1,i'.E_2)} \mapsto \mathtt{[I/i']E_2}}\ \texttt{step-NATcase-beta-s}$$

$$\frac{\mathtt{C} \xrightarrow{\text{whr}} \mathtt{C'}}{\mathtt{EQ_Ncase[i.j.p.A](C,E_1,i.j.p.E_2)} \mapsto \mathtt{EQ_Ncase[i.j.p.A](C',E_1,i.j.p.E_2)}}\ \texttt{step-EQNcase}$$

$$\frac{}{\mathtt{EQ_Ncase[i.j.p.A](eqn\_zz,E_1,i.j.p.E_2)} \mapsto \mathtt{E_1}}\ \texttt{step-EQNcase-beta-zz}$$

$$\frac{}{\mathtt{EQ_Ncase[i.j.p.A](eqn\_ss(I,J,P),E_1,i.j.p.E_2)} \mapsto \mathtt{[P/p][J/j][I/i]E_2}}\ \texttt{step-EQNcase-beta-ss}$$

5.3 Discussion of the Meta-theory

We have proved that our calculus is type safe and that it admits decidable type checking. Because our language has a complex notion of definitional equality, a direct proof of progress and preservation for the declarative rules presented above runs into trouble in a couple of places. In the `step-app-beta` case of preservation, inversions give $\Delta \vdash \mathtt{A}_2 \rightarrow \mathtt{A} \equiv \mathtt{B}_2 \rightarrow \mathtt{B} :: \mathtt{TYPE}$, and from this it is necessary to conclude that $\Delta \vdash \mathtt{A} \equiv \mathtt{B} :: \mathtt{TYPE}$. In the presence of transitivity of constructor equality (`deq-cn-trans`) and β-reduction for constructor-level functions (`deq-cn-app-beta`), this entailment is not obvious. The canonical forms lemmas necessary for progress also depend on analyzing definitional equality, as the type conversion rule (`oftp-deq`) is not syntax-directed: for example, depending on what definitional equality is, any value—not just pairs—could have type $\mathtt{A}_1 \times \mathtt{A}_2$.

To circumvent these difficulties, we have specified an independent algorithmic formulation of equality and typing and shown that it is equivalent to the declarative version. This yields not only the lemmas necessary for type safety, but also an effective algorithm for type checking. Indeed, because the algorithmic rules are well-moded, Twelf's logic programming operational semantics can run them effectively. Since our kind and constructor level is an extension of the types and objects of LF, it is not surprising that we were able to follow the algorithmic equality technique pioneered by Harper and Pfenning [25] rather closely. Their work gives an algorithm for deciding $\beta\eta$-equality of functions; in the present work, we have extended their technique to an algorithm for deciding β-only equality for $\mathtt{NAT}$ and $\mathtt{EQ}_\mathtt{N}(\mathtt{I}, \mathtt{J})$.

We have formalized much of the meta-theory of our language using Twelf's meta-theorem checker. Unfortunately, Twelf's meta-theorem apparatus does not currently support logical relations directly; thus, while we have formalized many of the lemmas leading up to it, the logical relations argument for completeness of algorithmic equality is on paper. Porting lemmas between paper and Twelf is justified by the *adequacy* theorems of the LF methodology, which establish a bijection between object-language syntax/judgements and canonical terms of particular types in LF. The full meta-theory is presented in Appendix B.

6 Related Work

In the following section, we compare other languages' mechanisms for defining, computing with, and reasoning about indices with ours; we do not discuss other novel features or interesting applications of these existing languages here. Many of these languages automate reasoning about indices, which we leave to future work.

Constructive Type Theory The concept of a dependent type is rooted in constructive type theory, a foundational framework for constructive mathematics that makes explicit the computational content of proofs. The principal influences on the present work are deBruijn's AUTOMATH project [39], which called attention to the central role of dependent types for formalized reasoning; Martin-Löf's seminal work on constructive type theory [33, 34, 35], which presented the first comprehensive type theory adequate for constructive mathematics; the NuPRL Project [12], which built the first implementation of a tactic-based interactive proof development system for type theory; and the Calculus of Constructions [14, 30], which explored an impredicative type theory extending higher-order logic.

Epigram Altenkirch, McBride, and McKinna's Epigram [36, 37, 3] is an impressive attempt to integrate dependent types into a practical programming language. Their design is based closely on the foundational constructive type theories (notably Luo's UTT framework [31]). Rather than employing a phase distinction, Epigram insists that all well-typed programs terminate and disallows computational effects (though the authors speculate on using a subsyntax or a monad to allow them [3]). The insistence on termination is sharply at odds with most other functional languages, which permit unbounded recursion. Our approach, in contrast, is designed at the outset to accommodate non-termination and other effects. The Epigram group has developed several techniques for practical dependent programming. For example, McBride's techniques [36] elaborate a concise pattern matching notation [13] to elimination forms like those we have used in this paper. In Section 4.4, we described Brady's compilation techniques that mitigate the run-time costs of dependent programming [6]. We may be able to apply these techniques to our language.

Cayenne Augustsson's Cayenne [4] is another recent proposal to integrate dependent types into a practical programming language. Like Epigram, Cayenne permits types to contain all programs, imposing no phase distinction. However, because Cayenne allows general recursion (but no other effects) and, moreover, allows non-terminating terms to appear in types, type checking is undecidable. Their approach is simply to ensure soundness of any equational reasoning (so, for example, a divergent expression cannot be deemed equal to a convergent expression) and permit the type checker to fail in cases where equations cannot be resolved after a certain number of reductions. Such an approach to type checking can be unpredictable: the programmer has to guess when an equality will be evident in few enough steps. Restricting the compile-time data to a language where equality is decidable avoids this problem.

λ_i^{ML} Harper and Morissett's λ_i^{ML} [24] supports intensional type analysis using two elimination forms for the constructors of kind TYPE: the constructor-level `Typerec` and the term-level `typecase`. Our `NATrec`, $\mathtt{EQ_N}$`rec`, `NATcase`, and $\mathtt{EQ_N}$`case` are analogues of these constructs for other kinds. For example, defining a type by induction on indices is analogous to the uses of `Typerec` in Harper and Morissett's work. Unlike λ_i^{ML}, our calculus does not include an elimination form for the kind TYPE itself.

LX The `typecase` construct of λ_i^{ML} allows run-time analysis of a language's types. However, when a compiler is translating a λ_i^{ML} program into an intermediate language that supports only analysis of its own types, the program must be rewritten to instead case-analyze the types of the intermediate language. Unfortunately, it is often difficult and sometimes impossible to rewrite the program in such a manner. LX [15] was designed to support run-time analysis of the original source language types in the compiler's intermediate languages. In the paper, inductive kinds are used to define (what we would call) the index domain of source language types; these inductive kinds could also be used to define index domains such as NAT. LX supports run-time case analysis of constructor-level sums via a construct called `ccase`; our `NATcase` and $\mathtt{EQ_N}$`case` are analogous. However, whereas our constructor-level is dependently typed, LX's constructor level is simply-typed, so one cannot use inductive families of kinds (for example, our $\mathtt{EQ_N}(\mathtt{I}, \mathtt{J})$) to represent propositions.

DML, Zenger's Indexed Types, and Extensions In DML [62, 56] and some extensions thereof (for example, Xi's ATS [59] extends DML with some imperative [65] and object-oriented [7] features; Dunfield and Pfenning combine DML-style dependent types with `datasort` refinements [18]), equality of indices is decided by a constraint solver. As we discussed in Section 1, this does not scale to programmer-defined index domains without some additional mechanism. Zenger's indexed types [63] are similar to DML—a language designer fixes the index domains and a decision procedure for them.

Programming with Proofs in ATS Chen and Xi have recently extended ATS to address some of the limitations of the DML-style framework [9]. On the surface, their work appears very similar to ours: their indices are represented as compile-time data; one reasons about indices using compile-time inductive families as propositions inhabited by explicit proofs. However, there are significant differences between their proposal and ours. First, their calculus does not admit index-level functions or elimination forms for indices and proofs (e.g., our NATrec and EQ_Nrec). Instead, a programmer must use the proposition mechanism to represent these functions relationally. For example, where in our calculus a programmer defines the index-level function plus by induction, in theirs he would inductively define a proposition Plus $(\mathtt{i}, \mathtt{j}, \mathtt{k})$ that relates two natural numbers to their sum. Instead of the type list (plus i j), he would have list (k) such that Plus $(\mathtt{i}, \mathtt{j}, \mathtt{k})$ is true. Second, their calculus does not admit run-time computation with indices and proofs.

Our resulting languages are quite different, and there are trade-offs between our approaches. On the one hand, because Chen and Xi's calculus does not allow inductive functions on indices, there is less need for a mechanism for retyping terms based on proofs of index equality. For example, to handle the commutativity of addition example in their calculus, it suffices to give the proof that Plus $(\mathtt{i}, \mathtt{j}, \mathtt{k})$ implies Plus $(\mathtt{j}, \mathtt{i}, \mathtt{k})$; the actual index in the type list (k) remains unchanged. Also, because their calculus does not allow run-time computation with static data, it is possible to give a complete erasure of indices and proofs.

On the other hand, the constructor- and term-level elimination forms for indices and proofs in our calculus are general and useful:

- By representing index-level operations as inductive functions whose computational behavior is part of definitional equality, our calculus automates some reasoning about indices. Moreover, unlike a constraint solver treating certain index operations specially, β-equality for induction operators scales to any index domain defined using an inductive kind. In contrast, defining functions relationally using the proposition mechanism forces a programmer to explicitly prove these equalities. For example, contrast our implementation of append in Section 4 with Chen and Xi's concat in their Figure 11: in ours, there is no need for proofs, as the index reasoning is handled entirely by definitional equality. There is a syntactic cost to manipulating proofs, especially because working with existential packages of proofs and terms requires let-binding each intermediate step of the computation.

- The constructor-level elimination operators for indices and proofs allow a programmer to define a type by induction on indices or proofs. Doing so is useful, for example, for exploiting index information to write functions in a manifestly total manner (recall the definition of head above). Because Chen and Xi's calculus does not allow elimination forms, defining a type by induction on indices is impossible.

- Run-time elimination forms allow proofs to be used to retype terms. While the lack of index-level elimination forms in Chen and Xi's calculus obviates many uses of retyping, it does not eliminate them all. When functions are represented relationally, one must sometimes provide separate evidence that they are in fact functions. For example, given Plus $(\mathtt{i}, \mathtt{j}, \mathtt{k})$ and Plus $(\mathtt{i}, \mathtt{j}, \mathtt{k}')$, it requires a separate proof to know that k and $\mathtt{k}'$ are actually equal. Unfortunately, because Chen and Xi's calculus does not allow run-time elimination forms for proofs, it is unclear how such a proof could be used to retype a list (k) to a list $(\mathtt{k}')$. One solution might be to build in a notion of propositional equality whose only proof is reflexivity, as described in Section 2; because reflexivity needs no run-time action, the elimination construct for this proof might still be compatible erasing all compile-time data.

- Run-time computation over indices prevents a a programmer from having to thread both constructor-level and term-level copies of the same data through the program. For example, in Chen and Xi's calculus, nth must be abstracted over both a NAT and a nat (i), whereas in our calculus the function can be written by case-analyzing the NAT directly. Altenkirch et al. [3] described this problem while comparing indexed types to Epigram's dependent types; our calculus shows that it is not a fundamental limitation of types indexed by compile-time data.

Moreover, as we mentioned in Section 4.4, there is hope for supporting these constructs with reasonable run-time costs without adhering to a complete erasure of indices.

Another contrast between our calculus and Chen and Xi's presentation of ATS is that much of their language does not seem to be formally defined and studied. First, while they use index-level functions whose equality must at least include β for their examples to type check, their core calculus does not include them. Second, they do not show how to compile programmer-defined index domains to their calculus—though, since they do not provide elimination constructs for indices, it should be possible to represent them simply as additional constants. Most significantly, they do not show how to support the inductive families that they use as propositions. While their examples seem to require typing rules for `case` that propagate index information, their core calculus does not treat `case`. Similarly, it is unclear how their calculus ensures exhaustiveness of pattern matching in proof-level functions.

Explicit proofs of type equality in Haskell Several papers have explored applications of using values as proofs of type equality in Haskell. This idea was pioneered by Weirich [53, 54], who defined the type of proofs that type `A` equals type `B` as

$$\mathtt{EQ}_{\mathtt{TYPE}}(\mathtt{A},\mathtt{B}) = \forall\, \mathtt{f}{::}\mathtt{TYPE} \rightarrow \mathtt{TYPE}.\, \mathtt{f}\ \mathtt{A} \rightarrow \mathtt{f}\ \mathtt{B}.$$

In Haskell, only $\mathtt{EQ}_{\mathtt{TYPE}}(\mathtt{A},\mathtt{A})$ is inhabited by a terminating term, and then the only member is the identity function. To cast a term using a proof, the programmer instantiates the polymorphic function and applies it. This notion of an equality proof has been used to implement a type-safe `cast` and type `dynamic` [53, 54, 10, 5] as well as polytypic programming [10]. However, it is problematic in two ways. First and foremost, Haskell is not a consistent logic—the purported proof might not terminate. In an ML-like language, we would have to contend with "proofs" that employ other effects such as mutation and I/O. Second, since the only terminating proof is the identity function, there is no observable effect of executing the casts at run-time; but since there is no way to guarantee that a proof terminates, it must be run. Retyping in our framework has a run-time action because the "equalities" witnessed by the coercions might not be the identity.

First-class phantom types and guarded recursive datatypes First-class phantom types [11] build the sort of type equality reasoning enabled by the explicit Haskell proofs mentioned above into the type checker. In particular, when specifying data constructors in a `data` declaration, the programmer can list type equalities that are necessary for an application of that constructor to be well-typed; when a term that was created with such a constructor is `case`-analyzed, the truth of its equations is assumed in typing the corresponding `case` arm; the type system uses congruence closure to determine whether the assumed facts imply that a necessary equation is true. Xi et al. [60] proposed a similar construct, guarded recursive datatypes, as an extension to SML. Because some method for deciding equations is baked into the system, it suffers from the limitations of constraint-solver-based approaches described in Section 1.

Ωmega Pasalic and Sheard's language Ωmega [41, 48] extends Haskell with first-class phantom types, programmer-defined type-level functions, and extensible kinds. As in our calculus, but in contrast to ATS, Ωmega supports index-level functions directly rather than relationally. However, while the authors discuss the need for restrictions [48], Ωmega currently does not enforce the totality and termination of type-level functions; consequently, type checking is undecidable [49]. We have restricted our type-level functions to primitive recursion to avoid this problem. Along the same lines, it is unclear if new kinds must be inductive or if arbitrary recursive kinds are allowed; in the latter case, similar problems with termination of type checking will arise.

Propositions about indices are handled in several ways in Ωmega, but none of them are quite satisfactory. First, index and type equality in Ωmega are built into the type checker using first-class phantom types. This mechanism is of course limited by whatever decision procedure is built into the language. Because Ωmega supports index-level functions, the need for additional propositional equalities that can be used to retype terms is more acute than in ATS; indeed, the authors observe the problem with commutativity of addition [49] but do not propose a solution. As a supplemental mechanism, it seems possible to prove equalities inductively by reflecting indices as run-time terms (using a form of singleton type); however, this approach admits non-terminating "proofs". In contrast, our calculus supports propositional equality as an indexed *kind*, and thus its proofs are necessarily normalizing. Finally, a recent extension to Ωmega suggests a mechanism whereby the phantom type decision procedure can be told to treat arbitrary indexed data*types* as propositions [50]. Using this mechanism, a programmer can write a term-level program whose type is then treated as a new proof rule by the internal decision procedure. However, it is unclear how the totality of such programs is ascertained and under what circumstances the decision procedure will successfully use a new rule. In our calculus, proofs inhabit a compile-time level that is restricted to terminating functions and exhaustive case-analyses; additionally, proofs are fully explicit and therefore predictable.

Finally, Ωmega does not allow computation over indices at run-time. Consequently, as in ATS, functions must be abstracted over both compile-time and run-time copies of their arguments (e.g., `nth` must take both a `NAT` and a `nat (i)`).

RSP1 RSP1 [55] supports both traditional dependent types (types contain elements of the syntactic class of run-time programs) and imperative features (in particular, hash tables); it does this by defining syntactic criteria for those terms that can appear in types. Whereas our calculus realizes the phase distinction as a separation between type constructors on the one hand and terms on the other, RSP1 erects a phase distinction between type constructors and pure on terms on the one hand and effectful terms on the other. Both of these formulations prevent effectful terms from appearing in types and both admit run-time computation with indices. However, our style of presenting the phase distinction is arguably cleaner: our types $\mathtt{A} \rightarrow \mathtt{B}$ and $\forall\, \mathtt{u}{::}\mathtt{K}.\, \mathtt{B}$ are collapsed into RSP1's single $\Pi\, \mathtt{x}{:}\mathtt{A}.\, \mathtt{B}$, but the distinction between the two is still present in their two typing rules for function application, which distinguish between applications to pure and impure arguments. Additionally, as we noted in Section 4.4, our presentation is compatible with the existing techniques for advanced module systems, which assume that the phase distinction is realized as a split between type constructors and terms.

In addition to this difference, RSP1 suffers from some of the of the same problems as other approaches. First, because proofs are represented as arbitrary terms of indexed datatypes, they may be effectful or non-terminating. Second, because RSP1 does not allow functions to appear in types, a programmer must adopt a relational approach to index functions that is similar to Chen and Xi's [9]; the same criticisms of the relational approach apply. Moreover, because index terms are also computed with at run-time, RSP1 does not provide a complete erasure of indices and proofs; this was the central benefit derived from representing functions as relations in Chen and Xi's work.

7 Conclusion

In this report, we have presented a language with types indexed by the index domain of natural numbers and rigorously developed its meta-theory. Our calculus maintains a phase distinction between compile-time data and run-time data; it treats index equalities using explicit proofs. Much of the language design is a consequence of the following decisions:

1. Indices are type constructors in an F_ω-like calculus. Index operations are represented directly as index-level functions that can be written using the inductive elimination forms for indices.

2. Inductive families of types are indexed by this compile-time data.

3. $\beta\eta$-equality functions and β-equality for inductive families of kinds are built into a notion of definitional equality that automates some reasoning about indices.

4. When these equalities are insufficient, a programmer can use explicit proofs of equality to establish properties of indices. Run-time elimination forms allow a programmer to write coercions that retype a term based on an equality proof. Other propositions could be represented as other inductive families of kinds.

   Instead of providing a run-time elimination form only for the identity proposition, we have chosen to permit run-time computation with all compile-time inductive families. This allows programs to be written by analyzing indices and proofs of arbitrary propositions; for example, in Section 4, we wrote `nth` by analyzing a compile-time number; in Appendix A, we sketch how run-time elimination forms for proofs allow retyping terms based on coarser notions of equality than syntactic identity.

5. When there is insufficient evidence for a proposition, run-time checks can be used to create proofs.

Our calculus enables programming in the style of Dependent ML [62] or languages with GADTs [48] using the standard constructs of dependent type theory. When indices are constructors, dependent function and pair types are simply standard universal and existential polymorphism. When proofs are explicit, DML's subset sorts (and, in later presentations, guard and assert types) are just quantification over proofs. The constraints generated by DML's pattern matching are accounted for using the standard elimination rules for inductive families of types.

There is much left to be done:

- In Section 4.4, we discussed several opportunities for improvement suggested by the examples.
- We must extend our language to support arbitrary inductive families of indexed types and kinds, following Dybjer's inductive families [19] and their implementation in Epigram [37].
- The standard restriction on mutable state—that the data in a `ref` cannot change type—does not make sense in our setting: a `list` (6) `ref` is not very interesting, as it can only be mutated to lists with the same length. Xi [58], Westbrook et al. [55], and Mandlebaum et al. [32] provide starting points for circumventing this restriction.

8 Acknowledgments

We thank Karl Crary, Tom Murphy VII, and Susmit Sarkar for suggesting solutions to some of the difficulties with the Twelf formalization of the meta-theory.

References

[1] `http://www.cs.cmu.edu/~drl/`.

[2] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. Steele, Jr., and S. Tobin-Hochstadt. The Fortress language specification. `http://research.sun.com/projects/plrg/`, November 2005.

[3] T. Altenkirch, C. McBride, and J. McKinna. Why dependent types matter. Draft, April 2005.

[4] L. Augustsson. Cayenne - a language with dependent types. In *International Conference on Functional Programming*, 1998.

[5] A. Baars and S. Swierstra. Typing dynamic typing. In *International Conference on Functional Programming*, 2002.

[6] E. Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, Durham University, 2005.

[7] C. Chen, R. Shi, and H. Xi. A typeful approach to object-oriented programming with multiple inheritance. In *International Symposium on Practical Aspects of Declarative Languages*, 2004.

[8] C. Chen and H. Xi. Implementing typeful program transformations. In *Workshop on Partial Evaluation and Semantics Based Program Manipulation*, 2003.

[9] C. Chen and H. Xi. Combining programming with theorem proving. In *International Conference on Functional Programming*, 2005.

[10] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Haskell Workshop*, Pittsburgh, PA, 2002.

[11] J. Cheney and R. Hinze. Phantom types. Technical Report CUCIS TR20003-1901, Cornell University, 2003.

[12] R. L. Constable et. al. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice Hall, 1986.

[13] T. Coquand. Pattern matching with dependent types. In *Types For Proofs and Programming*, 1992.

[14] T. Coquand and G. P. Huet. The calculus of constructions. *Information and Computation*, 76(2/3), 1988.

[15] K. Crary and S. Weirich. Flexible type analysis. In *International Conference on Functional Programming*, 1999.

[16] D. Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon University, 2005. CMU Technical Report CMU-CS-05-131.

[17] D. Dreyer, K. Crary, and R. Harper. A type theory for higher-order modules. In *Symposium on Principles of Programming Languages*, 2003.

[18] J. Dunfield and F. Pfenning. Tridirectional typechecking. In *Symposium on Principles of Programming Languages*, 2004.

[19] P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. *Logical Frameworks*, 1991.

[20] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[21] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[22] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1), 1993.

[23] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Symposium on Principles of Programming Languages*, 1990.

[24] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Symposium on Principles of Programming Languages*, 1995.

[25] R. Harper and F. Pfenning. On equivalence and canonical forms in the LF type theory. *Transactions on Computational Logic*, 2003.

[26] M. Hofmann. *Extensional Concepts in Intensional Type Theory*. PhD thesis, University of Edinburgh, 1995.

[27] D. Isbell and D. Savage. Mars climate orbiter failure board releases report, numerous NASA actions underway in response. NASA Press Release 99-134, 1999. `http://mars.jpl.nasa.gov/msp98/news/mco991110.html`.

[28] A. Kennedy. Relational parametricity and units of measure. In *Symposium on Principles of Programming Languages*, 1997.

[29] M. Lillibridge and R. Harper. A type-theoretic approach to higher-order modules with sharing. In *Symposium on Principles of Programming Languages*, 1994.

[30] Z. Luo. ECC, an extended calculus of constructions. In *Logic in Computer Science*, 1989.

[31] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*, volume 11 of *International Series of Monographs on Computer Science*. Oxford University Press, 1994.

[32] Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *International Conference on Functional Programming*, 2003.

[33] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In H. Rose and J. Shepherdson, editors, *Logic Colloquium*. Elsevier, 1975.

[34] P. Martin-Löf. Constructive mathematics and computer programming. *Logic, Methodology and Philosophy of Science*, VI:153–175, 1979.

[35] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[36] C. McBride. *Dependently Typed Functional Programs and Their Proofs*. PhD thesis, University of Edinburgh, 2000.

[37] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 15(1), January 2004.

[38] R. Milner, M. Tofte, R. Harper, , and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[39] R. Nederpelt, J. Geuvers, and R. de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1994.

[40] B. Nordström, K. Peterson, and J. Smith. *Programming in Martin-Löf's Type Theory, an Introduction*. Clarendon Press, 1990.

[41] E. Pasalic. *The Role of Type Equality in Meta-Programming*. PhD thesis, Oregon Health & Science University, OGI School of Science & Engineering, 2004.

[42] S. Peyton Jones, G. Washburn, and S. Weirich. Wobbly types: type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, University of Pennsylvania, 2005.

[43] F. Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Symposium on Logic in Computer Science*, 2001.

[44] F. Pfenning and C. Schrmann. System description: Twelf - a meta-logical framework for deductive systems. In *International Conference on Automated Deduction*, 1999.

[45] B. C. Pierce and D. N. Turner. Local type inference. In *Symposium on Principles of Programming Languages*, 1998.

[46] F. Pottier and Y. Régis-Gianas. Stratified type inference for generalized algebraic data types. In *Symposium on Principles of Programming Languages*, 2006.

[47] S. Sarkar. A cost-effective foundational certified code system. Thesis Proposal, Carenegie Mellon University, 2005.

[48] T. Sheard. Languages of the future. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[49] T. Sheard. Ωmega user's guide revision 1.1. Available from `http://www.cs.pdx.edu/~sheard/Omega/`, May 2005.

[50] T. Sheard. Putting Curry-Howard to work. In *Haskell Workshop*, 2005.

[51] V. Simonet and F. Pottier. Constraint-based type inference with guarded algebraic data types. Technical report, INRIA, 2003.

[52] C. A. Stone and R. Harper. Extensional equivalence and singleton types. *Transactions on Computational Logic*, 2004.

[53] S. Weirich. Type-safe cast: functional pearl. In *International Conference on Functional Programming*, 2000.

[54] S. Weirich. Type-safe cast. *Journal of Functional Programming*, 14(6), 2004.

[55] E. Westbrook, A. Stump, and I. Wehrman. A language-based approach to functionally correct imperative programming. In *International Conference on Functional Programming*, 2005.

[56] H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

[57] H. Xi. Dependently typed data structures. In *Workshop on Algorithmic Aspects of Advanced Programming Languages*, 1999.

[58] H. Xi. Imperative programming with dependent types. In *Symposium on Logic in Computer Science*, 2000.

[59] H. Xi. Applied type system (extended abstract). In *TYPES*, 2003.

[60] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Symposium on Principles of Programming Languages*, 2003.

[61] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Conference on Programming Language Design and Implementation*, 1998.

[62] H. Xi and F. Pfenning. Dependent types in practical programming. In *Symposium on Principles of Programming Languages*, 1999.

[63] C. Zenger. *Indizierte Typen*. PhD thesis, Universit at Karlsruhe, 1998.

[64] D. Zhu and H. Xi. A typeful and tagless representation for XML documents. In *First Asian Symposium on Programming Languages and Systems*, 2003.

[65] D. Zhu and H. Xi. Safe programming with pointers through stateful views. In *International Symposium on Practical Aspects of Declarative Languages*, 2005.

A Units of Measure and Run-time Elimination Forms for Proofs

In this section, we sketch an approach for tracking units of measure in types (as in Kennedy's languages [28] and Fortress [2]). This example should be programmable in a language with programmer-defined index domains and propositions. In particular, it illustrates why such a language should support run-time elimination forms for proofs.

First, we define an index domain representing units:

$$\texttt{kind}\,\texttt{U} \;= \texttt{met} \mid \texttt{sec} \mid \texttt{U}_1 \cdot \texttt{U}_2 \mid \texttt{U}^{-1} \mid \texttt{scalar}\,(\texttt{i} :: \texttt{NAT}).$$

The possible units are meters, seconds, the product of two units, the inverse of a unit, or a dimensionless scalar. Then, we define a type of floating-point numbers indexed by units:

$$\texttt{type}\,\texttt{ufloat}\,(\texttt{u} :: \texttt{U}) \;= \texttt{quantity}[\texttt{u} :: \texttt{U}]\ \texttt{float} : \texttt{ufloat}\,(\texttt{u}).$$

Then, for example, $\texttt{quantity}[\texttt{met}]\ 4.0$ represents four meters and has type $\texttt{ufloat}\,\texttt{met}$. Now, we define operations that obey unit constraints; for example, addition is only defined for quantities with the same unit, and the unit of a multiplication is the product of the units:

$$\begin{array}{l}\texttt{uplus} : \forall\, \texttt{u}::\texttt{U}.\, \texttt{ufloat}\ \texttt{u} \times \texttt{ufloat}\ \texttt{u} \rightarrow \texttt{ufloat}\,(\texttt{u}) \\ \texttt{umult} : \forall\, \texttt{u}, \texttt{v}::\texttt{U}.\, \texttt{ufloat}\ \texttt{u} \times \texttt{ufloat}\ \texttt{v} \rightarrow \texttt{ufloat}\,(\texttt{u} \cdot \texttt{v}).\end{array}$$

These functions can be implemented by extracting the underlying `float`s, performing the equivalent operation, and then packaging the result with the correct unit. If we then made `ufloat` abstract, exposing a way to create a `ufloat` from a `float` and the primitive arithmetic operation but not a way to project out the underlying `float`, then the programmer would have no choice but to use `ufloat`s in a unit-respecting manner (as defined by the primitives).

So far, we have said nothing about the algebraic properties of units. This is problematic: for example, a programmer cannot add a velocity of type $\texttt{ufloat}\,(\texttt{met} \cdot \texttt{sec}^{-1})$ with another velocity, of type $\texttt{ufloat}\,(\texttt{met} \cdot \texttt{sec}^{-1} \cdot \texttt{sec}^{-1} \cdot \texttt{sec})$, computed from an acceleration and a time. To allow such computation, we can define a notion of propositional equality that includes these algebraic laws:

$$\begin{array}{lcl}\texttt{kind}\,\texttt{EQ}_\texttt{U}(\texttt{u} :: \texttt{U}, \texttt{v} :: \texttt{U}) & = & \texttt{refl}\ \texttt{u} :: \texttt{EQ}_\texttt{U}(\texttt{u}, \texttt{u}) \\ & \mid & \texttt{sym}\ \texttt{u}\ \texttt{v}\ (\texttt{p} :: \texttt{EQ}_\texttt{U}(\texttt{u}, \texttt{v})) :: \texttt{EQ}_\texttt{U}(\texttt{v}, \texttt{u}) \\ & \mid & \texttt{trans}\ \texttt{u}\ \texttt{v}\ \texttt{w}\ (\texttt{p12} :: \texttt{EQ}_\texttt{U}(\texttt{u}, \texttt{v}))\ (\texttt{p23} :: \texttt{EQ}_\texttt{U}(\texttt{v}, \texttt{w})) :: \texttt{EQ}_\texttt{U}(\texttt{u}, \texttt{w}) \\ & \mid & \texttt{assoc}\ \texttt{u}\ \texttt{v}\ \texttt{w} :: \texttt{EQ}_\texttt{U}(\texttt{u} \cdot (\texttt{v} \cdot \texttt{w}), (\texttt{u} \cdot \texttt{v}) \cdot \texttt{w}) \\ & \mid & \texttt{ident}\ \texttt{u} :: \texttt{EQ}_\texttt{U}(\texttt{scalar}(\texttt{s}\ \texttt{z}) \cdot \texttt{u}, \texttt{u}) \\ & \mid & \texttt{inv}\ \texttt{u} :: \texttt{EQ}_\texttt{U}(\texttt{u} \cdot \texttt{u}^{-1}, \texttt{scalar}\ (\texttt{s}\ \texttt{z})) \\ & \mid & \texttt{comm}\ \texttt{u}\ \texttt{v} :: \texttt{EQ}_\texttt{U}(\texttt{u} \cdot \texttt{v}, \texttt{v} \cdot \texttt{u}) \\ & \mid & \texttt{multCong}\ \texttt{u1}\ \texttt{v1}\ \texttt{u2}\ \texttt{v2}\ (\texttt{pu} :: \texttt{EQ}_\texttt{U}(\texttt{u1}, \texttt{u2}))\ (\texttt{pv} :: \texttt{EQ}_\texttt{U}(\texttt{v1}, \texttt{v2})) :: \texttt{EQ}_\texttt{U}(\texttt{u1} \cdot \texttt{v1}, \texttt{u2} \cdot \texttt{v2}) \\ & \mid & \texttt{invCong}\ \texttt{u}\ \texttt{v}\ (\texttt{p} :: \texttt{EQ}_\texttt{U}(\texttt{u}, \texttt{v})) :: \texttt{EQ}_\texttt{U}(\texttt{u}^{-1}, \texttt{v}^{-1}).\end{array}$$

We could then prove $\mathtt{EQ_U}(\mathtt{met}\cdot\mathtt{sec}^{-1}, \mathtt{met}\cdot\mathtt{sec}^{-1}\cdot\mathtt{sec}^{-1}\cdot\mathtt{sec})$ as follows:

$$\begin{array}{lcl}
\mathtt{p1} :: \mathtt{EQ_U}(\mathtt{met}\cdot\mathtt{sec}^{-1}, \mathtt{met}\cdot\mathtt{sec}^{-1}) & = & \mathtt{refl}\ (\mathtt{met}\cdot\mathtt{sec}^{-1}) \\
\mathtt{p2} :: \mathtt{EQ_U}(\mathtt{sec}\cdot\mathtt{sec}^{-1}, \mathtt{scalar}(\mathtt{s\ z})) & = & \mathtt{inv\ sec} \\
\mathtt{p3} :: \mathtt{EQ_U}(\mathtt{scalar}(\mathtt{s\ z}), \mathtt{sec}^{-1}\cdot\mathtt{sec}) & = & \mathtt{sym}\ (\mathtt{trans}\ (\mathtt{comm\ sec}^{-1}\ \mathtt{sec})\ \mathtt{p2}) \\
\mathtt{p4} :: \mathtt{EQ_U}(\mathtt{met}\cdot\mathtt{sec}^{-1}\cdot\mathtt{scalar}(\mathtt{s\ z}), \mathtt{met}\cdot\mathtt{sec}^{-1}\cdot\mathtt{sec}^{-1}\cdot\mathtt{sec}) & = & \mathtt{multCong\ p1\ p3} \\
\mathtt{p5} :: \mathtt{EQ_U}(\mathtt{scalar}(\mathtt{s\ z})\cdot(\mathtt{met}\cdot\mathtt{sec}^{-1}), \mathtt{met}\cdot\mathtt{sec}^{-1}) & = & \mathtt{ident}\ (\mathtt{met}\cdot\mathtt{sec}^{-1}) \\
\mathtt{p6} :: \mathtt{EQ_U}(\mathtt{met}\cdot\mathtt{sec}^{-1}, ((\mathtt{met}\cdot\mathtt{sec}^{-1})\cdot\mathtt{scalar}(\mathtt{s\ z}))) & = & \mathtt{trans}\ (\mathtt{sym\ p5})\ (\mathtt{comm}\ (\mathtt{scalar}(\mathtt{s\ z}))\ (\mathtt{met}\cdot\mathtt{sec}^{-1})) \\
\mathtt{p7} :: \mathtt{EQ_U}(\mathtt{met}\cdot\mathtt{sec}^{-1}, \mathtt{met}\cdot\mathtt{sec}^{-1}\cdot\mathtt{sec}^{-1}\cdot\mathtt{sec}) & = & \mathtt{trans\ p6\ p4}.
\end{array}$$

But, if we wish to call uplus on terms with these types, the proof is not enough: we need to use the proof to retype one of the terms. Unfortunately, it would not be type safe to adopt a run-time elimination form like subst in Section 2 that has no run-time action—when $\mathtt{EQ_U}(\mathtt{u},\mathtt{v})$ is true, ufloat (u) and ufloat (v) do not always classify the same terms. The retyping for $\mathtt{EQ_U}(\mathtt{u},\mathtt{v})$ must have a run-time action that coerces a ufloat (u) to a ufloat (v). In this case, the equalities that we have postulated are not the identity on ufloat (u), but they are the identity on the underlying float: replacing algebraically equal units does not change the magnitude of a quantity. Consequently, the correct proof action in this case does not depend on how the particular equality proof was constructed:

$$\begin{array}{l}
\mathtt{retype/U} :: \forall\,\mathtt{u},\mathtt{v}{::}\mathtt{U}.\ \forall\,\_{::}\mathtt{EQ_U}(\mathtt{u},\mathtt{v}).\ \mathtt{ufloat}(\mathtt{u}) \to \mathtt{ufloat}(\mathtt{v}) \\
\mathtt{retype/U\ u\ v}\ \_\ (\mathtt{quantity}[\mathtt{u}]\ \mathtt{x})\ =\ \mathtt{quantity}[\mathtt{v}]\ \mathtt{x}
\end{array}$$

This notion of equality captures the algebraic properties of units. However, we might want a yet coarser "equality" that relates all units of the same dimension. For example, both meters and feet represent quantities of the same dimension, length, but they differ by a factor of scale; in this case, we could define feet notationally as $3048\cdot 10000^{-1}\cdot\mathtt{met}$. We can define a proposition that relates units of the same dimension:

$$\begin{array}{lcl}
\mathtt{kind\ SAMEDIM}\ (\mathtt{u} :: \mathtt{U}, \mathtt{v} :: \mathtt{U}) & = & \mathtt{sameUnit\ u\ v}\ (\mathtt{p} :: \mathtt{EQ_U}(\mathtt{u},\mathtt{v})) :: \mathtt{SAMEDIM}\ (\mathtt{u},\mathtt{v}) \\
 & | & \mathtt{scale\ u\ v\ n}\ (\mathtt{p} :: \mathtt{EQ_U}(\mathtt{u}, \mathtt{scalar}(\mathtt{n})\cdot\mathtt{v})) :: \mathtt{SAMEDIM}\ (\mathtt{u},\mathtt{v}).
\end{array}$$

However, as NASA so infamously discovered [27], the coercions for retyping based on this proposition are not the identity on the quantity. A correct coercion must act differently for different proofs; it can be defined using run-time analysis of proofs:

```
NATtoFloat : ∀ n::N. float
NATtoFloat z = 0.0
NATtoFloat (s i) = 1.0 + (NATtoFloat i)

scaleFactor : ∀ n::N. ufloat((scalar n)^-1) = quantity[(scalar n)^-1](NATtoFloat n)
P :: Π_k n::N. Π_k v::U. EQ_U((scalar n)^-1 · ((scalar n) · v), v) = ...

retype/SAMEDIM : ∀ u, v::U. ∀ _::SAMEDIM(u, v). ufloat(u) → ufloat(v)
retype/SAMEDIM u v (sameUnit u v p) = retype/U[u][v][p]
retype/SAMEDIM u v (scale u v n p) =
  fn x : ufloat(u) =>
    let x′ : ufloat(scalar(n) · v) = retype/U[u][scalar(n) · v][p] x in
     let x″ : ufloat((scalar n)^-1 · ((scalar n) · v)) = umult[(scalar n)^-1][(scalar n) · v] (scaleFactor[n]) x′ in
       retype/U [(scalar n)^-1 · ((scalar n) · v)][v][P n v] x″.
```

In addition to case-analyzing the proof, this example requires run-time analysis of an index to compute the scale factor.[8]

[8] We defined SAMEDIM with two constructors for illustrative purposes, but just scale would have sufficed because one can always apply ident to show $\mathtt{EQ_U}(\mathtt{scalar}(\mathtt{s\ z})\cdot\mathtt{u},\mathtt{u})$. With this alternate definition, the retyping function would have only one case, but it would still be necessary to deconstruct the given proof to extract the scale factor and the proof of unit equality.

In this example, we have defined propositions representing coarser equivalence relations than syntactic identity, and we have shown how the actions of these equivalences on run-time terms can be defined by case-analyzing indices and proof at run-time. Of course, it is still the programmer's responsibility to ensure that the proposition adequately represents the notion of equality that he has in his head and that the coercions correctly witness that equality. However, once he has done so, the programmer can work at the level of abstraction afforded by the proposition. In this example, instead of manually chaining together arbitrary arithmetic operations to change units, the programmer will give the proof that the units are equivalent, the type system will check that the proof correctly mediates the units in question, and then the correct coercions for that proof can be applied.

B Full Meta-theory

B.1 Outline

In this section, we prove type safety and decidability of type checking for the declarative presentation of our language in Section 5. To do so, we first give an equivalent algorithmic formulation of the language; then, we prove type safety and decidability of type checking for the algorithmic formulation. The algorithm is based on Harper and Pfenning's treatment of LF [25], and our development follows theirs closely. In this method, definitional equality is decided by two judgements, $\Psi \vdash \mathtt{C} \Longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{K}}$ and $\Psi \vdash \mathtt{C} \longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{K}}$. The first judgement is kind-directed; the second is structural. The kind-directed part relies on the weak head reduction judgement presented in Section 5 to reduce constructors to weak head normal form. Both judgements operate over kinds with dependencies erased; this greatly simplifies showing transitivity of the algorithm.

In the present work, we have extended this technique to an algorithm for deciding β-only equality for $\mathtt{NAT}$ and $\mathtt{EQ}_{\mathtt{N}}(\mathtt{I}, \mathtt{J})$. There was one trick required in adapting the algorithm to inductive kinds. The judgement $\Psi \vdash \mathtt{C} \Longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{K}}$ is kind-directed, so there must be only one rule for each kind (exempting the weak head reduction rules). For example, at function kinds, the algorithm applies extensionality of functions. It is easy to see that this works for kinds like functions or pairs with only one introduction form; however, it was not immediately obvious how to apply it to kinds like $\mathtt{NAT}$ that have constructors of more than one shape. Our solution is as follows: the single kind-directed equality for $\mathtt{NAT}$ simply refers to a mutually-defined judgement, $\Psi \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{NAT}} \mathtt{C}'$, that handles the structural comparison of the various intro forms of kind $\mathtt{NAT}$. That is, equality at kind $\mathtt{NAT}$ is defined by a separate "horizontal" judgement that complements usual "vertical" induction over kinds that defines the algorithm. The logical relations argument used to show completeness of the algorithm must account for this fact. In our proof, the logical relations in general are defined by induction over the classifying kind; in addition, the logical relation at $\mathtt{NAT}$ is itself inductively defined by a separate rule induction. This technique is an adaption of the strong normalization proofs of Gödel's T (see Girard et al. for a presentation [21]) to our setting.

The proof is organized as follows. In Section B.2, we establish some basic lemmas about the declarative presentation. In Section B.3, we give an algorithmic version of equality and show that it is equivalent the declarative specification of definitional equality. In Section B.4, we give algorithmic versions of kinding and typing and show them equivalent to the declarative definitions. In Section B.5, we prove type safety. Finally, in Section B.6, we prove decidability of type checking. All Twelf proofs referenced here are available on the Web [1].

B.2 Basic Properties of the Declarative System

We tacitly assume that all contexts appearing in the premises of the following theorem statements are well-formed according to the definition in Section 5.

THEOREM B.1: ADEQUACY. *These lemmas refer to the LF signature that is available in the companion Twelf code [1]. An encoding function is* compositional *if it commutes with substitution.*

1. *There are compositional bijections between the following.*

Syntactic Category	***Canonical LF Terms of Type***	***in LF contexts***
$\mathtt{K}$ *with FV in* $\mathtt{u}_1 \ldots \mathtt{u}_n$	$\mathtt{kd}$	$\mathtt{u}_1 : \mathtt{cn} \ldots$
$\mathtt{C}$ *with FV in* $\mathtt{u}_1 \ldots \mathtt{u}_n$	$\mathtt{cn}$	$\mathtt{u}_1 : \mathtt{cn} \ldots$
$\mathtt{E}$ *with FV in* $\mathtt{u}_1 \ldots \mathtt{u}_n, \mathtt{x}_1 \ldots \mathtt{x}_n$	$\mathtt{tm}$	$\mathtt{u}_1 : \mathtt{cn} \ldots, \mathtt{x}_1 : \mathtt{tm} \ldots$

We use $\ulcorner \cdot \urcorner$ *and* $\llcorner \cdot \lrcorner$ *to refer to the functions witnessing any of these bijections.*

2. *There are bijections between the following.*

Derivations of	***Canon. LF Terms of Type***	***in LF contexts***
$\mathtt{u}_1 :: \mathtt{K}_1 \ldots \vdash \mathtt{K}\ \mathtt{kind}$	$\mathtt{wf\_kd}\ \ulcorner\mathtt{K}\urcorner$	$\mathtt{u}_1 : \mathtt{cn}, \mathtt{du}_1 : \mathtt{ofkd}\ \mathtt{u}_1\ \ulcorner\mathtt{K}_1\urcorner, \mathtt{dequ}_1 : \mathtt{deq\_cn}\ \mathtt{u}_1\ \mathtt{u}_1\ \ulcorner\mathtt{K}_1\urcorner \ldots$
$\mathtt{u}_1 :: \mathtt{K}_1 \ldots \vdash \mathtt{K} \equiv \mathtt{K}'\ \mathtt{kind}$	$\mathtt{deq\_kd}\ \ulcorner\mathtt{K}\urcorner \ulcorner\mathtt{K}'\urcorner$	$\mathtt{u}_1 : \mathtt{cn}, \mathtt{du}_1 : \mathtt{ofkd}\ \mathtt{u}_1\ \ulcorner\mathtt{K}_1\urcorner, \mathtt{dequ}_1 : \mathtt{deq\_cn}\ \mathtt{u}_1\ \mathtt{u}_1\ \ulcorner\mathtt{K}_1\urcorner \ldots$
$\mathtt{u}_1 :: \mathtt{K}_1 \ldots \vdash \mathtt{C} :: \mathtt{K}$	$\mathtt{ofkd}\ \ulcorner\mathtt{C}\urcorner \ulcorner\mathtt{K}\urcorner$	$\mathtt{u}_1 : \mathtt{cn}, \mathtt{du}_1 : \mathtt{ofkd}\ \mathtt{u}_1\ \ulcorner\mathtt{K}_1\urcorner, \mathtt{dequ}_1 : \mathtt{deq\_cn}\ \mathtt{u}_1\ \mathtt{u}_1\ \ulcorner\mathtt{K}_1\urcorner \ldots$
$\mathtt{u}_1 :: \mathtt{K}_1 \ldots \vdash \mathtt{C} \equiv \mathtt{C}' :: \mathtt{K}$	$\mathtt{deq\_cn}\ \ulcorner\mathtt{C}\urcorner\ \ulcorner\mathtt{C}'\urcorner \ulcorner\mathtt{K}\urcorner$	$\mathtt{u}_1 : \mathtt{cn}, \mathtt{du}_1 : \mathtt{ofkd}\ \mathtt{u}_1\ \ulcorner\mathtt{K}_1\urcorner, \mathtt{dequ}_1 : \mathtt{deq\_cn}\ \mathtt{u}_1\ \mathtt{u}_1\ \ulcorner\mathtt{K}_1\urcorner \ldots$
$\mathtt{u}_i :: \mathtt{K}_i\ ;\ \mathtt{x}_j : \mathtt{A}_j \vdash \mathtt{E} : \mathtt{A}$	$\mathtt{oftp}\ \ulcorner\mathtt{E}\urcorner \ulcorner\mathtt{A}\urcorner$	$\mathtt{u}_i : \mathtt{cn}, \mathtt{du}_i : \mathtt{ofkd}\ \mathtt{u}_i\ \ulcorner\mathtt{K}_i\urcorner, \mathtt{dequ}_i : \mathtt{deq\_cn}\ \mathtt{u}_i\ \mathtt{u}_i\ \ulcorner\mathtt{K}_i\urcorner,$ $\mathtt{x}_j : \mathtt{tm}, \mathtt{dx}_j : \mathtt{oftp}\ \mathtt{x}_j\ \ulcorner\mathtt{A}_j\urcorner$
$\mathtt{C} \xrightarrow{\text{whr}} \mathtt{C}'$, *FV in* $\mathtt{u}_1 \ldots$	$\mathtt{whr}\ \ulcorner\mathtt{C}\urcorner\ \ulcorner\mathtt{C}'\urcorner$	$\mathtt{u}_1 : \mathtt{cn} \ldots$
$\mathtt{C} \overset{\text{whr}}{\longrightarrow^*} \mathtt{C}'$, *FV in* $\mathtt{u}_1 \ldots$	$\mathtt{whrrt}\ \ulcorner\mathtt{C}\urcorner\ \ulcorner\mathtt{C}'\urcorner$	$\mathtt{u}_1 : \mathtt{cn} \ldots$
$\mathtt{E}\ \mathtt{value}$, *FV in* $\mathtt{u}_1 \ldots, \mathtt{x}_1 \ldots$	$\mathtt{value}\ \ulcorner\mathtt{E}\urcorner$	$\mathtt{u}_1 : \mathtt{cn} \ldots, \mathtt{x}_1 : \mathtt{tm} \ldots$
$\mathtt{E} \mapsto \mathtt{E}'$, *FV in* $\mathtt{u}_1 \ldots, \mathtt{x}_1 \ldots$	$\mathtt{step}\ \ulcorner\mathtt{E}\urcorner \ulcorner\mathtt{E}'\urcorner$	$\mathtt{u}_1 : \mathtt{cn} \ldots, \mathtt{x}_1 : \mathtt{tm} \ldots$

Proof. The encodings we use follow standard techniques [22]: the syntax encodings use higher-order abstract syntax, representing object-language variables with meta-language variables; the derivations and judgements are encoded using the judgements-as-types methodology. Consequently, the proofs of adequacy are also by standard means; Harper, Honsell, and Plotkin present some examples [22]. □

LEMMA B.2: SUBSTITUTION INTO A SUBSTITUTION.
If $\mathtt{v}$ *is not free in* $\mathtt{C}_2$ *then* $[\mathtt{C}_2/\mathtt{u}][\mathtt{C}_1/\mathtt{v}]\mathtt{C}$ *is* $[[\mathtt{C}_2/\mathtt{u}]\mathtt{C}_1/\mathtt{v}][\mathtt{C}_2/\mathtt{u}]\mathtt{C}$. *Under the same restrictions,* $[\mathtt{C}_2/\mathtt{u}][\mathtt{C}_1/\mathtt{v}]\mathtt{K}$ *is* $[[\mathtt{C}_2/\mathtt{u}]\mathtt{C}_1/\mathtt{v}][\mathtt{C}_2/\mathtt{u}]\mathtt{K}$. *Note: when used in this sense, "is" means syntactic identity up to* α*-conversion.*

Proof. By mutual induction on the structure of $\mathtt{C}$ and $\mathtt{K}$. In some cases, we replace equals for equals until the two sides are identical; then the equality is given by reflexivity. Reading this backward would show how to construct a derivation of equality.

- To show:
  $$[\mathtt{C}_2/\mathtt{u}][\mathtt{C}_1/\mathtt{v}]\mathtt{u} \textit{ is } [[\mathtt{C}_2/\mathtt{u}]\mathtt{C}_1/\mathtt{v}][\mathtt{C}_2/\mathtt{u}]\mathtt{u}.$$
  The LHS reduces to $[\mathtt{C}_2/\mathtt{u}]\mathtt{u}$ because $\mathtt{u}$ and $\mathtt{v}$ are different and then to $\mathtt{C}_2$ because $\mathtt{u}$ and $\mathtt{u}$ are the same. The RHS reduces to $[[\mathtt{C}_2/\mathtt{u}]\mathtt{C}_1/\mathtt{v}]\mathtt{C}_2$ because $\mathtt{u}$ and $\mathtt{u}$ are the same and then to $\mathtt{C}_2$ because $\mathtt{v}$ is not free in $\mathtt{C}_2$.

- To show:
  $$[\mathtt{C}_2/\mathtt{u}][\mathtt{C}_1/\mathtt{v}]\mathtt{v} \textit{ is } [[\mathtt{C}_2/\mathtt{u}]\mathtt{C}_1/\mathtt{v}][\mathtt{C}_2/\mathtt{u}]\mathtt{v}.$$
  The LHS reduces to $[\mathtt{C}_2/\mathtt{u}]\mathtt{C}_1$ because $\mathtt{v}$ and $\mathtt{v}$ are the same; the RHS reduces to $[[\mathtt{C}_2/\mathtt{u}]\mathtt{C}_1/\mathtt{v}]\mathtt{v}$ because $\mathtt{u}$ and $\mathtt{v}$ are different and then to $[\mathtt{C}_2/\mathtt{u}]\mathtt{C}_1$ because $\mathtt{v}$ and $\mathtt{v}$ are the same.

- To show:
$$[\mathtt{C}_2/\mathtt{u}][\mathtt{C}_1/\mathtt{v}]\mathtt{w} \textit{ is } [[\mathtt{C}_2/\mathtt{u}]\mathtt{C}_1/\mathtt{v}][\mathtt{C}_2/\mathtt{u}]\mathtt{w}.$$
Both sides reduce to `w` because the variables are different.

- To show:
$$[\mathtt{C}_2/\mathtt{u}][\mathtt{C}_1/\mathtt{v}]\mathtt{unit} \textit{ is } [[\mathtt{C}_2/\mathtt{u}]\mathtt{C}_1/\mathtt{v}][\mathtt{C}_2/\mathtt{u}]\mathtt{unit}.$$
Both sides reduce to `unit` because substitution into it is a no-op.

- To show:
$$[\mathtt{C}_2/\mathtt{u}][\mathtt{C}_1/\mathtt{v}](\mathtt{A} \rightarrow \mathtt{B}) \textit{ is } [[\mathtt{C}_2/\mathtt{u}]\mathtt{C}_1/\mathtt{v}][\mathtt{C}_2/\mathtt{u}](\mathtt{A} \rightarrow \mathtt{B}).$$
By induction,
$$[\mathtt{C}_2/\mathtt{u}][\mathtt{C}_1/\mathtt{v}]\mathtt{A} \textit{ is } [[\mathtt{C}_2/\mathtt{u}]\mathtt{C}_1/\mathtt{v}][\mathtt{C}_2/\mathtt{u}]\mathtt{A}$$
$$[\mathtt{C}_2/\mathtt{u}][\mathtt{C}_1/\mathtt{v}]\mathtt{B} \textit{ is } [[\mathtt{C}_2/\mathtt{u}]\mathtt{C}_1/\mathtt{v}][\mathtt{C}_2/\mathtt{u}]\mathtt{B}.$$
By congruence of identity,
$$[\mathtt{C}_2/\mathtt{u}][\mathtt{C}_1/\mathtt{v}]\mathtt{A} \rightarrow [\mathtt{C}_2/\mathtt{u}][\mathtt{C}_1/\mathtt{v}]\mathtt{B} \textit{ is } [[\mathtt{C}_2/\mathtt{u}]\mathtt{C}_1/\mathtt{v}][\mathtt{C}_2/\mathtt{u}]\mathtt{A} \rightarrow [[\mathtt{C}_2/\mathtt{u}]\mathtt{C}_1/\mathtt{v}][\mathtt{C}_2/\mathtt{u}]\mathtt{B}$$
Then the definition of substitution for $\rightarrow$ allows the substitution to be pulled outside on each side.

- To show:
$$[\mathtt{C}_2/\mathtt{u}][\mathtt{C}_1/\mathtt{v}](\lambda_\mathsf{c}\, \mathtt{w}{::}\mathtt{K}.\, \mathtt{C}) \textit{ is } [[\mathtt{C}_2/\mathtt{u}]\mathtt{C}_1/\mathtt{v}][\mathtt{C}_2/\mathtt{u}](\lambda_\mathsf{c}\, \mathtt{w}{::}\mathtt{K}.\, \mathtt{C}).$$
By induction,
$$[\mathtt{C}_2/\mathtt{u}][\mathtt{C}_1/\mathtt{v}]\mathtt{K} \textit{ is } [[\mathtt{C}_2/\mathtt{u}]\mathtt{C}_1/\mathtt{v}][\mathtt{C}_2/\mathtt{u}]\mathtt{K}$$
$$[\mathtt{C}_2/\mathtt{u}][\mathtt{C}_1/\mathtt{v}]\mathtt{C} \textit{ is } [[\mathtt{C}_2/\mathtt{u}]\mathtt{C}_1/\mathtt{v}][\mathtt{C}_2/\mathtt{u}]\mathtt{C}.$$
In order to apply the definition of substitution for λ, we must know that `w` is distinct from `u` and `v` and that `w` is not free in any of the substituted terms. Fortunately, this can be achieved by α-renaming the bound variable `w` to something fresh.

- All other cases are similar to the previous three. When there are no subexpressions, substitution is a no-op. Otherwise, apply induction, congruence, and the definition of substitution; in binding forms, α-renaming the bound variable to something fresh ensures that the definition can be applied.

□

LEMMA B.3: WEAKENING. *If* $\Delta, \Delta' \vdash J$ *and* $\Delta, \mathtt{u} :: \mathtt{K}, \Delta'$ *is well-formed then* $\Delta, \mathtt{u} :: \mathtt{K}, \Delta' \vdash J$.

Proof. By induction over the given derivation. Alternatively, this statement of weakening is true in LF [25], so this follows from THEOREM B.1. □

LEMMA B.4: SUBSTITUTION.

1. *If* $\Delta, \mathtt{u} :: \mathtt{K}, \Delta' \vdash J$ *and* $\Delta \vdash \mathtt{C} :: \mathtt{K}$ *then* $\Delta, [\mathtt{C}/\mathtt{u}]\Delta' \vdash [\mathtt{C}/\mathtt{u}]J$.
2. *If* $\Delta, \mathtt{u} :: \mathtt{K}, \Delta'; \Gamma \vdash J$ *and* $\Delta \vdash \mathtt{C} :: \mathtt{K}$ *then* $\Delta, [\mathtt{C}/\mathtt{u}]\Delta'; [\mathtt{C}/\mathtt{u}]\Gamma \vdash [\mathtt{C}/\mathtt{u}]J$.
3. *If* $\Delta; \Gamma, \mathtt{x} : \mathtt{A}, \Gamma' \vdash J$ *and* $\Delta \vdash \mathtt{E} :: \mathtt{A}$ *then* $\Delta; \Gamma, \Gamma' \vdash [\mathtt{E}/\mathtt{x}]J$.

Proof. By induction over the given derivation. Alternatively, this statement of substitution is true in LF [25], so this follows from THEOREM B.1. □

Using this lemma, we can show that the context in its result, $\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta'$, is well-formed.

LEMMA B.5: REFLEXIVITY OF DEFINITIONAL EQUALITY.

1. *If* $\Delta \vdash \mathtt{K}\,\mathtt{kind}$ *then* $\Delta \vdash \mathtt{K} \equiv \mathtt{K}\,\mathtt{kind}$.

2. *If* $\Delta \vdash \mathtt{C} :: \mathtt{K}$ *then* $\Delta \vdash \mathtt{C} \equiv \mathtt{C} :: \mathtt{K}$.

Proof. In Twelf. □

Two inversion lemmas are necessary for functionality; fortunately, they can be proven first:

LEMMA B.6: INVERIONS, PART 1.

1. *If* $\Delta \vdash \mathtt{s}\ \mathtt{I} :: \mathtt{K}$ *then* $\Delta \vdash \mathtt{I} :: \mathtt{K}'$ *and* $\Delta \vdash \mathtt{K}' \equiv \mathtt{K}\,\mathtt{kind}$.

2. *If* $\Delta \vdash \mathtt{s}\ \mathtt{I} :: \mathtt{NAT}$ *then* $\Delta \vdash \mathtt{I} :: \mathtt{NAT}$.

Proof. In Twelf. □

LEMMA B.7: FUNCTIONALITY OF SUBSTITUTION INTO IDENTICALS. *Assume* $\Delta \vdash \mathtt{C}_2 \equiv \mathtt{C}'_2 :: \mathtt{K}_2$, $\Delta \vdash \mathtt{C}_2 :: \mathtt{K}_2$, $\Delta \vdash \mathtt{C}'_2 :: \mathtt{K}_2$, *and* $\Delta \vdash \mathtt{K}_2\,\mathtt{kind}$.

1. *If* $\Delta, \mathtt{u} :: \mathtt{K}_2, \Delta' \vdash \mathtt{K}\,\mathtt{kind}$ *then* $\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta' \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{K} \equiv [\mathtt{C}'_2/\mathtt{u}]\mathtt{K}\,\mathtt{kind}$.

2. *If* $\Delta, \mathtt{u} :: \mathtt{K}_2, \Delta' \vdash \mathtt{C} :: \mathtt{K}$ *then* $\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta' \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{C} \equiv [\mathtt{C}'_2/\mathtt{u}]\mathtt{C} :: [\mathtt{C}_2/\mathtt{u}]\mathtt{K}$.

Proof. The proof proceeds by a simple mutual induction on the given derivations, but even stating this theorem in Twelf requires some tricks because of the substitution into the context. Thus, it is on paper for now. In the cases we claim are analogous to a previous case, observe that substitution into the constructors and kind in question is always defined analogously to substitution into those in the previous case.

1. The proof is by induction on the derivation of $\Delta, \mathtt{u} :: \mathtt{K}_2, \Delta' \vdash \mathtt{K}\,\mathtt{kind}$.

   - Case for `wf-kd-type`. By the definition of substitution, $[\mathtt{C}/\mathtt{u}]\mathtt{TYPE}$ is $\mathtt{TYPE}$, and `deq-kd-type` gives that $\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta' \vdash \mathtt{TYPE} \equiv \mathtt{TYPE}\,\mathtt{kind}$ (the context in the conclusion of the rule is arbitrary).
   - Case for `wf-kd-nat`. Analogous to the above, except we use `deq-kd-nat`.
   - Case for

$$\dfrac{\overset{\mathcal{D}_1}{\Delta, \mathtt{u} :: \mathtt{K}_2, \Delta' \vdash \mathtt{K}_\mathtt{f}\ \mathtt{kind}} \quad \overset{\mathcal{D}_2}{\Delta, \mathtt{u} :: \mathtt{K}_2, \Delta', \mathtt{v} :: \mathtt{K}_\mathtt{f} \vdash \mathtt{K}_\mathtt{t}\ \mathtt{kind}}}{\Delta, \mathtt{u} :: \mathtt{K}_2, \Delta' \vdash \Pi_\mathtt{k}\, \mathtt{u}{::}\mathtt{K}_\mathtt{f}.\, \mathtt{K}_\mathtt{t}\ \mathtt{kind}}\ \mathtt{wf\text{-}kd\text{-}pi}.$$

     By the IH on $\mathcal{D}_1$, $\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta' \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{K}_\mathtt{f} \equiv [\mathtt{C}'_2/\mathtt{u}]\mathtt{K}_\mathtt{f}\,\mathtt{kind}$. By the IH on $\mathcal{D}_2$, $\Delta, [\mathtt{C}_2/\mathtt{u}](\Delta', \mathtt{v} :: \mathtt{K}_\mathtt{f}) \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{K}_\mathtt{t} \equiv [\mathtt{C}'_2/\mathtt{u}]\mathtt{K}_\mathtt{t}\,\mathtt{kind}$. The the definition of substitution gives that $\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta', \mathtt{v} :: [\mathtt{C}_2/\mathtt{u}]\mathtt{K}_\mathtt{f} \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{K}_\mathtt{t} \equiv [\mathtt{C}'_2/\mathtt{u}]\mathtt{K}_\mathtt{t}\,\mathtt{kind}$, so by `deq-kd-pi` and the definition of substitution we get the result.
   - Case for `wf-kd-eqn`. By the IH, $\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta' \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{I} \equiv [\mathtt{C}'_2/\mathtt{u}]\mathtt{I} :: [\mathtt{C}_2/\mathtt{u}]\mathtt{NAT}$ and $\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta' \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{J} \equiv [\mathtt{C}'_2/\mathtt{u}]\mathtt{J} :: [\mathtt{C}_2/\mathtt{u}]\mathtt{NAT}$. $[\mathtt{C}_2/\mathtt{u}]\mathtt{NAT}$ is $\mathtt{NAT}$, so we can apply `deq-kd-eqn`; then, the definition of substitution gives the result.

2. The proof is by induction on the derivation of $\Delta, \mathtt{u} :: \mathtt{K}_2, \Delta' \vdash \mathtt{C} :: \mathtt{K}$.

- Case for
$$\frac{\Delta, \mathtt{u} :: \mathtt{K_2}, \Delta' \vdash \mathtt{C} :: \mathtt{K} \quad \Delta, \mathtt{u} :: \mathtt{K_2}, \Delta' \vdash \mathtt{K} \equiv \mathtt{K'}\,\mathtt{kind}}{\Delta, \mathtt{u} :: \mathtt{K_2}, \Delta' \vdash \mathtt{C} :: \mathtt{K'}}\ \texttt{ofkd-deq}.$$

  By the IH, $\Delta, [\mathtt{C_2}/\mathtt{u}]\Delta' \vdash [\mathtt{C_2}/\mathtt{u}]\mathtt{C} \equiv [\mathtt{C_2'}/\mathtt{u}]\mathtt{C} :: [\mathtt{C_2}/\mathtt{u}]\mathtt{K}$. By substitution (LEMMA B.4) applied to the second premise derivation, $\Delta, [\mathtt{C_2}/\mathtt{u}]\Delta' \vdash [\mathtt{C_2}/\mathtt{u}]\mathtt{K} \equiv [\mathtt{C_2}/\mathtt{u}]\mathtt{K'}\,\mathtt{kind}$. Then `deq-cn-deq-kd` gives the result.

- Case for `ofkd-var`. We distinguish two subcases, based on whether the variable in question is the one we are substituting for in the theorem statement or not:
  - Case for
  $$\frac{}{\Delta, \mathtt{u} :: \mathtt{K_2}, \Delta' \vdash \mathtt{u} :: \mathtt{K_2}}\ \texttt{ofkd-var}.$$

    By the definition of substitution, $[\mathtt{C_2}/\mathtt{u}]\mathtt{u}$ is $\mathtt{C_2}$ and $[\mathtt{C_2'}/\mathtt{u}]\mathtt{u}$ is $\mathtt{C_2'}$. By assumption, $\Delta \vdash \mathtt{C_2} \equiv \mathtt{C_2'} :: \mathtt{K_2}$, and, since $\mathtt{u}$ is not free in $\mathtt{K_2}$ (by well-formedness of the context), this derivation has the necessary kind. Then, the result is true by weakening (LEMMA B.3).
  - Case for
  $$\frac{(\mathtt{v} :: \mathtt{K} \text{ in } \Delta \text{ or } \Delta')}{\Delta, \mathtt{u} :: \mathtt{K_2}, \Delta' \vdash \mathtt{v} :: \mathtt{K}}\ \texttt{ofkd-var}.$$

    By the definition of substitution, $[\mathtt{X}/\mathtt{u}]\mathtt{v}$ is $\mathtt{v}$. If $\mathtt{v} :: \mathtt{K}$ is in Δ, then by definition of substitution, $\mathtt{v} :: \mathtt{K}$ is in $\Delta, [\mathtt{C_2}/\mathtt{u}]\Delta'$, so we can obtain $\Delta, [\mathtt{C_2}/\mathtt{u}]\Delta' \vdash \mathtt{v} \equiv \mathtt{v} :: \mathtt{K}$ by `deq-cn-var`. Because $\mathtt{u}$ is not free in $\mathtt{K}$, this is what we need. If on the other hand $\mathtt{v} :: \mathtt{K}$ is in Δ', then by the definition of substitution $\mathtt{v} :: [\mathtt{C_2}/\mathtt{u}]\mathtt{K}$ is in $[\mathtt{C_2}/\mathtt{u}]\Delta'$, so by `deq-cn-var`
    $\Delta, [\mathtt{C_2}/\mathtt{u}]\Delta' \vdash \mathtt{v} \equiv \mathtt{v} :: [\mathtt{C_2}/\mathtt{u}]\mathtt{K}$.

- Case for
$$\frac{\Delta, \mathtt{u} :: \mathtt{K_2}, \Delta' \vdash \mathtt{C_f} :: \mathtt{TYPE} \quad \Delta, \mathtt{u} :: \mathtt{K_2}, \Delta' \vdash \mathtt{C_t} :: \mathtt{TYPE}}{\Delta, \mathtt{u} :: \mathtt{K_2}, \Delta' \vdash \mathtt{C_f} \rightarrow \mathtt{C_t} :: \mathtt{TYPE}}\ \texttt{ofkd-arrow}.$$

  Applying the IH to each premise derivation gives that $\Delta, [\mathtt{C_2}/\mathtt{u}]\Delta' \vdash [\mathtt{C_2}/\mathtt{u}]\mathtt{C_f} \equiv [\mathtt{C_2'}/\mathtt{u}]\mathtt{C_f} :: \mathtt{TYPE}$ and $\Delta, [\mathtt{C_2}/\mathtt{u}]\Delta' \vdash [\mathtt{C_2}/\mathtt{u}]\mathtt{C_t} \equiv [\mathtt{C_2'}/\mathtt{u}]\mathtt{C_t} :: \mathtt{TYPE}$ (by the definition of substitution, $[\mathtt{C_2}/\mathtt{u}]\mathtt{TYPE}$ is $\mathtt{TYPE}$). Then `deq-cn-arrow` and the definition of substitution (to pull the substitution outside the $\rightarrow$, and to give the substitution into $\mathtt{TYPE}$) give the result.

- Case for `ofkd-prod`. This case is analogous to `ofkd-arrow`, using `deq-cn-prod`.
- Case for `ofkd-sum`. This case is analogous to `ofkd-arrow`, using `deq-cn-sum`.
- Case for
$$\frac{\Delta, \mathtt{u} :: \mathtt{K_2}, \Delta' \vdash \mathtt{K}\,\mathtt{kind} \quad \Delta, \mathtt{u} :: \mathtt{K_2}, \Delta' \vdash \mathtt{C} :: \Pi_k\, \_ :: \mathtt{K}.\, \mathtt{TYPE}}{\Delta, \mathtt{u} :: \mathtt{K_2}, \Delta' \vdash \forall_{\mathtt{K}}\, \mathtt{C} :: \mathtt{TYPE}}\ \texttt{ofkd-all}.$$

  By the IH, $\Delta, [\mathtt{C_2}/\mathtt{u}]\Delta' \vdash [\mathtt{C_2}/\mathtt{u}]\mathtt{K} \equiv [\mathtt{C_2'}/\mathtt{u}]\mathtt{K}\,\mathtt{kind}$ and
  $\Delta, [\mathtt{C_2}/\mathtt{u}]\Delta' \vdash [\mathtt{C_2}/\mathtt{u}]\mathtt{C} \equiv [\mathtt{C_2'}/\mathtt{u}]\mathtt{C} :: [\mathtt{C_2}/\mathtt{u}]\Pi_k\, \_ :: \mathtt{K}.\, \mathtt{TYPE}$. By the definition of substitution, we can push the substitution inside the Π to get $\Pi_k\, \_ :: [\mathtt{C_2}/\mathtt{u}]\mathtt{K}.\, \mathtt{TYPE}$ (since substitution into $\mathtt{TYPE}$ is a no-op). Then, we can apply `deq-cn-all` and use the definition of substitution to get the result.

- Case for `ofkd-exists`. This case is analogous to `ofkd-all`, using `deq-cn-exists`.
- Case for `ofkd-unit`. By `deq-cn-unit`, $\Delta, [\mathtt{C_2}/\mathtt{u}]\Delta' \vdash \mathtt{unit} \equiv \mathtt{unit} :: \mathtt{TYPE}$. Then, by the definition of substitution, $[\mathtt{X}/\mathtt{u}]\mathtt{unit}$ is $\mathtt{unit}$ and $[\mathtt{X}/\mathtt{u}]\mathtt{TYPE}$ is $\mathtt{TYPE}$, so we have the result.

- Case for `ofkd-void`. This case is analogous to `ofkd-unit`, using `deq-cn-void`.
- Case for
$$\frac{\Delta, \mathtt{u} :: \mathtt{K}_2, \Delta' \vdash \mathtt{I} :: \mathtt{NAT}}{\Delta, \mathtt{u} :: \mathtt{K}_2, \Delta' \vdash \mathtt{nat}\ \mathtt{I} :: \mathtt{TYPE}}\ \texttt{ofkd-nat}.$$
By the IH, $\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta' \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{I} \equiv [\mathtt{C}_2'/\mathtt{u}]\mathtt{I} :: \mathtt{NAT}$ (using the definition of substitution into $\mathtt{NAT}$). Then `deq-cn-nat` and the definition of substitution give the result.
- Case for `ofkd-list`. This case is analogous to `ofkd-nat`, using `deq-cn-list`.
- Case for
$$\frac{\Delta, \mathtt{u} :: \mathtt{K}_2, \Delta' \vdash \mathtt{K}_\mathtt{f}\ \mathtt{kind} \quad \Delta, \mathtt{u} :: \mathtt{K}_2, \Delta', \mathtt{v} :: \mathtt{K}_\mathtt{f} \vdash \mathtt{C} :: \mathtt{K}_\mathtt{t}}{\Delta, \mathtt{u} :: \mathtt{K}_2, \Delta' \vdash \lambda_\mathtt{c}\, \mathtt{v}{::}\mathtt{K}_\mathtt{f}.\, \mathtt{C} :: \Pi_\mathtt{k}\, \mathtt{v}{::}\mathtt{K}_\mathtt{f}.\, \mathtt{K}_\mathtt{t}}\ \texttt{ofkd-fn}.$$
By the IH applied to the premise derivations (and using the definition of substitution),
$\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta' \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{K}_\mathtt{f} \equiv [\mathtt{C}_2'/\mathtt{u}]\mathtt{K}_\mathtt{f}\ \mathtt{kind}$ and
$\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta', \mathtt{v} :: [\mathtt{C}_2/\mathtt{u}]\mathtt{K}_\mathtt{f} \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{C} \equiv [\mathtt{C}_2'/\mathtt{u}]\mathtt{C} :: [\mathtt{C}_2/\mathtt{u}]\mathtt{K}_\mathtt{t}$. Then, by `deq-cn-fn`,
$\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta' \vdash \lambda_\mathtt{c}\, \mathtt{v}{::}[\mathtt{C}_2/\mathtt{u}]\mathtt{K}_\mathtt{f}.\, [\mathtt{C}_2/\mathtt{u}]\mathtt{C} \equiv \lambda_\mathtt{c}\, \mathtt{v}{::}[\mathtt{C}_2'/\mathtt{u}]\mathtt{K}_\mathtt{f}.\, [\mathtt{C}_2'/\mathtt{u}]\mathtt{C} :: (\Pi_\mathtt{k}\, \mathtt{v}{::}[\mathtt{C}_2/\mathtt{u}]\mathtt{K}_\mathtt{f}.\, [\mathtt{C}_2/\mathtt{u}]\mathtt{K}_\mathtt{t})$, so the definition of substitution gives the result (the bound variable is chosen so it does not interfere).
- Case for
$$\frac{\Delta, \mathtt{u} :: \mathtt{K}_2, \Delta' \vdash \mathtt{C}_\mathtt{f} :: \Pi_\mathtt{k}\, \mathtt{v}{::}\mathtt{K}_\mathtt{a}.\, \mathtt{K} \quad \Delta, \mathtt{u} :: \mathtt{K}_2, \Delta' \vdash \mathtt{C}_\mathtt{a} :: \mathtt{K}_\mathtt{a}}{\Delta, \mathtt{u} :: \mathtt{K}_2, \Delta' \vdash \mathtt{C}_\mathtt{f}\ \mathtt{C}_\mathtt{a} :: [\mathtt{C}_\mathtt{a}/\mathtt{v}]\mathtt{K}}\ \texttt{ofkd-app}.$$
Apply the IH to each premise, using the definition of substitution to push the substitution inside the $\Pi_\mathtt{k}$; then use `deq-cn-app`. This gives
$\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta' \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{C}_\mathtt{f}\ [\mathtt{C}_2/\mathtt{u}]\mathtt{C}_\mathtt{a} \equiv [\mathtt{C}_2'/\mathtt{u}]\mathtt{C}_\mathtt{f}\ [\mathtt{C}_2'/\mathtt{u}]\mathtt{C}_\mathtt{a} :: [[\mathtt{C}_2/\mathtt{u}]\mathtt{C}_\mathtt{a}/\mathtt{v}][\mathtt{C}_2/\mathtt{u}]\mathtt{K}$. This result kind is equal to $[\mathtt{C}_2/\mathtt{u}][\mathtt{C}_\mathtt{a}/\mathtt{v}]\mathtt{K}$ by LEMMA B.2 (the bound variable $\mathtt{v}$ can be chosen fresh, so it will not be free in $\mathtt{C}_2$) and the definition of substitution lets us pull the substitution outside the application on each side. This gives the result.
- Case for `ofkd-z`. This case is analogous to `ofkd-unit`, using `deq-cn-z`.
- Case for `ofkd-s`. This case is analogous to `ofkd-nat`, using `deq-cn-s`.
- Case for `ofkd-natrec`. By the IH applied to the premise derivations (and using the definition of substitution),
$$\begin{array}{l}
\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta', \mathtt{i} :: \mathtt{NAT} \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{K} \equiv [\mathtt{C}_2'/\mathtt{u}]\mathtt{K}\ \mathtt{kind} \\
\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta' \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{I} \equiv [\mathtt{C}_2'/\mathtt{u}]\mathtt{I} :: \mathtt{NAT} \\
\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta' \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{C}_\mathtt{z} \equiv [\mathtt{C}_2'/\mathtt{u}]\mathtt{C}_\mathtt{z} :: [\mathtt{C}_2/\mathtt{u}][\mathtt{z}/\mathtt{i}]\mathtt{K} \\
\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta', \mathtt{i}' :: \mathtt{NAT}, \mathtt{r} :: [\mathtt{C}_2/\mathtt{u}][\mathtt{i}'/\mathtt{i}]\mathtt{K} \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{C}_\mathtt{s} \equiv [\mathtt{C}_2'/\mathtt{u}]\mathtt{C}_\mathtt{s} :: [\mathtt{C}_2/\mathtt{u}][\mathtt{s}\ \mathtt{i}'/\mathtt{i}]\mathtt{K}
\end{array}.$$
The bound variable $\mathtt{i}$ can be chosen fresh; then it is not identical to $\mathtt{u}$ and not free in $\mathtt{C}_2$ (since $\mathtt{C}_2$ is well-typed without it in the context). Then by LEMMA B.2 and the definition of substitution (into $\mathtt{z}$, $\mathtt{s}$, and $\mathtt{i}$, where by above we know that $\mathtt{i}$ is distinct from $\mathtt{u}$), we can commute the substitutions into $\mathtt{K}$ in the last two lines; then we can apply `deq-cn-natrec` and use the definition of substitution to get the result.
- Case for `okfd-eqn-zz`. By `deq-cn-eqn-zz`, $\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta' \vdash \mathtt{eqn\_zz} \equiv \mathtt{eqn\_zz} :: \mathtt{EQ}_\mathtt{N}(\mathtt{z}, \mathtt{z})$, since the context in the conclusion of the rule is arbitrary. By the definition of substitution, $[\mathtt{C}_2/\mathtt{u}]\mathtt{eqn\_zz}$ is $\mathtt{eqn\_zz}$ and $[\mathtt{C}_2/\mathtt{u}]\mathtt{EQ}_\mathtt{N}(\mathtt{z}, \mathtt{z})$ is $\mathtt{EQ}_\mathtt{N}(\mathtt{z}, \mathtt{z})$, so we have the result.

- Case for `ofkd-eqn-ss`. By the IH,

$$\begin{array}{l}\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta' \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{I} \equiv [\mathtt{C}_2'/\mathtt{u}]\mathtt{I} :: [\mathtt{C}_2/\mathtt{u}]\mathtt{NAT} \\ \Delta, [\mathtt{C}_2/\mathtt{u}]\Delta' \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{J} \equiv [\mathtt{C}_2'/\mathtt{u}]\mathtt{J} :: [\mathtt{C}_2/\mathtt{u}]\mathtt{NAT} \\ \Delta, [\mathtt{C}_2/\mathtt{u}]\Delta' \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{P} \equiv [\mathtt{C}_2'/\mathtt{u}]\mathtt{P} :: [\mathtt{C}_2/\mathtt{u}]\mathtt{EQ_N}(\mathtt{I},\mathtt{J})\end{array}$$

  Then, by the definition of substitution $[\mathtt{C}_2/\mathtt{u}]\mathtt{NAT}$ is $\mathtt{NAT}$ and $[\mathtt{C}_2/\mathtt{u}]\mathtt{EQ_N}(\mathtt{I},\mathtt{J})$ is $\mathtt{EQ_N}([\mathtt{C}_2/\mathtt{u}]\mathtt{I}, [\mathtt{C}_2/\mathtt{u}]\mathtt{J})$, so we can apply `deq-cn-eqn-ss` and then use the definition of substitution to pull the substitution outside each eqn_ss and the result kind.

- Case for `ofkd-eqn-rec`. By the IH,

$$\begin{array}{l}\Delta, [\mathtt{C}_2/\mathtt{u}](\Delta', \mathtt{i} :: \mathtt{N}, \mathtt{j} :: \mathtt{N}, \mathtt{p} :: \mathtt{EQ_N}(\mathtt{i},\mathtt{j})) \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{K} \equiv [\mathtt{C}_2'/\mathtt{u}]\mathtt{K}'\,\mathtt{kind} \\ \Delta, [\mathtt{C}_2/\mathtt{u}]\Delta' \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{C} \equiv [\mathtt{C}_2'/\mathtt{u}]\mathtt{C}' :: [\mathtt{C}_2/\mathtt{u}]\mathtt{EQ_N}(\mathtt{I},\mathtt{J}) \\ \Delta, [\mathtt{C}_2/\mathtt{u}]\Delta' \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{C_{zz}} \equiv [\mathtt{C}_2'/\mathtt{u}]\mathtt{C}_{\mathtt{zz}}' :: [\mathtt{C}_2/\mathtt{u}][\mathtt{eqn\_zz}/\mathtt{p}][\mathtt{z}/\mathtt{j}][\mathtt{z}/\mathtt{i}]\mathtt{K} \\ \Delta, [\mathtt{C}_2/\mathtt{u}](\Delta', \mathtt{i} :: \mathtt{N}, \mathtt{j} :: \mathtt{N}, \mathtt{p} :: \mathtt{EQ_N}(\mathtt{i},\mathtt{j}), \mathtt{r} :: \mathtt{K}) \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{C_{ss}} \equiv [\mathtt{C}_2'/\mathtt{u}]\mathtt{C}_{\mathtt{ss}}' :: [\mathtt{C}_2/\mathtt{u}][\mathtt{eqn\_ss}(\mathtt{i},\mathtt{j},\mathtt{p})/\mathtt{p}][\mathtt{s}\ \mathtt{j}/\mathtt{j}][\mathtt{s}\ \mathtt{i}/\mathtt{i}]\mathtt{K}.\end{array}$$

  Recall that all context variables are assumed to be distinct. Then, observe that

  - in the first line, $[\mathtt{C}_2/\mathtt{u}](\Delta', \mathtt{i} :: \mathtt{N}, \mathtt{j} :: \mathtt{N}, \mathtt{p} :: \mathtt{EQ_N}(\mathtt{i},\mathtt{j}))$ is $[\mathtt{C}_2/\mathtt{u}]\Delta'$, $\mathtt{i} :: \mathtt{N}$,$\mathtt{j} :: \mathtt{N}$,$\mathtt{p} :: \mathtt{EQ_N}(\mathtt{i},\mathtt{j})$ by the definitions of substitution into contexts, $\mathtt{N}$, $\mathtt{EQ_N}(\mathtt{C}_1,\mathtt{C}_2)$, and variables.
  - In the second line, $[\mathtt{C}_2/\mathtt{u}]\mathtt{EQ_N}(\mathtt{I},\mathtt{J})$ is $\mathtt{EQ_N}([\mathtt{C}_2/\mathtt{u}]\mathtt{I}, [\mathtt{C}_2/\mathtt{u}]\mathtt{J})$.
  - In the third line, $[\mathtt{C}_2/\mathtt{u}][\mathtt{eqn\_zz}/\mathtt{p}][\mathtt{z}/\mathtt{j}][\mathtt{z}/\mathtt{i}]\mathtt{K}$ is $[\mathtt{eqn\_zz}/\mathtt{p}][\mathtt{z}/\mathtt{j}][\mathtt{z}/\mathtt{i}][\mathtt{C}_2/\mathtt{u}]\mathtt{K}$ by LEMMA B.2 and the definition of substitution for z and eqn_zz (we can choose fresh bound variables to satisfy the premises of the lemma).
  - Similarly, in the fourth line, $[\mathtt{C}_2/\mathtt{u}][\mathtt{eqn\_ss}(\mathtt{i},\mathtt{j},\mathtt{p})/\mathtt{p}][\mathtt{s}\ \mathtt{j}/\mathtt{j}][\mathtt{s}\ \mathtt{i}/\mathtt{i}]\mathtt{K}$ is $[\mathtt{eqn\_ss}(\mathtt{i},\mathtt{j},\mathtt{p})/\mathtt{p}][\mathtt{s}\ \mathtt{j}/\mathtt{j}][\mathtt{s}\ \mathtt{i}/\mathtt{i}][\mathtt{C}_2/\mathtt{u}]\mathtt{K}$ Also, the substitution into the context, $[\mathtt{C}_2/\mathtt{u}](\Delta', \mathtt{i} :: \mathtt{N}, \mathtt{j} :: \mathtt{N}, \mathtt{p} :: \mathtt{EQ_N}(\mathtt{i},\mathtt{j}), \mathtt{r} :: \mathtt{K})$, is Δ',$\mathtt{i} :: \mathtt{N}$,$\mathtt{j} :: \mathtt{N}$,$\mathtt{p} :: \mathtt{EQ_N}(\mathtt{i},\mathtt{j})$,$[\mathtt{C}_2/\mathtt{u}]\mathtt{r} :: \mathtt{K}$.

  Applying these syntactic equalities of meta-operations to the above derivations puts them in a form where we can apply `deq-cn-eqn-rec`, and then we can use the definition of substitution to pull the substitutions outside each side and the result kind.

□

LEMMA B.8: FUNCTIONALITY OF SUBSTITUTION INTO DEFINITIONAL EQUALS. *Assume* $\Delta \vdash \mathtt{C}_2 \equiv \mathtt{C}_2' :: \mathtt{K}_2$, $\Delta \vdash \mathtt{C}_2 :: \mathtt{K}_2$, $\Delta \vdash \mathtt{C}_2' :: \mathtt{K}_2$, *and* $\Delta \vdash \mathtt{K}_2\ \mathtt{kind}$.

1. *If* $\Delta, \mathtt{u} :: \mathtt{K}_2, \Delta' \vdash \mathtt{K} \equiv \mathtt{K}'\,\mathtt{kind}$, $\Delta, \mathtt{u} :: \mathtt{K}_2, \Delta' \vdash \mathtt{K}\ \mathtt{kind}$, *and* $\Delta, \mathtt{u} :: \mathtt{K}_2, \Delta' \vdash \mathtt{K}'\ \mathtt{kind}$ *then* $\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta' \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{K} \equiv [\mathtt{C}_2'/\mathtt{u}]\mathtt{K}'\,\mathtt{kind}$.

2. *If* $\Delta, \mathtt{u} :: \mathtt{K}_2, \Delta' \vdash \mathtt{C} \equiv \mathtt{C}' :: \mathtt{K}$ *and* $\Delta, \mathtt{u} :: \mathtt{K}_2, \Delta' \vdash \mathtt{K}\ \mathtt{kind}$ *then* $\Delta, [\mathtt{C}_2/\mathtt{u}]\Delta' \vdash [\mathtt{C}_2/\mathtt{u}]\mathtt{C} \equiv [\mathtt{C}_2'/\mathtt{u}]\mathtt{C}' :: [\mathtt{C}_2/\mathtt{u}]\mathtt{K}$.

Proof. In Twelf. These are immediate consequences of LEMMA B.7 and LEMMA B.5. The extra well-formedness premises are necessary because we have not yet shown regularity; once we do, they will be redundant. □

LEMMA B.9: REGULARITY.

1. *If* $\Delta \vdash \mathtt{K} \equiv \mathtt{K}'\,\mathtt{kind}$ *then* $\Delta \vdash \mathtt{K}\ \mathtt{kind}$ *and* $\Delta \vdash \mathtt{K}'\ \mathtt{kind}$.

2. *If* $\Delta \vdash \mathtt{C} :: \mathtt{K}$ *then* $\Delta \vdash \mathtt{K}\ \mathtt{kind}$

3. *If* $\Delta \vdash \mathtt{C} \equiv \mathtt{C}' :: \mathtt{K}$ *then* $\Delta \vdash \mathtt{C} :: \mathtt{K}$, $\Delta \vdash \mathtt{C}' :: \mathtt{K}$, *and* $\Delta \vdash \mathtt{K}\ \mathtt{kind}$.

4. *If* $\Delta\,;\,\Gamma \vdash \mathtt{E} : \mathtt{A}$ *then* $\Delta \vdash \mathtt{A} :: \mathtt{TYPE}$.

Proof. In Twelf. □

LEMMA B.10: INVERSION.

1. *Inversion of kind equality:*

   - *If* $\Delta \vdash \Pi_k\,\mathtt{u}{::}\mathtt{K_2}.\,\mathtt{K} \equiv \mathtt{L}\,\mathtt{kind}$ *then* $\mathtt{L}$ *is* $\Pi_k\,\mathtt{u}{::}\mathtt{K'_2}.\,\mathtt{K'}$ *where* $\Delta \vdash \mathtt{K_2} \equiv \mathtt{K'_2}\,\mathtt{kind}$ *and* $\Delta, \mathtt{u} :: \mathtt{K_2} \vdash \mathtt{K} \equiv \mathtt{K'}\,\mathtt{kind}$.
   - *If* $\Delta \vdash \mathtt{L} \equiv \Pi_k\,\mathtt{u}{::}\mathtt{K_2}.\,\mathtt{K}\,\mathtt{kind}$ *then* $\mathtt{L}$ *is* $\Pi_k\,\mathtt{u}{::}\mathtt{K'_2}.\,\mathtt{K'}$ *where* $\Delta \vdash \mathtt{K_2} \equiv \mathtt{K'_2}\,\mathtt{kind}$ *and* $\Delta, \mathtt{u} :: \mathtt{K_2} \vdash \mathtt{K} \equiv \mathtt{K'}\,\mathtt{kind}$.
   - *If* $\Delta \vdash \Pi_k\,\mathtt{u}{::}\mathtt{K_2}.\,\mathtt{K} \equiv \Pi_k\,\mathtt{u}{::}\mathtt{K'_2}.\,\mathtt{K'}\,\mathtt{kind}$ *then* $\Delta \vdash \mathtt{K_2} \equiv \mathtt{K'_2}\,\mathtt{kind}$ *and* $\Delta, \mathtt{u} :: \mathtt{K_2} \vdash \mathtt{K} \equiv \mathtt{K'}\,\mathtt{kind}$.

   - *If* $\Delta \vdash \mathtt{EQ_N}(\mathtt{I},\mathtt{J}) \equiv \mathtt{L}\,\mathtt{kind}$ *then* $\mathtt{L}$ *is* $\mathtt{EQ_N}(\mathtt{I'},\mathtt{J'})$ *where* $\Delta \vdash \mathtt{I} \equiv \mathtt{I'} :: \mathtt{N}$ *and* $\Delta \vdash \mathtt{J} \equiv \mathtt{J'} :: \mathtt{N}$.
   - *If* $\Delta \vdash \mathtt{L} \equiv \mathtt{EQ_N}(\mathtt{I},\mathtt{J})\,\mathtt{kind}$ *then* $\mathtt{L}$ *is* $\mathtt{EQ_N}(\mathtt{I'},\mathtt{J'})$ *where* $\Delta \vdash \mathtt{I} \equiv \mathtt{I'} :: \mathtt{N}$ *and* $\Delta \vdash \mathtt{J} \equiv \mathtt{J'} :: \mathtt{N}$.
   - *If* $\Delta \vdash \mathtt{EQ_N}(\mathtt{I},\mathtt{J}) \equiv \mathtt{EQ_N}(\mathtt{I'},\mathtt{J'})\,\mathtt{kind}$ *then* $\Delta \vdash \mathtt{I} \equiv \mathtt{I'} :: \mathtt{N}$ *and* $\Delta \vdash \mathtt{J} \equiv \mathtt{J'} :: \mathtt{N}$.

2. *Inversion of kinding:*

   - *If* $\Delta \vdash \mathtt{C_1} \rightarrow \mathtt{C_2} :: \mathtt{K}$ *then* $\Delta \vdash \mathtt{K} \equiv \mathtt{TYPE}\,\mathtt{kind}$ *and* $\Delta \vdash \mathtt{C_1} :: \mathtt{TYPE}$ *and* $\Delta \vdash \mathtt{C_2} :: \mathtt{TYPE}$. *The analogous statement holds for* $\mathtt{C_1} \times \mathtt{C_2}$ *and* $\mathtt{C_1} + \mathtt{C_2}$.
   - *If* $\Delta \vdash \forall_{\mathtt{K_2}}\,\mathtt{C} :: \mathtt{K}$ *then* $\Delta \vdash \mathtt{K} \equiv \mathtt{TYPE}\,\mathtt{kind}$ *and* $\Delta \vdash \mathtt{C} :: \Pi_k\,\mathtt{u}{::}\mathtt{K_2}.\,\mathtt{TYPE}$. *The analogous statement holds for* $\exists_{\mathtt{K_2}}\,\mathtt{C}$.
   - *If* $\Delta \vdash \mathtt{nat}\,\mathtt{I} :: \mathtt{K}$ *then* $\Delta \vdash \mathtt{K} \equiv \mathtt{TYPE}\,\mathtt{kind}$ *and* $\Delta \vdash \mathtt{I} :: \mathtt{NAT}$. *The analogous statement holds for* $\mathtt{list}\,\mathtt{I}$.
   - *If* $\Delta \vdash \lambda_c\,\mathtt{u}{::}\mathtt{K_2}.\,\mathtt{C} :: \mathtt{K_r}$ *then* $\Delta, \mathtt{u} :: \mathtt{K_2} \vdash \mathtt{C} :: \mathtt{K}$ *and* $\Delta \vdash \mathtt{K_r} \equiv \Pi_k\,\mathtt{u}{::}\mathtt{K_2}.\,\mathtt{K}\,\mathtt{kind}$.
   - *If* $\Delta \vdash \mathtt{C}\,\mathtt{C_2} :: \mathtt{K_r}$ *then* $\Delta \vdash \mathtt{C} :: \Pi_k\,\mathtt{u}{::}\mathtt{K_2}.\,\mathtt{K}$ *and* $\Delta \vdash \mathtt{C_2} :: \mathtt{K_2}$ *and* $\Delta \vdash \mathtt{K_r} \equiv [\mathtt{C_2}/\mathtt{u}]\mathtt{K}\,\mathtt{kind}$.
   - *If* $\Delta \vdash \mathtt{z} :: \mathtt{K}$ *then* $\Delta \vdash \mathtt{K} \equiv \mathtt{NAT}\,\mathtt{kind}$.
   - *If* $\Delta \vdash \mathtt{NATrec}[\mathtt{u}.\mathtt{K}](\mathtt{I}, \mathtt{C_z}, \mathtt{i'}.\mathtt{r}.\mathtt{C_s}) :: \mathtt{K_r}$ *then* $\Delta, \mathtt{u} :: \mathtt{NAT} \vdash \mathtt{K}\,\mathtt{kind}$, $\Delta \vdash \mathtt{I} :: \mathtt{NAT}$, $\Delta \vdash \mathtt{C_z} :: [\mathtt{z}/\mathtt{u}]\mathtt{K}$, $\Delta, \mathtt{i'} :: \mathtt{NAT}, \mathtt{r} :: [\mathtt{i'}/\mathtt{u}]\mathtt{K} \vdash \mathtt{C_s} :: [\mathtt{s}\ \mathtt{i'}/\mathtt{u}]\mathtt{K}$, *and* $\Delta \vdash \mathtt{K_r} \equiv [\mathtt{I}/\mathtt{u}]\mathtt{K}\,\mathtt{kind}$.
   - *If* $\Delta \vdash \mathtt{eqn\_zz} :: \mathtt{K}$ *then* $\Delta \vdash \mathtt{K} \equiv \mathtt{EQ_N}(\mathtt{z},\mathtt{z})\,\mathtt{kind}$.
   - *If* $\Delta \vdash \mathtt{eqn\_ss}(\mathtt{I},\mathtt{J},\mathtt{P}) :: \mathtt{K}$ *then* $\Delta \vdash \mathtt{P} :: \mathtt{EQ_N}(\mathtt{I},\mathtt{J})$ *and* $\Delta \vdash \mathtt{K} \equiv \mathtt{EQ_N}(\mathtt{s}\,\mathtt{I}, \mathtt{s}\,\mathtt{J})\,\mathtt{kind}$.
   - *If* $\Delta \vdash \mathtt{EQ_N rec}[\mathtt{i}.\mathtt{j}.\mathtt{p}.\mathtt{K}](\mathtt{C}, \mathtt{C_1}, \mathtt{i}.\mathtt{j}.\mathtt{p}.\mathtt{r}.\mathtt{C_2}) :: \mathtt{K_r}$ *then* $\Delta, \mathtt{i} :: \mathtt{N}, \mathtt{j} :: \mathtt{N}, \mathtt{p} :: \mathtt{EQ_N}(\mathtt{i},\mathtt{j}), \mathtt{r} :: \mathtt{K} \vdash \mathtt{C_2} :: [\mathtt{eqn\_ss}(\mathtt{i},\mathtt{j},\mathtt{p})/\mathtt{p}][\mathtt{s}\ \mathtt{j}/\mathtt{j}][\mathtt{s}\ \mathtt{i}/\mathtt{i}]\mathtt{K}$, $\Delta \vdash \mathtt{C_1} :: [\mathtt{eqn\_zz}/\mathtt{p}][\mathtt{z}/\mathtt{j}][\mathtt{z}/\mathtt{i}]\mathtt{K}$, $\Delta \vdash \mathtt{C} :: \mathtt{EQ_N}(\mathtt{I},\mathtt{J})$, $\Delta, \mathtt{i} :: \mathtt{N}, \mathtt{j} :: \mathtt{N}, \mathtt{p} :: \mathtt{EQ_N}(\mathtt{i},\mathtt{j}) \vdash \mathtt{K}\,\mathtt{kind}$, *and* $\Delta \vdash \mathtt{K_r} \equiv [\mathtt{C}/\mathtt{p}][\mathtt{J}/\mathtt{j}][\mathtt{I}/\mathtt{i}]\mathtt{K}\,\mathtt{kind}$.

3. *Inversion of constructor equality:*

   - *If* $\Delta \vdash \mathtt{s}\ \mathtt{I} \equiv \mathtt{s}\ \mathtt{I'} :: \mathtt{NAT}$ *then* $\Delta \vdash \mathtt{I} \equiv \mathtt{I'} :: \mathtt{NAT}$.

Proof. In Twelf. The lemmas in the first two categories follow from straightforward induction. For the third category, general inversion properties of constructor equality are not easily provable at this point (intuitively, because of the β rules and transitivity). Indeed, these properties are one of the principle motivations for the algorithmic formulation of definitional equality that we will soon develop. However, because we need this last lemma in developing algorithmic equality, we prove it now; fortunately, it is derivable using `NATrec` to take the predecessor of each side. □

B.3 Deciding Constructor Equality

B.3.1 Algorithmic Equality

Erased kinds Algorithmic equality is directed by approximate kinds, where all dependencies are erased. The erased kinds are

$$\widehat{\mathtt{K}} ::= \widehat{\mathtt{TYPE}} \mid \widehat{\mathtt{K}}_1 \widehat{\rightarrow}_k \widehat{\mathtt{K}}_2 \mid \widehat{\mathtt{NAT}} \mid \widehat{\mathtt{EQ_N}}.$$

A new form of context maps constructor variables to erased kinds:

$$\Psi ::= \cdot \mid \Psi, \mathtt{u} :: \widehat{\mathtt{K}}.$$

All syntactically correct erased kinds are well-formed, so the only condition on a well-formed Ψ is that no variable occurs more than once.

The erasure function $(\cdot)^-$ from kinds to erased kinds is defined as follows:

$$\begin{array}{rcl} (\mathtt{TYPE})^- & = & \widehat{\mathtt{TYPE}} \\ (\Pi_k\, \mathtt{u}{::}\mathtt{K}_2.\, \mathtt{K})^- & = & (\mathtt{K}_2)^- \widehat{\rightarrow}_k (\mathtt{K})^- \\ (\mathtt{NAT})^- & = & \widehat{\mathtt{NAT}} \\ (\mathtt{EQ_N}(\mathtt{C}_1, \mathtt{C}_2))^- & = & \widehat{\mathtt{EQ_N}}. \end{array}$$

We extend this function pointwise to contexts, denoted by $(\Delta)^-$. Because a well-formed Δ binds each variable once, $(\Delta)^-$ is well-formed when Δ is.

LEMMA B.11: ERASURE PROPERTIES.

1. *For all kinds* $\mathtt{K}$, $(\mathtt{K})^-$ *exists.*
2. *If* $(\mathtt{K})^- = \widehat{\mathtt{K}}$ *and* $(\mathtt{K})^- = \widehat{\mathtt{K}}'$ *then* $\widehat{\mathtt{K}}$ *is* $\widehat{\mathtt{K}}'$.
3. *If* $\Delta \vdash \mathtt{K} \equiv \mathtt{K}'\,\mathtt{kind}$ *then* $(\mathtt{K})^-$ *is* $(\mathtt{K}')^-$.
4. *If* $\mathtt{u}$ *is potentially free in* $\mathtt{K}$, *then* $([\mathtt{C}/\mathtt{u}]\mathtt{K})^-$ *is* $(\mathtt{K})^-$.

Proof. In Twelf. □

The first two parts of this lemma justify using function notation for $(\cdot)^-$.

Definition of Algorithmic Equality

$\boxed{\Psi \vdash \mathtt{K} \iff \mathtt{K}'\,\mathtt{kind}}$

$$\frac{}{\Psi \vdash \mathtt{TYPE} \iff \mathtt{TYPE}\,\mathtt{kind}}\ \texttt{norm-eq-kd-type}$$

$$\frac{\Psi \vdash \mathtt{K}_1 \iff \mathtt{K}_1'\,\mathtt{kind} \quad \Psi, \mathtt{u} :: (\mathtt{K}_1)^- \vdash \mathtt{K}_2 \iff \mathtt{K}_2'\,\mathtt{kind}}{\Psi \vdash \Pi_k\, \mathtt{u}{::}\mathtt{K}_1.\, \mathtt{K}_2 \iff \Pi_k\, \mathtt{u}{::}\mathtt{K}_1'.\, \mathtt{K}_2'\,\mathtt{kind}}\ \texttt{norm-eq-kd-pi}$$

$$\frac{}{\Psi \vdash \mathtt{NAT} \iff \mathtt{NAT}\,\mathtt{kind}}\ \texttt{norm-eq-kd-nat}$$

$$\frac{\Psi \vdash \mathtt{I} \iff \mathtt{I}' :: \widehat{\mathtt{NAT}} \quad \Psi \vdash \mathtt{J} \iff \mathtt{J}' :: \widehat{\mathtt{NAT}}}{\Psi \vdash \mathtt{EQ_N}(\mathtt{I}, \mathtt{J}) \iff \mathtt{EQ_N}(\mathtt{I}', \mathtt{J}')\,\mathtt{kind}}\ \texttt{norm-eq-kd-eqn}$$

$\boxed{\widehat{\mathtt{K}}\ \mathtt{base}}$

$$\frac{}{\widehat{\mathtt{TYPE}}\ \mathtt{base}}\ \texttt{base-kd-type} \qquad \frac{}{\widehat{\mathtt{NAT}}\ \mathtt{base}}\ \texttt{base-kd-nat} \qquad \frac{}{\widehat{\mathtt{EQ_N}}\ \mathtt{base}}\ \texttt{base-kd-eqn}$$

$\boxed{\Psi \vdash \mathtt{C} \Longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{K}}}$

$$\frac{\widehat{\mathtt{K}}\ \mathtt{base} \quad \mathtt{C}_1 \xrightarrow{\text{whr}} \mathtt{C}_1' \quad \Psi \vdash \mathtt{C}_1' \Longleftrightarrow \mathtt{C}_2 :: \widehat{\mathtt{K}}}{\Psi \vdash \mathtt{C}_1 \Longleftrightarrow \mathtt{C}_2 :: \widehat{\mathtt{K}}}\ \texttt{norm-eq-cn-whr-left}$$

$$\frac{\widehat{\mathtt{K}}\ \mathtt{base} \quad \mathtt{C}_2 \xrightarrow{\text{whr}} \mathtt{C}_2' \quad \Psi \vdash \mathtt{C}_1 \Longleftrightarrow \mathtt{C}_2' :: \widehat{\mathtt{K}}}{\Psi \vdash \mathtt{C}_1 \Longleftrightarrow \mathtt{C}_2 :: \widehat{\mathtt{K}}}\ \texttt{norm-eq-cn-whr-right}$$

$$\frac{\Psi \vdash \mathtt{C}_1 \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{C}_2}{\Psi \vdash \mathtt{C}_1 \Longleftrightarrow \mathtt{C}_2 :: \widehat{\mathtt{TYPE}}}\ \texttt{norm-eq-cn-type}$$

$$\frac{\Psi, \mathtt{u} :: \widehat{\mathtt{K}}_2 \vdash \mathtt{C}\ \mathtt{u} \Longleftrightarrow \mathtt{C}'\ \mathtt{u} :: \widehat{\mathtt{K}}}{\Psi \vdash \mathtt{C} \Longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{K}}_2 \widehat{\to}_{\mathtt{k}} \widehat{\mathtt{K}}}\ \texttt{norm-eq-cn-arrow}$$

$$\frac{\Psi \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{NAT}} \mathtt{C}'}{\Psi \vdash \mathtt{C} \Longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{NAT}}}\ \texttt{norm-eq-cn-nat}$$

$$\frac{\Psi \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{EQ_N}} \mathtt{C}'}{\Psi \vdash \mathtt{C} \Longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{EQ_N}}}\ \texttt{norm-eq-cn-eqn}$$

$\boxed{\Psi \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{C}'}$

$$\frac{\Psi \vdash \mathtt{C} \longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{TYPE}}}{\Psi \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{C}'}\ \texttt{norm-eq-cn/type-neut-eq}$$

$$\frac{\Psi \vdash \mathtt{C}_1 \Longleftrightarrow \mathtt{C}_1' :: \widehat{\mathtt{TYPE}} \quad \Psi \vdash \mathtt{C}_2 \Longleftrightarrow \mathtt{C}_2' :: \widehat{\mathtt{TYPE}}}{\Psi \vdash \mathtt{C}_1 \to \mathtt{C}_2 \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{C}_1' \to \mathtt{C}_2'}\ \texttt{norm-eq-cn/type-arrow}$$

$$\frac{\Psi \vdash \mathtt{C}_1 \Longleftrightarrow \mathtt{C}_1' :: \widehat{\mathtt{TYPE}} \quad \Psi \vdash \mathtt{C}_2 \Longleftrightarrow \mathtt{C}_2' :: \widehat{\mathtt{TYPE}}}{\Psi \vdash \mathtt{C}_1 \times \mathtt{C}_2 \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{C}_1' \times \mathtt{C}_2'}\ \texttt{norm-eq-cn/type-prod}$$

$$\frac{\Psi \vdash \mathtt{C}_1 \Longleftrightarrow \mathtt{C}_1' :: \widehat{\mathtt{TYPE}} \quad \Psi \vdash \mathtt{C}_2 \Longleftrightarrow \mathtt{C}_2' :: \widehat{\mathtt{TYPE}}}{\Psi \vdash \mathtt{C}_1 + \mathtt{C}_2 \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{C}_1' + \mathtt{C}_2'}\ \texttt{norm-eq-cn/type-sum}$$

$$\frac{\Psi \vdash \mathtt{K}_2 \Longleftrightarrow \mathtt{K}_2'\ \mathtt{kind} \quad \Psi \vdash \mathtt{C} \Longleftrightarrow \mathtt{C}' :: (\mathtt{K}_2)^{-} \widehat{\to}_{\mathtt{k}} \widehat{\mathtt{TYPE}}}{\Psi \vdash \forall_{\mathtt{K}_2}\ \mathtt{C} \Longleftrightarrow_{\mathtt{TYPE}} \forall_{\mathtt{K}_2'}\ \mathtt{C}'}\ \texttt{norm-eq-cn/type-all}$$

$$\frac{\Psi \vdash \mathtt{K}_2 \Longleftrightarrow \mathtt{K}_2'\ \mathtt{kind} \quad \Psi \vdash \mathtt{C} \Longleftrightarrow \mathtt{C}' :: (\mathtt{K}_2)^{-} \widehat{\to}_{\mathtt{k}} \widehat{\mathtt{TYPE}}}{\Psi \vdash \exists_{\mathtt{K}_2}\ \mathtt{C} \Longleftrightarrow_{\mathtt{TYPE}} \exists_{\mathtt{K}_2'}\ \mathtt{C}'}\ \texttt{norm-eq-cn/type-exists}$$

$$\frac{}{\Psi \vdash \mathtt{unit} \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{unit}}\ \texttt{norm-eq-cn/type-unit}$$

$$\frac{}{\Psi \vdash \mathtt{void} \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{void}}\ \texttt{norm-eq-cn/type-void}$$

$$\frac{\Psi \vdash \mathtt{C} \Longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{NAT}}}{\Psi \vdash \mathtt{nat}\, \mathtt{C} \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{nat}\, \mathtt{C}'}\ \texttt{norm-eq-cn/type-nat}$$

$$\frac{\Psi \vdash \mathtt{C} \Longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{NAT}}}{\Psi \vdash \mathtt{list}\, \mathtt{C} \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{list}\, \mathtt{C}'}\ \texttt{norm-eq-cn/type-list}$$

$\boxed{\Psi \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{NAT}} \mathtt{C}'}$

$$\frac{\Psi \vdash \mathtt{C} \longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{NAT}}}{\Psi \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{NAT}} \mathtt{C}'}\ \texttt{norm-eq-cn/nat-neut-eq}$$

$$\frac{}{\Psi \vdash \mathtt{z} \Longleftrightarrow_{\mathtt{NAT}} \mathtt{z}}\ \texttt{norm-eq-cn/nat-z} \qquad \frac{\Psi \vdash \mathtt{C} \Longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{NAT}}}{\Psi \vdash \mathtt{s}\, \mathtt{C} \Longleftrightarrow_{\mathtt{NAT}} \mathtt{s}\, \mathtt{C}'}\ \texttt{norm-eq-cn/nat-s}$$

$\boxed{\Psi \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{EQ_N}} \mathtt{C}'}$

$$\frac{\Psi \vdash \mathtt{C} \longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{EQ_N}}}{\Psi \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{EQ_N}} \mathtt{C}'}\ \texttt{norm-eq-cn/eqn-neut-eq}$$

$$\frac{}{\Psi \vdash \mathtt{eqn\_zz} \Longleftrightarrow_{\mathtt{EQ_N}} \mathtt{eqn\_zz}}\ \texttt{norm-eq-cn/eqn-zz}$$

$$\frac{\Psi \vdash \mathtt{I} \Longleftrightarrow \mathtt{I}' :: \widehat{\mathtt{NAT}} \quad \Psi \vdash \mathtt{J} \Longleftrightarrow \mathtt{J}' :: \widehat{\mathtt{NAT}} \quad \Psi \vdash \mathtt{P} \Longleftrightarrow \mathtt{P}' :: \widehat{\mathtt{EQ_N}}}{\Psi \vdash \mathtt{eqn\_ss}(\mathtt{I}, \mathtt{J}, \mathtt{P}) \Longleftrightarrow_{\mathtt{EQ_N}} \mathtt{eqn\_ss}(\mathtt{I}', \mathtt{J}', \mathtt{P}')}\ \texttt{norm-eq-cn/eqn-ss}$$

$\boxed{\Psi \vdash \mathtt{C} \longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{K}}}$

$$\frac{}{\Psi, \mathtt{u} :: \widehat{\mathtt{K}}, \Psi' \vdash \mathtt{u} \longleftrightarrow \mathtt{u} :: \widehat{\mathtt{K}}}\ \texttt{neut-eq-cn-var}$$

$$\frac{\Psi \vdash \mathtt{C}_1 \longleftrightarrow \mathtt{C}_1' :: \widehat{\mathtt{K}}_2 \widehat{\rightarrow}_k \widehat{\mathtt{K}} \quad \Psi \vdash \mathtt{C}_2 \Longleftrightarrow \mathtt{C}_2' :: \widehat{\mathtt{K}}_2}{\Psi \vdash \mathtt{C}_1\, \mathtt{C}_2 \longleftrightarrow \mathtt{C}_1'\, \mathtt{C}_2' :: \widehat{\mathtt{K}}}\ \texttt{neut-eq-cn-app}$$

$$\frac{\begin{array}{c}\Psi, \mathtt{u} :: \widehat{\mathtt{NAT}} \vdash \mathtt{K} \Longleftrightarrow \mathtt{K}'\, \mathsf{kind} \quad (\mathtt{K})^- \text{ is } \widehat{\mathtt{K}} \quad \Psi \vdash \mathtt{I} \longleftrightarrow \mathtt{I}' :: \widehat{\mathtt{NAT}} \\ \Psi \vdash \mathtt{C_z} \Longleftrightarrow \mathtt{C_z'} :: \widehat{\mathtt{K}} \\ \Psi, \mathtt{i}' :: \widehat{\mathtt{NAT}}, \mathtt{r} :: \widehat{\mathtt{K}} \vdash \mathtt{C_s} \Longleftrightarrow \mathtt{C_s'} :: \widehat{\mathtt{K}}\end{array}}{\Psi \vdash \mathtt{NATrec}[\mathtt{u}.\mathtt{K}](\mathtt{I}, \mathtt{C_z}, \mathtt{i}'.\mathtt{r}.\mathtt{C_s}) \longleftrightarrow \mathtt{NATrec}[\mathtt{u}.\mathtt{K}'](\mathtt{I}', \mathtt{C_z'}, \mathtt{i}'.\mathtt{r}.\mathtt{C_s'}) :: \widehat{\mathtt{K}}}\ \texttt{neut-eq-cn-natrec}$$

$$\frac{\begin{array}{c}\Psi, \mathtt{i} :: \widehat{\mathtt{NAT}}, \mathtt{j} :: \widehat{\mathtt{NAT}}, \mathtt{p} :: \widehat{\mathtt{EQ_N}} \vdash \mathtt{K} \Longleftrightarrow \mathtt{K}'\, \mathsf{kind} \quad (\mathtt{K})^- \text{ is } \widehat{\mathtt{K}} \quad \Psi \vdash \mathtt{P} \longleftrightarrow \mathtt{P}' :: \widehat{\mathtt{EQ_N}} \\ \Psi \vdash \mathtt{C_{zz}} \Longleftrightarrow \mathtt{C_{zz}'} :: \widehat{\mathtt{K}} \\ \Psi, \mathtt{i} :: \widehat{\mathtt{NAT}}, \mathtt{j} :: \widehat{\mathtt{NAT}}, \mathtt{p} :: \widehat{\mathtt{EQ_N}}, \mathtt{r} :: \widehat{\mathtt{K}} \vdash \mathtt{C_{ss}} \Longleftrightarrow \mathtt{C_{ss}'} :: \widehat{\mathtt{K}}\end{array}}{\Psi \vdash \mathtt{EQ_N rec}[\mathtt{i}.\mathtt{j}.\mathtt{p}.\mathtt{K}](\mathtt{P}, \mathtt{C_{zz}}, \mathtt{i}.\mathtt{j}.\mathtt{p}.\mathtt{r}.\mathtt{C_{ss}}) \longleftrightarrow \mathtt{EQ_N rec}[\mathtt{i}.\mathtt{j}.\mathtt{p}.\mathtt{K}'](\mathtt{P}', \mathtt{C_{zz}'}, \mathtt{i}.\mathtt{j}.\mathtt{p}.\mathtt{r}.\mathtt{C_{ss}'}) :: \widehat{\mathtt{K}}}\ \texttt{neut-eq-cn-eqnrec}$$

$\boxed{\mathtt{C} \overset{\text{whr}}{\longrightarrow}{}^* \mathtt{C}'}$

Weak head reduction, $\mathtt{C} \overset{\text{whr}}{\longrightarrow} \mathtt{C}'$, was defined in Section 5.

$$\frac{}{\mathtt{C} \overset{\text{whr}}{\longrightarrow}{}^* \mathtt{C}'}\ \texttt{whrrt-refl} \qquad \frac{\mathtt{C}_1 \overset{\text{whr}}{\longrightarrow} \mathtt{C}_1' \quad \mathtt{C}_1' \overset{\text{whr}}{\longrightarrow}{}^* \mathtt{C}_2}{\mathtt{C}_1 \overset{\text{whr}}{\longrightarrow}{}^* \mathtt{C}_2}\ \texttt{whrrt-whr}$$

$\boxed{\mathtt{C}\ \mathtt{whnorm}\text{ and }\mathtt{C}\ \mathtt{whneut}}$

These judgements can be given by a subsyntax; the meta-variables not being defined here still refer to any constructor:

$$\begin{array}{lcl}
\mathtt{R}\ \mathtt{whneut} & ::= & \mathtt{u} \mid \mathtt{R}\ \mathtt{C}_2 \mid \ \mid \mathtt{NATrec}[\mathtt{u.K}](\mathtt{R}, \mathtt{C_z}, \mathtt{i'.r.C_s}) \mid \mathtt{EQ_N rec}[\mathtt{i.j.p.K}](\mathtt{R}, \mathtt{C_{zz}}, \mathtt{i.j.p.r.C_{ss}}) \\
\mathtt{N}\ \mathtt{whnorm} & ::= & \mathtt{R} \mid \mathtt{C}_1 \rightarrow \mathtt{C}_2 \mid \mathtt{C}_1 \times \mathtt{C}_2 \mid \mathtt{C}_1 + \mathtt{C}_2 \mid \forall_{\mathtt{K}_2}\ \mathtt{C} \mid \exists_{\mathtt{K}_2}\ \mathtt{C} \mid \mathtt{unit} \mid \mathtt{void} \mid \mathtt{nat}\ \mathtt{I} \mid \mathtt{list}\ \mathtt{I} \\
 & & \mid \lambda_c\, \mathtt{u{::}K.\,C} \mid \mathtt{z} \mid \mathtt{s}\ \mathtt{I} \mid \mathtt{eqn\_zz} \mid \mathtt{eqn\_ss}(\mathtt{I}, \mathtt{J}, \mathtt{P})
\end{array}$$

Discussion of Algorithmic Equality We refer to $\Psi \vdash \mathtt{C} \Longleftrightarrow \mathtt{C'} :: \widehat{\mathtt{K}}$ and its auxiliary judgements ($\Psi \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{C'}$, $\Psi \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{NAT}} \mathtt{C'}$, and $\Psi \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{EQ_N}} \mathtt{C'}$) as *normal equality* (or, more precisely, *normalizing equality*) because these judgements determine equality by normalizing constructors. We refer to $\Psi \vdash \mathtt{C} \longleftrightarrow \mathtt{C'} :: \widehat{\mathtt{K}}$ as *neutral equality* because this judgements determines equality of neutral constructors. The rules for these judgements are well-moded. Operationally, the erased kind in $\Psi \vdash \mathtt{C} \longleftrightarrow \mathtt{C'} :: \widehat{\mathtt{K}}$ and the right-hand constructor in $\mathtt{C} \xrightarrow{\text{whr}} \mathtt{C'}$ and $\mathtt{C} \xrightarrow{\text{whr}}{}^{*} \mathtt{C'}$are outputs; all other meta-variables appearing in the judgements are inputs.

Properties of Algorithmic Equality

LEMMA B.12: ADEQUACY OF ALGORITHMIC EQUALITY ENCODING. *These lemmas refer to the LF signature that is available in the companion Twelf code [1].*

1. *There is a bijection between the following.*

Syntactic Category	***Canonical LF Terms of Type***	***in LF contexts***
$\widehat{\mathtt{K}}$	$\mathtt{k\hat{d}}$	$\cdot$

2. *There are bijections between the following.*

Derivations of	***Canon. LF Terms of Type***	***in LF contexts***
$\widehat{\mathtt{K}}$ base, *FV in* $\mathtt{u}_1 \ldots$	$\mathtt{base\_k\hat{d}}\ \ulcorner\widehat{\mathtt{K}}\urcorner$	$\mathtt{u}_1 : \mathtt{cn}$
$(\mathtt{K})^- = \widehat{\mathtt{K}}$, *FV in* $\mathtt{u}_1 \ldots$	$\mathtt{ed/kd}\ \ulcorner\mathtt{K}\urcorner\ \ulcorner\widehat{\mathtt{K}}\urcorner$	$\mathtt{u}_1 : \mathtt{cn}$
$\mathtt{C}$ whnorm, *FV in* $\mathtt{u}_1 \ldots$	$\mathtt{whnorm}\ \ulcorner\mathtt{C}\urcorner$	$\mathtt{u}_1 : \mathtt{cn}$
$\mathtt{C}$ whneut, *FV in* $\mathtt{u}_1 \ldots$	$\mathtt{whneut}\ \ulcorner\mathtt{C}\urcorner$	$\mathtt{u}_1 : \mathtt{cn}$
$\mathtt{u}_1 :: \widehat{\mathtt{K}}_1 \ldots \vdash \mathtt{K} \Longleftrightarrow \mathtt{K'}\ \mathtt{kind}$	$\mathtt{norm\_eq\_kd}\ \ulcorner\mathtt{K}\urcorner\ \ulcorner\mathtt{K'}\urcorner$	$\mathtt{u}_1 : \mathtt{cn}, \mathtt{nequ}_1 : \mathtt{neut\_eq\_cn}\ \mathtt{u}_1\ \mathtt{u}_1\ \ulcorner\widehat{\mathtt{K}}_1\urcorner \ldots$
$\mathtt{u}_1 :: \widehat{\mathtt{K}}_1 \ldots \vdash \mathtt{C} \Longleftrightarrow \mathtt{C'} :: \widehat{\mathtt{K}}$	$\mathtt{norm\_eq\_cn}\ \ulcorner\mathtt{C}\urcorner\ \ulcorner\mathtt{C'}\urcorner\ \ulcorner\widehat{\mathtt{K}}\urcorner$	$\mathtt{u}_1 : \mathtt{cn}, \mathtt{nequ}_1 : \mathtt{neut\_eq\_cn}\ \mathtt{u}_1\ \mathtt{u}_1\ \ulcorner\widehat{\mathtt{K}}_1\urcorner \ldots$
$\mathtt{u}_1 :: \widehat{\mathtt{K}}_1 \ldots \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{C'}$	$\mathtt{norm\_eq\_cn/type}\ \ulcorner\mathtt{C}\urcorner\ \ulcorner\mathtt{C'}\urcorner$	$\mathtt{u}_1 : \mathtt{cn}, \mathtt{nequ}_1 : \mathtt{neut\_eq\_cn}\ \mathtt{u}_1\ \mathtt{u}_1\ \ulcorner\widehat{\mathtt{K}}_1\urcorner \ldots$
$\mathtt{u}_1 :: \widehat{\mathtt{K}}_1 \ldots \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{NAT}} \mathtt{C'}$	$\mathtt{norm\_eq\_cn/nat}\ \ulcorner\mathtt{C}\urcorner\ \ulcorner\mathtt{C'}\urcorner$	$\mathtt{u}_1 : \mathtt{cn}, \mathtt{nequ}_1 : \mathtt{neut\_eq\_cn}\ \mathtt{u}_1\ \mathtt{u}_1\ \ulcorner\widehat{\mathtt{K}}_1\urcorner \ldots$
$\mathtt{u}_1 :: \widehat{\mathtt{K}}_1 \ldots \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{EQ_N}} \mathtt{C'}$	$\mathtt{norm\_eq\_cn/eqn}\ \ulcorner\mathtt{C}\urcorner\ \ulcorner\mathtt{C'}\urcorner$	$\mathtt{u}_1 : \mathtt{cn}, \mathtt{nequ}_1 : \mathtt{neut\_eq\_cn}\ \mathtt{u}_1\ \mathtt{u}_1\ \ulcorner\widehat{\mathtt{K}}_1\urcorner \ldots$
$\mathtt{u}_1 :: \widehat{\mathtt{K}}_1 \ldots \vdash \mathtt{C} \longleftrightarrow \mathtt{C'} :: \widehat{\mathtt{K}}$	$\mathtt{neut\_eq\_cn}\ \ulcorner\mathtt{C}\urcorner\ \ulcorner\mathtt{C'}\urcorner\ \ulcorner\widehat{\mathtt{K}}\urcorner$	$\mathtt{u}_1 : \mathtt{cn}, \mathtt{nequ}_1 : \mathtt{neut\_eq\_cn}\ \mathtt{u}_1\ \mathtt{u}_1\ \ulcorner\widehat{\mathtt{K}}_1\urcorner \ldots$

Proof. Again, the proofs of adequacy follow standard techniques [22]. □

LEMMA B.13: ERASURES OF ALGORITHMIC EQUALS ARE IDENTICAL.
If $\Psi \vdash \mathtt{K} \Longleftrightarrow \mathtt{K'}\ \mathtt{kind}$ *then* $(\mathtt{K})^-$ *is* $(\mathtt{K'})^-$.

Proof. In Twelf. □

LEMMA B.14: WEAKENING OF ALGORITHMIC EQUALITY.
For algorithmic equality judgements J, if $\Psi, \Psi' \vdash J$ *and* $\Psi, \mathtt{u} :: \widehat{\mathtt{K}}, \Psi'$ *is well-formed then* $\Psi, \mathtt{u} :: \widehat{\mathtt{K}}, \Psi' \vdash J$.

Proof. By induction over the given derivation. Alternatively, weakening is true in LF, so this follows from LEMMA B.12. □

LEMMA B.15: DETERMINACY OF WEAK HEAD REDUCTION.
If $\mathtt{C} \xrightarrow{\text{whr}} \mathtt{C}'$ *and* $\mathtt{C} \xrightarrow{\text{whr}} \mathtt{C}''$ *then* $\mathtt{C}'$ *is* $\mathtt{C}''$.

Proof. In Twelf. □

LEMMA B.16: CONSTRUCTORS ARE NEUTRALLY EQUAL AT A UNIQUE KIND.
If $\Psi \vdash \mathtt{C}_1 \longleftrightarrow \mathtt{C}_2 :: \widehat{\mathtt{K}}$ *and* $\Psi \vdash \mathtt{C}_2 \longleftrightarrow \mathtt{C}_3 :: \widehat{\mathtt{K}}'$ *then* $\widehat{\mathtt{K}}$ *is* $\widehat{\mathtt{K}}'$.

Proof. In Twelf. □

LEMMA B.17: SUBJECTS OF AUXILLARY JUDGEMENTS ARE WEAK HEAD NORMAL.

1. *If* $\Psi \vdash \mathtt{C}_1 \longleftrightarrow \mathtt{C}_2 :: \widehat{\mathtt{K}}$ *then* $\mathtt{C}_1$ $\mathtt{whneut}$ *and* $\mathtt{C}_2$ $\mathtt{whneut}$
2. *If* $\Psi \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{C}'$ *then* $\mathtt{C}$ $\mathtt{whnorm}$ *and* $\mathtt{C}'$ $\mathtt{whnorm}$
3. *If* $\Psi \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{NAT}} \mathtt{C}'$ *then* $\mathtt{C}$ $\mathtt{whnorm}$ *and* $\mathtt{C}'$ $\mathtt{whnorm}$
4. *If* $\Psi \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{EQ_N}} \mathtt{C}'$ *then* $\mathtt{C}$ $\mathtt{whnorm}$ *and* $\mathtt{C}'$ $\mathtt{whnorm}$

Proof. In Twelf. □

LEMMA B.18: WEAK HEAD NORMAL CONSTRUCTORS ARE NOT WEAK HEAD REDUCIBLE.

1. $\mathtt{C}$ $\mathtt{whneut}$ *and* $\mathtt{C} \xrightarrow{\text{whr}} \mathtt{C}'$ *imply a contradiction.*
2. $\mathtt{C}$ $\mathtt{whnorm}$ *and* $\mathtt{C} \xrightarrow{\text{whr}} \mathtt{C}'$ *imply a contradiction.*

Proof. In Twelf. □

LEMMA B.19: SYMMETRY OF ALGORITHMIC EQUALITY.

1. *If* $\Psi \vdash \mathtt{K}_1 \Longleftrightarrow \mathtt{K}_2\,\mathtt{kind}$ *then* $\Psi \vdash \mathtt{K}_2 \Longleftrightarrow \mathtt{K}_1\,\mathtt{kind}$.
2. *If* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow \mathtt{C}_2 :: \widehat{\mathtt{K}}$ *then* $\Psi \vdash \mathtt{C}_2 \Longleftrightarrow \mathtt{C}_1 :: \widehat{\mathtt{K}}$.
3. *If* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{C}_2$ *then* $\Psi \vdash \mathtt{C}_2 \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{C}_1$.
4. *If* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow_{\mathtt{NAT}} \mathtt{C}_2$ *then* $\Psi \vdash \mathtt{C}_2 \Longleftrightarrow_{\mathtt{NAT}} \mathtt{C}_1$.
5. *If* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow_{\mathtt{EQ_N}} \mathtt{C}_2$ *then* $\Psi \vdash \mathtt{C}_2 \Longleftrightarrow_{\mathtt{NAT}} \mathtt{C}_1$.
6. *If* $\Psi \vdash \mathtt{C}_1 \longleftrightarrow \mathtt{C}_2 :: \widehat{\mathtt{K}}$ *then* $\Psi \vdash \mathtt{C}_2 \longleftrightarrow \mathtt{C}_1 :: \widehat{\mathtt{K}}$.

Proof. In Twelf. □

LEMMA B.20: TRANSITIVITY OF ALGORITHMIC EQUALITY.

1. *If* $\Psi \vdash \mathtt{K}_1 \Longleftrightarrow \mathtt{K}_2\,\mathtt{kind}$ *and* $\Psi \vdash \mathtt{K}_2 \Longleftrightarrow \mathtt{K}_3\,\mathtt{kind}$ *then* $\Psi \vdash \mathtt{K}_1 \Longleftrightarrow \mathtt{K}_3\,\mathtt{kind}$.
2. *If* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow \mathtt{C}_2 :: \widehat{\mathtt{K}}$ *and* $\Psi \vdash \mathtt{C}_2 \Longleftrightarrow \mathtt{C}_3 :: \widehat{\mathtt{K}}$ *then* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow \mathtt{C}_3 :: \widehat{\mathtt{K}}$.
3. *If* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{C}_2$ *and* $\Psi \vdash \mathtt{C}_2 \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{C}_3$ *then* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{C}_3$.
4. *If* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow_{\mathtt{NAT}} \mathtt{C}_2$ *and* $\Psi \vdash \mathtt{C}_2 \Longleftrightarrow_{\mathtt{NAT}} \mathtt{C}_3$ *then* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow_{\mathtt{NAT}} \mathtt{C}_3$.
5. *If* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow_{\mathtt{EQ_N}} \mathtt{C}_2$ *and* $\Psi \vdash \mathtt{C}_2 \Longleftrightarrow_{\mathtt{EQ_N}} \mathtt{C}_3$ *then* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow_{\mathtt{EQ_N}} \mathtt{C}_3$.
6. *If* $\Psi \vdash \mathtt{C}_1 \longleftrightarrow \mathtt{C}_2 :: \widehat{\mathtt{K}}$ *and* $\Psi \vdash \mathtt{C}_2 \longleftrightarrow \mathtt{C}_3 :: \widehat{\mathtt{K}}$ *then* $\Psi \vdash \mathtt{C}_1 \longleftrightarrow \mathtt{C}_3 :: \widehat{\mathtt{K}}$.

Proof. In Twelf. □

B.3.2 Soundness of Algorithmic Equality

The algorithm is only sound when its subjects are well-formed, so these have typing premises.

LEMMA B.21: SOUNDNESS OF WEAK HEAD REDUCTION.
If $\Delta \vdash \mathtt{C} :: \mathtt{K}$ *and* $\mathtt{C} \xrightarrow{\text{whr}} \mathtt{C}'$, *then* $\Delta \vdash \mathtt{C} \equiv \mathtt{C}' :: \mathtt{K}$.

Proof. In Twelf. □

THEOREM B.22: SOUNDNESS OF ALGORITHMIC EQUALITY.

1. *If* $\Delta \vdash \mathtt{K}\ \mathtt{kind}$, $\Delta \vdash \mathtt{K}'\ \mathtt{kind}$, *and* $(\Delta)^- \vdash \mathtt{K} \Longleftrightarrow \mathtt{K}'\ \mathtt{kind}$, *then* $\Delta \vdash \mathtt{K} \equiv \mathtt{K}'\ \mathtt{kind}$.
2. *If* $\Delta \vdash \mathtt{C} :: \mathtt{K}$, $\Delta \vdash \mathtt{C}' :: \mathtt{K}$, *and* $(\Delta)^- \vdash \mathtt{C} \Longleftrightarrow \mathtt{C}' :: (\mathtt{K})^-$, *then* $\Delta \vdash \mathtt{C} \equiv \mathtt{C}' :: \mathtt{K}$.
3. *If* $\Delta \vdash \mathtt{C} :: \mathtt{K}$, $\Delta \vdash \mathtt{C}' :: \mathtt{K}$, *and* $(\Delta)^- \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{C}'$, *then* $\Delta \vdash \mathtt{C} \equiv \mathtt{C}' :: \mathtt{K}$.
4. *If* $\Delta \vdash \mathtt{C} :: \mathtt{K}$, $\Delta \vdash \mathtt{C}' :: \mathtt{K}$, *and* $(\Delta)^- \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{NAT}} \mathtt{C}'$, *then* $\Delta \vdash \mathtt{C} \equiv \mathtt{C}' :: \mathtt{K}$.
5. *If* $\Delta \vdash \mathtt{C} :: \mathtt{K}$, $\Delta \vdash \mathtt{C}' :: \mathtt{K}$, *and* $(\Delta)^- \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{EQ_N}} \mathtt{C}'$, *then* $\Delta \vdash \mathtt{C} \equiv \mathtt{C}' :: \mathtt{K}$.
6. *If* $\Delta \vdash \mathtt{C} :: \mathtt{K}$, $\Delta \vdash \mathtt{C}' :: \mathtt{K}'$, *and* $(\Delta)^- \vdash \mathtt{C} \longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{L}}$, *then* $\Delta \vdash \mathtt{C} \equiv \mathtt{C}' :: \mathtt{K}$, $\Delta \vdash \mathtt{K} \equiv \mathtt{K}'\ \mathtt{kind}$, *and* $(\mathtt{K})^-$ *is* $(\mathtt{K}')^-$ *is* $\widehat{\mathtt{L}}$.

Proof. In Twelf. □

B.3.3 Completeness of Algorithmic Equality

Supporting Concepts

DEFINITION B.23: CONTEXT EXTENSION. A context Ψ' extends a context Ψ, written $\Psi' \geq \Psi$, iff Ψ' contains all declarations in Ψ and possibly more.

LEMMA B.24: ALGORITHMIC EQUALITY IS CLOSED UNDER CONTEXT EXTENSION.
For all algorithmic equality judgements J, if $\Psi \vdash J$ *and* $\Psi_+ \geq \Psi$ *then* $\Psi_+ \vdash J$.

Proof. Apply LEMMA B.14 repeatedly; this will terminate because all contexts are finite. □

DEFINITION B.25: SIMULTANEOUS SUBSTITUTIONS. Simultaneous substitutions are defined by the following grammar:

$$\sigma \quad ::= \quad \cdot \mid \sigma, \mathtt{C}/\mathtt{u}$$

Application of these substitutions is written on the right as $\mathtt{C}[\sigma]$ and $\mathtt{K}[s]$ to distinguish it from the previously-defined notion of substitution. Substitution application is defined by mutual induction on kinds and constructors. We maintain the invariant that all variables in the domain of a substitution are distinct; binding forms are tacitly α-renamed if necessary when we write $\sigma, \mathtt{u}/\mathtt{u}$ for a bound variable $\mathtt{u}$. Additionally, we only apply a substitution σ to an expression when σ substitutes for all free variables in the expression. Finally, we tacitly assume the usual side conditions that ensure capture-avoidance.

$$\begin{array}{rcl}
(\mathtt{TYPE})[\sigma] & = & \mathtt{TYPE} \\
(\Pi_{\mathtt{k}}\, \mathtt{u}{::}\mathtt{K}_2.\, \mathtt{K})[\sigma] & = & \Pi_{\mathtt{k}}\, \mathtt{u}{::}\mathtt{K}_2[\sigma].\, \mathtt{K}[\sigma, \mathtt{u}/\mathtt{u}] \\
(\mathtt{NAT})[\sigma] & = & \mathtt{NAT} \\
(\mathtt{EQ_N}(\mathtt{I}, \mathtt{J}))[\sigma] & = & \mathtt{EQ_N}(\mathtt{I}[\sigma], \mathtt{J}[\sigma]) \\
\\
(\mathtt{C}_1 \to \mathtt{C}_2)[\sigma] & = & \mathtt{C}_1[\sigma] \to \mathtt{C}_2[\sigma] \\
(\mathtt{C}_1 \times \mathtt{C}_2)[\sigma] & = & \mathtt{C}_1[\sigma] \times \mathtt{C}_2[\sigma] \\
(\mathtt{C}_1 + \mathtt{C}_2)[\sigma] & = & \mathtt{C}_1[\sigma] + \mathtt{C}_2[\sigma] \\
(\forall_{\mathtt{K}_2}\, \mathtt{C})[\sigma] & = & \forall_{\mathtt{K}_2[\sigma]}\, \mathtt{C}[\sigma] \\
(\exists_{\mathtt{K}_2}\, \mathtt{C})[\sigma] & = & \exists_{\mathtt{K}_2[\sigma]}\, \mathtt{C}[\sigma] \\
(\mathtt{unit})[\sigma] & = & \mathtt{unit} \\
(\mathtt{void})[\sigma] & = & \mathtt{void} \\
(\mathtt{nat}\ \mathtt{I})[\sigma] & = & \mathtt{nat}\ \mathtt{I}[\sigma] \\
(\mathtt{u})[\sigma, \mathtt{C}_2/\mathtt{u}, \sigma'] & = & \mathtt{C}_2 \\
(\lambda_{\mathtt{c}}\, \mathtt{u}{::}\mathtt{K}.\, \mathtt{C})[\sigma] & = & \lambda_{\mathtt{c}}\, \mathtt{u}{::}\mathtt{K}_2[\sigma].\, \mathtt{C}[\sigma, \mathtt{u}/\mathtt{u}] \\
(\mathtt{C}_1\ \mathtt{C}_2)[\sigma] & = & \mathtt{C}_1[\sigma]\ \mathtt{C}_2[\sigma] \\
(\mathtt{z})[\sigma] & = & \mathtt{z} \\
(\mathtt{s}\ \mathtt{I})[\sigma] & = & \mathtt{s}\ \mathtt{I}[\sigma] \\
(\mathtt{NATrec}[\mathtt{u.K}](\mathtt{I}, \mathtt{C_z}, \mathtt{i'.r.C_s}))[\sigma] & = & \mathtt{NATrec}[\mathtt{u.K}[\sigma, \mathtt{u}/\mathtt{u}]](\mathtt{I}[\sigma], \mathtt{C_z}[\sigma], \mathtt{i'.r.C_s}[\sigma, \mathtt{i'}/\mathtt{i'}, \mathtt{r}/\mathtt{r}]) \\
(\mathtt{eqn\_zz})[\sigma] & = & \mathtt{eqn\_zz} \\
(\mathtt{eqn\_ss}(\mathtt{I}, \mathtt{J}, \mathtt{P}))[\sigma] & = & \mathtt{eqn\_ss}(\mathtt{I}[\sigma], \mathtt{J}[\sigma], \mathtt{P}[\sigma]) \\
(\mathtt{EQ_Nrec}[\mathtt{i.j.p.K}](\mathtt{P}, \mathtt{C_{zz}}, \mathtt{i.j.p.r.C_{ss}}))[\sigma] & = & \mathtt{EQ_Nrec}[\mathtt{i.j.p.K}[\sigma, \mathtt{i}/\mathtt{i}, \mathtt{j}/\mathtt{j}, \mathtt{p}/\mathtt{p}]](\mathtt{P}[\sigma], \mathtt{C_{zz}}[\sigma], \mathtt{i.j.p.r.C_{ss}}[\sigma, \mathtt{i}/\mathtt{i}, \mathtt{j}/\mathtt{j}, \mathtt{p}/\mathtt{p}, \mathtt{r}/\mathtt{r}])
\end{array}$$

This definition gives substitutions that are simultaneous in the sense that $\mathtt{u}[\sigma, \mathtt{C}/\mathtt{u}, \sigma']$ is $\mathtt{C}$; the substitutions in σ and σ' are not applied to $\mathtt{C}$.

LEMMA B.26: SUBSTITUTION AND SIMULTANEOUS SUBSTITUTION.

1. *If* $\mathtt{u}$ *is not free in* $\mathtt{C}$ *then* $\mathtt{C}[\sigma, \mathtt{C}_2/\mathtt{u}, \sigma']$ *is* $\mathtt{C}[\sigma, \sigma']$*. If* $\mathtt{u}$ *is not free in* $\mathtt{K}$ *then* $\mathtt{K}[\sigma, \mathtt{C}_2/\mathtt{u}, \sigma']$ *is* $\mathtt{K}[\sigma, \sigma']$*.*

2. *For all* σ *and* σ' *such that* $\mathtt{u}$ *is not free,* $\mathtt{C}[\sigma, \mathtt{C}_2/\mathtt{u}, \sigma']$ *is* $[\mathtt{C}_2/\mathtt{u}](\mathtt{C}[\sigma, \mathtt{u}/\mathtt{u}, \sigma'])$*. For all* σ *and* σ' *where* $\mathtt{u}$ *is not free,* $\mathtt{K}[\sigma, \mathtt{C}_2/\mathtt{u}, \sigma']$ *is* $[\mathtt{C}_2/\mathtt{u}](\mathtt{K}[\sigma, \mathtt{u}/\mathtt{u}, \sigma'])$*.*

3. $\mathtt{C}[\sigma, \mathtt{C}_2[\sigma, \sigma']/\mathtt{u}, \sigma']$ *is* $([\mathtt{C}_2/\mathtt{u}]\mathtt{C})[\sigma, \sigma']$*.* $\mathtt{K}[\sigma, \mathtt{C}_2[\sigma, \sigma']/\mathtt{u}, \sigma']$ *is* $([\mathtt{K}/\mathtt{u}]\mathtt{C}_1)[\sigma, \sigma']$*.*

Proof. Each part is by mutual induction on $\mathtt{C}$ and $\mathtt{K}$. The third uses the first. □

Logical Relations A straightforward inductive proof of completeness breaks down because it is not obvious that algorithmic equality is a congruence for the elimination forms. Our solution is to use logical relations. The first relation, between two constructors, is defined by induction on erased kinds.

DEFINITION B.27: LOGICALLY RELATED CONSTRUCTORS.

1. $\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{TYPE}}]\!]$ iff $\Psi \vdash \mathtt{C} \Longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{TYPE}}$.

2. $\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{K}_2} \widehat{\to}_{\mathtt{k}} \widehat{\mathtt{K}}]\!]$ iff for all $\Psi_+ \geq \Psi$ and all $\mathtt{C}_2$ and $\mathtt{C}_2'$ such that Ψ_+ is well-formed and $\Psi_+ \vdash \mathtt{C}_2 = \mathtt{C}_2' \in [\![\widehat{\mathtt{K}_2}]\!]$, $\Psi_+ \vdash \mathtt{C}\,\mathtt{C}_2 = \mathtt{C}'\,\mathtt{C}_2' \in [\![\widehat{\mathtt{K}}]\!]$.

3. $\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{NAT}}]\!]$ is defined inductively as the least relation closed under the following inference rules:

$$\frac{\mathtt{C}_1 \xrightarrow{\text{whr}} \mathtt{C}_1' \quad \Psi \vdash \mathtt{C}_1' = \mathtt{C}_2 \in [\![\widehat{\mathtt{NAT}}]\!]}{\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2 \in [\![\widehat{\mathtt{NAT}}]\!]}\ \texttt{lr-nat-whr-left}$$

$$\frac{\mathtt{C}_2 \xrightarrow{\text{whr}} \mathtt{C}_2' \quad \Psi \vdash \mathtt{C}_1 = \mathtt{C}_2' \in [\![\widehat{\mathtt{NAT}}]\!]}{\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2 \in [\![\widehat{\mathtt{NAT}}]\!]} \ \texttt{lr-nat-whr-right}$$

$$\frac{\Psi \vdash \mathtt{C} \longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{NAT}}}{\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{NAT}}]\!]} \ \texttt{lr-nat-neut-eq}$$

$$\frac{}{\Psi \vdash \mathtt{z} = \mathtt{z} \in [\![\widehat{\mathtt{NAT}}]\!]} \ \texttt{lr-nat-z} \qquad \frac{\Psi \vdash \mathtt{I} = \mathtt{I}' \in [\![\widehat{\mathtt{NAT}}]\!]}{\Psi \vdash \mathtt{s}\,\mathtt{I} = \mathtt{s}\,\mathtt{I}' \in [\![\widehat{\mathtt{NAT}}]\!]} \ \texttt{lr-nat-s}$$

Because the logical relation at kind $\widehat{\mathtt{NAT}}$ is defined inductively, rule induction can be used to reason from the knowledge that $\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{NAT}}]\!]$.

4. $\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{EQ_N}}]\!]$ is defined inductively as the least relation closed under the following inference rules:

$$\frac{\mathtt{C}_1 \xrightarrow{\text{whr}} \mathtt{C}_1' \quad \Psi \vdash \mathtt{C}_1' = \mathtt{C}_2 \in [\![\widehat{\mathtt{EQ_N}}]\!]}{\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2 \in [\![\widehat{\mathtt{EQ_N}}]\!]} \ \texttt{lr-eqn-whr-left}$$

$$\frac{\mathtt{C}_2 \xrightarrow{\text{whr}} \mathtt{C}_2' \quad \Psi \vdash \mathtt{C}_1 = \mathtt{C}_2' \in [\![\widehat{\mathtt{EQ_N}}]\!]}{\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2 \in [\![\widehat{\mathtt{EQ_N}}]\!]} \ \texttt{lr-eqn-whr-right}$$

$$\frac{\Psi \vdash \mathtt{C} \longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{EQ_N}}}{\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{EQ_N}}]\!]} \ \texttt{lr-eqn-neut-eq}$$

$$\frac{}{\Psi \vdash \mathtt{eqn\_zz} = \mathtt{eqn\_zz} \in [\![\widehat{\mathtt{EQ_N}}]\!]} \ \texttt{lr-eqn-zz}$$

$$\frac{\Psi \vdash \mathtt{I} = \mathtt{I}' \in [\![\widehat{\mathtt{NAT}}]\!] \quad \Psi \vdash \mathtt{J} = \mathtt{J}' \in [\![\widehat{\mathtt{NAT}}]\!] \quad \Psi \vdash \mathtt{P} = \mathtt{P}' \in [\![\widehat{\mathtt{EQ_N}}]\!]}{\Psi \vdash \mathtt{eqn\_ss}(\mathtt{I},\mathtt{J},\mathtt{P}) = \mathtt{eqn\_ss}(\mathtt{I}',\mathtt{J}',\mathtt{P}') \in [\![\widehat{\mathtt{EQ_N}}]\!]} \ \texttt{lr-eqn-ss}$$

Next, we extend this to a relation between two substitutions; here, the logical relation is defined by induction on the structure of erased contexts.

DEFINITION B.28: LOGICALLY RELATED SUBSTITUTIONS.

1. $\Psi \vdash \sigma = \sigma' \in [\![\cdot]\!]$ iff σ is $\cdot$ and σ' is $\cdot$

2. $\Psi \vdash \sigma = \sigma' \in [\![\Theta, \mathtt{u} :: \widehat{\mathtt{K}}]\!]$ iff σ is $\sigma_1,\mathtt{C}/\mathtt{u}$ and σ' is $\sigma_1',\mathtt{C}'/\mathtt{u}$, where $\Psi \vdash \sigma_1 = \sigma_1' \in [\![\Theta]\!]$ and $\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{K}}]\!]$.

Logically Related Constructors are Algorithmically Equal

LEMMA B.29: LOGICALLY RELATED CONSTRUCTORS ARE ALGORITHMICALLY EQUAL.

1. *If* $\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{NAT}}]\!]$ *then* $\Psi \vdash \mathtt{C} \Longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{NAT}}$.
2. *If* $\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{EQ_N}}]\!]$ *then* $\Psi \vdash \mathtt{C} \Longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{EQ_N}}$.
3. *If* $\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{K}}]\!]$ *then* $\Psi \vdash \mathtt{C} \Longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{K}}$.
4. *If* $\Psi \vdash \mathtt{C} \longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{K}}$ *then* $\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{K}}]\!]$.

Proof. We prove the first part independently by rule induction on the assumed derivation. The second part is then proven by rule induction using the first. Then, the last two parts are proven by mutual induction on the classifying erased kind $\widehat{\mathtt{K}}$.

1.
   - Case for `lr-nat-whr-left`.
     By the IH, $\Psi \vdash \mathtt{C}_1' \Longleftrightarrow \mathtt{C}_2 :: \widehat{\mathtt{NAT}}$, so applying `norm-eq-cn-whr-left` to this and the premise reduction derivation (observe that $\widehat{\mathtt{NAT}}$ is a base kind) gives the result.
   - Case for `lr-nat-whr-right`.
     By the IH, $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow \mathtt{C}_2' :: \widehat{\mathtt{NAT}}$, so applying `norm-eq-cn-whr-right` to this and the premise reduction derivation (observe that $\widehat{\mathtt{NAT}}$ is a base kind) gives the result.
   - Case for `lr-nat-neut-eq`. By `norm-eq-cn/nat-neut-eq` applied to the premise equality derivation, $\Psi \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{NAT}} \mathtt{C}'$, so `norm-eq-cn-nat` gives the result.
   - Case for `lr-nat-z`. By `norm-eq-cn/nat-z`, $\Psi \vdash \mathtt{z} \Longleftrightarrow_{\mathtt{NAT}} \mathtt{z}$, so `norm-eq-cn-nat` gives the result.
   - Case for `lr-nat-s`. By the IH, $\Psi \vdash \mathtt{I} \Longleftrightarrow \mathtt{I}' :: \widehat{\mathtt{NAT}}$. By `norm-eq-cn/nat-s`, $\Psi \vdash \mathtt{s}\ \mathtt{I} \Longleftrightarrow_{\mathtt{NAT}} \mathtt{s}\ \mathtt{I}'$, so `norm-eq-cn-nat` gives the result.
2.
   - Case for `lr-eqn-whr-left`.
     By the IH, $\Psi \vdash \mathtt{C}_1' \Longleftrightarrow \mathtt{C}_2 :: \widehat{\mathtt{EQ_N}}$, so applying `norm-eq-cn-whr-left` to this and the premise reduction derivation (observe that $\widehat{\mathtt{EQ_N}}$ is a base kind) gives the result.
   - Case for `lr-eqn-whr-right`.
     By the IH, $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow \mathtt{C}_2' :: \widehat{\mathtt{EQ_N}}$, so applying `norm-eq-cn-whr-right` to this and the premise reduction derivation (observe that $\widehat{\mathtt{EQ_N}}$ is a base kind) gives the result.
   - Case for `lr-eqn-neut-eq`. By `norm-eq-cn/eqn-neut-eq` applied to the premise equality derivation, $\Psi \vdash \mathtt{C} \Longleftrightarrow_{\mathtt{EQ_N}} \mathtt{C}'$, so `norm-eq-cn-eqn` gives the result.
   - Case for `lr-eqn-zz`. By `norm-eq-cn/eqn-zz`, $\Psi \vdash \mathtt{eqn\_zz} \Longleftrightarrow_{\mathtt{EQ_N}} \mathtt{eqn\_zz}$, so `norm-eq-cn-eqn` gives the result.
   - Case for `lr-nat-s`. By the previous part, $\Psi \vdash \mathtt{I} \Longleftrightarrow \mathtt{I}' :: \widehat{\mathtt{NAT}}$ and $\Psi \vdash \mathtt{J} \Longleftrightarrow \mathtt{J}' :: \widehat{\mathtt{NAT}}$. By the IH, $\Psi \vdash \mathtt{P} \Longleftrightarrow \mathtt{P}' :: \widehat{\mathtt{EQ_N}}$. Then `norm-eq-cn/eqn-ss` and `norm-eq-cn-eqn` give the result.
3.
   - Case for $\widehat{\mathtt{TYPE}}$. Direct from the definition of the LR.

- Case for $\widehat{\mathtt{K}}_2 \widehat{\rightarrow}_{\mathtt{k}} \widehat{\mathtt{K}}$.
  $\Psi, \mathtt{u} :: \widehat{\mathtt{K}}_2 \vdash \mathtt{u} \longleftrightarrow \mathtt{u} :: \widehat{\mathtt{K}}_2$ — `neut-eq-cn-var`
  $\widehat{\mathtt{K}}_2$ is a subexpression of $\widehat{\mathtt{K}}_2 \widehat{\rightarrow}_{\mathtt{k}} \widehat{\mathtt{K}}$ — Def subexpr
  $\Psi, \mathtt{u} :: \widehat{\mathtt{K}}_2 \vdash \mathtt{u} = \mathtt{u} \in [\![\widehat{\mathtt{K}}_2]\!]$ — IH (4) on $\widehat{\mathtt{K}}_2$
  $\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{K}}_2 \widehat{\rightarrow}_{\mathtt{k}} \widehat{\mathtt{K}}]\!]$ — Assumption
  $\Psi, \mathtt{u} :: \widehat{\mathtt{K}} \geq \Psi$ — Def $\geq$
  $\Psi, \mathtt{u} :: \widehat{\mathtt{K}}_2 \vdash \mathtt{C}\,\mathtt{u} = \mathtt{C}'\,\mathtt{u} \in [\![\widehat{\mathtt{K}}]\!]$ — Def LR for $[\![\widehat{\mathtt{K}}_2 \widehat{\rightarrow}_{\mathtt{k}} \widehat{\mathtt{K}}]\!]$
  $\widehat{\mathtt{K}}$is a subexpression of $\widehat{\mathtt{K}}_2 \widehat{\rightarrow}_{\mathtt{k}} \widehat{\mathtt{K}}$ — Def subexpr
  $\Psi, \mathtt{u} :: \widehat{\mathtt{K}}_2 \vdash \mathtt{C}\,\mathtt{u} \Longleftrightarrow \mathtt{C}'\,\mathtt{u} :: \widehat{\mathtt{K}}$ — IH (3) on $\widehat{\mathtt{K}}$
  $\Psi \vdash \mathtt{C} \Longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{K}}_2 \widehat{\rightarrow}_{\mathtt{k}} \widehat{\mathtt{K}}$ — `norm-eq-cn-fn-ext`.
- Case for $\widehat{\mathtt{NAT}}$. Apply Part 1.
- Case for $\widehat{\mathtt{EQ}_\mathtt{N}}$. Apply Part 2.

4.
- Case for $\widehat{\mathtt{TYPE}}$. By `norm-eq-cn/type-neut-eq` and `norm-eq-cn-type` applied to the assumption, the constructors are normally equal; then the definition of $[\![\widehat{\mathtt{TYPE}}]\!]$ gives the result.
- Case for $\widehat{\mathtt{K}}_2 \widehat{\rightarrow}_{\mathtt{k}} \widehat{\mathtt{K}}$.
  By assumption, $\Psi \vdash \mathtt{C} \longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{K}}_2 \widehat{\rightarrow}_{\mathtt{k}} \widehat{\mathtt{K}}$. We are going to use the definition of the logical relation for $\widehat{\mathtt{K}}_2 \widehat{\rightarrow}_{\mathtt{k}} \widehat{\mathtt{K}}$, so assume for the "for all" arbitrary $\Psi_+ \geq \Psi$, $\mathtt{C}_2$ and $\mathtt{C}'_2$ such that
  $\Psi_+ \vdash \mathtt{C}_2 = \mathtt{C}'_2 \in [\![\widehat{\mathtt{K}}_2]\!]$. Then
  $\Psi_+ \vdash \mathtt{C}_2 \Longleftrightarrow \mathtt{C}'_2 :: \widehat{\mathtt{K}}_2$ — IH(3) applied to $\widehat{\mathtt{K}}_2$ and this assumption
  $\Psi_+ \vdash \mathtt{C} \longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{K}}_2 \widehat{\rightarrow}_{\mathtt{k}} \widehat{\mathtt{K}}$ — LEMMA B.24
  $\Psi_+ \vdash \mathtt{C}\,\mathtt{C}_2 \longleftrightarrow \mathtt{C}'\,\mathtt{C}_2 :: \widehat{\mathtt{K}}$ — `neut-eq-cn-app`
  $\Psi_+ \vdash \mathtt{C}\,\mathtt{C}_2 = \mathtt{C}'\,\mathtt{C}_2 \in [\![\widehat{\mathtt{K}}]\!]$ — IH (4) applied to $\widehat{\mathtt{K}}$
  This satisfies the "for all", so the result is true by the definition of the logical relation.
- Case for $\widehat{\mathtt{NAT}}$. `lr-nat-neut-eq` applied to the assumption gives the result.
- Case for $\widehat{\mathtt{EQ}_\mathtt{N}}$. `lr-eqn-neut-eq` applied to the assumption gives the result.

□

Definitionally Equal Constructors are Logically Related

LEMMA B.30: WEAKENING OF THE LOGICAL RELATIONS. *Assume $\Psi, \mathtt{u} :: \widehat{\mathtt{K}}_2, \Psi'$ is a well-formed context.*

1. *If $\Psi, \Psi' \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{NAT}}]\!]$ then $\Psi, \mathtt{u} :: \widehat{\mathtt{K}}_2, \Psi' \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{NAT}}]\!]$.*
2. *If $\Psi, \Psi' \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]$ then $\Psi, \mathtt{u} :: \widehat{\mathtt{K}}_2, \Psi' \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]$.*
3. *If $\Psi, \Psi' \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{K}}]\!]$ then $\Psi, \mathtt{u} :: \widehat{\mathtt{K}}_2, \Psi' \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{K}}]\!]$.*
4. *If $\Psi, \Psi' \vdash \sigma = \sigma' \in [\![\Theta]\!]$ then $\Psi, \mathtt{u} :: \widehat{\mathtt{K}}_2, \Psi' \vdash \sigma = \sigma' \in [\![\Theta]\!]$.*

Proof. First we prove Part 1 by rule induction; then, we prove Part 2 by rule induction using Part 1. Next, we prove Part 3 by induction on the erased kind; finally, we prove Part 4 by induction on the erased context.

1.
- Case for `lr-nat-whr-left`. By the IH, we can weaken the premise LR derivation, and then applying `lr-nat-whr-left` to this and the premise reduction derivation gives the result.
- Case for `lr-nat-whr-right`. By the IH, we can weaken the premise LR derivation, and then applying `lr-nat-whr-right` to this and the premise reduction derivation gives the result.

- Case for `lr-nat-neut-eq`. By LEMMA B.14 we can weaken the premise derivation; then applying `lr-nat-neut-eq` gives the result.
- Case for `lr-nat-z`. This case is immediate by `lr-nat-z` because the context in the result is arbitrary.
- Case for `lr-nat-s`. By the IH, we can weaken the derivation of $\Psi, \Psi' \vdash \mathtt{I} = \mathtt{I}' \in [\![\widehat{\mathtt{NAT}}]\!]$, and then applying `lr-nat-s` to this gives the result.

2. - Case for `lr-eqn-whr-left`. By the IH, we can weaken the premise LR derivation, and then applying `lr-eqn-whr-left` to this and the premise reduction derivation gives the result.
   - Case for `lr-eqn-whr-right`. By the IH, we can weaken the premise LR derivation, and then applying `lr-eqn-whr-right` to this and the premise reduction derivation gives the result.
   - Case for `lr-eqn-neut-eq`. By LEMMA B.14 we can weaken the premise derivation; then applying `lr-eqn-neut-eq` gives the result.
   - Case for `lr-eqn-zz`. This case is immediate by `lr-eqn-zz` because the context in the result is arbitrary.
   - Case for `lr-eqn-ss`. By the previous part, we can weaken the derivations of $\Psi, \Psi' \vdash \mathtt{I} = \mathtt{I}' \in [\![\widehat{\mathtt{NAT}}]\!]$ and $\Psi, \Psi' \vdash \mathtt{J} = \mathtt{J}' \in [\![\widehat{\mathtt{NAT}}]\!]$. By the IH, we can weaken the derivation of $\Psi, \Psi' \vdash \mathtt{P} = \mathtt{P}' \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]$. Then applying `lr-eqn-ss` gives the result.

3. - Case for $\widehat{\mathtt{TYPE}}$. By definition of $[\![\widehat{\mathtt{TYPE}}]\!]$, $\Psi, \Psi' \vdash \mathtt{C} \iff \mathtt{C}' :: \widehat{\mathtt{TYPE}}$, so by LEMMA B.14 $\Psi, \mathtt{u} :: \widehat{\mathtt{K}}_2, \Psi' \vdash \mathtt{C} \iff \mathtt{C}' :: \widehat{\mathtt{TYPE}}$; then the definition of $[\![\widehat{\mathtt{TYPE}}]\!]$ gives the result.
   - Case for $\widehat{\mathtt{K}}_\mathtt{f} \widehat{\rightarrow}_\mathtt{k} \widehat{\mathtt{K}}_\mathtt{t}$. We are going to use the definition of $[\![\widehat{\mathtt{K}}_2 \widehat{\rightarrow}_\mathtt{k} \widehat{\mathtt{K}}]\!]$, so assume for arbitrary $\Psi_+ \geq \Psi, \mathtt{u} :: \widehat{\mathtt{K}}_2, \Psi'$ and $\mathtt{C}_\mathtt{f}, \mathtt{C}_\mathtt{f}'$ that $\Psi_+ \vdash \mathtt{C}_\mathtt{f} = \mathtt{C}_\mathtt{f}' \in [\![\widehat{\mathtt{K}}_\mathtt{f}]\!]$. Observe that $\Psi, \mathtt{u} :: \widehat{\mathtt{K}}_2, \Psi'$ extends Ψ, Ψ', so by transitivity of extension $\Psi_+ \geq \Psi, \Psi'$. Then, by our assumption, $\Psi, \Psi' \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{K}}_\mathtt{f} \widehat{\rightarrow}_\mathtt{k} \widehat{\mathtt{K}}_\mathtt{t}]\!]$, so, by the definition of the LR, $\Psi_+ \vdash \mathtt{C}\,\mathtt{C}_\mathtt{f} = \mathtt{C}'\,\mathtt{C}_\mathtt{f}' \in [\![\widehat{\mathtt{K}}_\mathtt{t}]\!]$. By the definition of $[\![\widehat{\mathtt{K}}_\mathtt{f} \widehat{\rightarrow}_\mathtt{k} \widehat{\mathtt{K}}_\mathtt{t}]\!]$ (recall that we assumed an arbitrary Ψ_+ extending $\Psi, \mathtt{u} :: \widehat{\mathtt{K}}_2, \Psi'$) we have the result.
   - Case for $\widehat{\mathtt{NAT}}$. Apply Part 1.
   - Case for $\widehat{\mathtt{EQ}_\mathtt{N}}$. Apply Part 2.

4. - Case for $\cdot$. Immediate by the definition of $[\![\cdot]\!]$, as the context in the definition is arbitrary.
   - Case for $[\![\Theta, \mathtt{u} :: \widehat{\mathtt{K}}]\!]$. By assumption $\Psi, \Psi' \vdash \sigma = \sigma \in [\![\Theta, \mathtt{u} :: \widehat{\mathtt{K}}]\!]$, so by the definition of the LR σ is σ_1,C/u and σ' is σ_1',C$'$/u where $\Psi, \Psi' \vdash \sigma_1 = \sigma_1' \in [\![\Theta]\!]$ and $\Psi, \Psi' \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{K}}]\!]$. By the IH $\Psi, \mathtt{u} :: \widehat{\mathtt{K}}_2, \Psi' \vdash \sigma_1 = \sigma_1' \in [\![\Theta]\!]$ and by Part 3 $\Psi, \mathtt{u} :: \widehat{\mathtt{K}}_2, \Psi' \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{K}}]\!]$, so the definition of $[\![\Theta, \mathtt{u} :: \widehat{\mathtt{K}}]\!]$ gives the result.

□

LEMMA B.31: CLOSURE OF THE LOGICAL RELATIONS UNDER CONTEXT EXTENSION.

1. *If* $\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{K}}]\!]$ *and* $\Psi_+ \geq \Psi$ *then* $\Psi_+ \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{K}}]\!]$.

2. *If* $\Psi \vdash \sigma = \sigma' \in [\![\Theta]\!]$ *and* $\Psi_+ \geq \Psi$ *then* $\Psi_+ \vdash \sigma = \sigma' \in [\![\mathtt{\Theta}]\!]$.

Proof. Apply weakening repeatedly. □

LEMMA B.32: SYMMETRY OF THE LOGICAL RELATIONS.

1. *If* $\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{NAT}}]\!]$ *then* $\Psi \vdash \mathtt{C}' = \mathtt{C} \in [\![\widehat{\mathtt{NAT}}]\!]$.

2. *If* $\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{EQ_N}}]\!]$ *then* $\Psi \vdash \mathtt{C}' = \mathtt{C} \in [\![\widehat{\mathtt{EQ_N}}]\!]$.

3. *If* $\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{K}}]\!]$ *then* $\Psi \vdash \mathtt{C}' = \mathtt{C} \in [\![\widehat{\mathtt{K}}]\!]$.

4. *If* $\Psi \vdash \sigma = \sigma' \in [\![\Theta]\!]$ *then* $\Psi \vdash \sigma' = \sigma \in [\![\Theta]\!]$.

Proof. First we prove Part 1 by rule induction; then, we prove Part 2 by rule induction using Part 1. Next, we prove Part 3 by induction on the erased kind; finally, we prove Part 4 by induction on the erased context.

1. - Case for `lr-nat-whr-left`.
     By the IH, $\Psi \vdash \mathtt{C}_2 = \mathtt{C}'_1 \in [\![\widehat{\mathtt{NAT}}]\!]$, and then `lr-nat-whr-right` applied to this derivation and the premise reduction derivation gives the result.
   - Case for `lr-nat-whr-right`.
     By the IH, $\Psi \vdash \mathtt{C}'_2 = \mathtt{C}_1 \in [\![\widehat{\mathtt{NAT}}]\!]$, and then `lr-nat-whr-left` applied to this derivation and the premise reduction derivation gives the result.
   - Case for `lr-nat-neut-eq`.
     By symmetry of algorithmic equality (LEMMA B.19), we get the symmetric structural equality derivation, and then we apply `lr-nat-neut-eq` to get the result.
   - Case for `lr-nat-z`. Return the given derivation.
   - Case for `lr-nat-s`. By the IH, we get the symmetric premise derivation, and then we apply `lr-nat-s` to get the result.

2. - Case for `lr-eqn-whr-left`.
     By the IH, $\Psi \vdash \mathtt{C}_2 = \mathtt{C}'_1 \in [\![\widehat{\mathtt{EQ_N}}]\!]$, and then `lr-eqn-whr-right` applied to this derivation and the premise reduction derivation gives the result.
   - Case for `lr-eqn-whr-right`.
     By the IH, $\Psi \vdash \mathtt{C}'_2 = \mathtt{C}_1 \in [\![\widehat{\mathtt{EQ_N}}]\!]$, and then `lr-eqn-whr-left` applied to this derivation and the premise reduction derivation gives the result.
   - Case for `lr-eqn-neut-eq`. By symmetry of algorithmic equality (LEMMA B.19), we get the symmetric neutral equality derivation, and then we apply `lr-eqn-neut-eq` to get the result.
   - Case for `lr-eqn-zz`. Return the given derivation.
   - Case for `lr-eqn-ss`. By the previous part, we compute the symmetric derivations for $\widehat{\mathtt{NAT}}$. By the IH, we get the symmetric premise derivation for $\widehat{\mathtt{EQ_N}}$. Then we apply `lr-nat-s` to get the result.

3. - Case for $\widehat{\mathtt{TYPE}}$. By assumption in this case, $\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{TYPE}}]\!]$, so by the definition of $[\![\widehat{\mathtt{TYPE}}]\!]$, $\Psi \vdash \mathtt{C} \Longleftrightarrow \mathtt{C}' :: \widehat{\mathtt{TYPE}}$. By symmetry of algorithmic equality (LEMMA B.19), $\Psi \vdash \mathtt{C}' \Longleftrightarrow \mathtt{C} :: \widehat{\mathtt{TYPE}}$, and then the definition of $[\![\widehat{\mathtt{TYPE}}]\!]$ gives the result.
   - Case for $\widehat{\mathtt{K}}_2 \widehat{\rightarrow}_{\mathtt{k}} \widehat{\mathtt{K}}$. We are going to use the definition of $[\![\widehat{\mathtt{K}}_2 \widehat{\rightarrow}_{\mathtt{k}} \widehat{\mathtt{K}}]\!]$, so assume for the "for all" that for arbitrary $\Psi_+ \geq \Psi$, $\mathtt{C}_2$, and $\mathtt{C}'_2$, $\Psi_+ \vdash \mathtt{C}_2 = \mathtt{C}'_2 \in [\![\widehat{\mathtt{K}}_2]\!]$. We must show that
     $\Psi_+ \vdash \mathtt{C}'\,\mathtt{C}_2 = \mathtt{C}\,\mathtt{C}'_2 \in [\![\widehat{\mathtt{K}}]\!]$ so that the definition of the LR will give the result. By the IH applied to $\widehat{\mathtt{K}}_2$ and the assumption above, $\Psi_+ \vdash \mathtt{C}'_2 = \mathtt{C}_2 \in [\![\widehat{\mathtt{K}}_2]\!]$. By assumption in this case, $\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{K}}_2 \widehat{\rightarrow}_{\mathtt{k}} \widehat{\mathtt{K}}]\!]$, so by the definition of the logical relation,
     $\Psi_+ \vdash \mathtt{C}\,\mathtt{C}'_2 = \mathtt{C}'\,\mathtt{C}_2 \in [\![\widehat{\mathtt{K}}]\!]$. Then the IH on $\widehat{\mathtt{K}}$ gives what we needed to show.
   - Case for $\widehat{\mathtt{NAT}}$. Apply Part 1.
   - Case for $\widehat{\mathtt{EQ_N}}$. Apply Part 2.

4. • Case for $\cdot$. By assumption, $\Psi \vdash \sigma = \sigma' \in [\![\cdot]\!]$, so by the definition of the LR, σ is $\cdot$ and σ' is $\cdot$. Then the definition of the LR gives the result.

   • Case for $\Theta, \mathtt{u} :: \widehat{\mathtt{K}}$. By the definition of the LR σ is $\sigma_1, \mathtt{C}/\mathtt{u}$ and σ' is $\sigma_1', \mathtt{C}'/\mathtt{u}$ where $\Psi \vdash \sigma_1 = \sigma_1' \in [\![\Theta]\!]$ and $\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![\widehat{\mathtt{K}}]\!]$. By induction $\Psi \vdash \sigma_1' = \sigma_1 \in [\![\Theta]\!]$ and by Part 3 $\Psi \vdash \mathtt{C}' = \mathtt{C} \in [\![\widehat{\mathtt{K}}]\!]$, so the definition of the LR gives the result.

□

LEMMA B.33: TRANSITIVITY OF THE LOGICAL RELATIONS.

1. *If* $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2 \in [\![\widehat{\mathtt{NAT}}]\!]$ *and* $\Psi \vdash \mathtt{C}_2 = \mathtt{C}_3 \in [\![\widehat{\mathtt{NAT}}]\!]$ *then* $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_3 \in [\![\widehat{\mathtt{NAT}}]\!]$.
2. *If* $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2 \in [\![\widehat{\mathtt{EQ_N}}]\!]$ *and* $\Psi \vdash \mathtt{C}_2 = \mathtt{C}_3 \in [\![\widehat{\mathtt{EQ_N}}]\!]$ *then* $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_3 \in [\![\widehat{\mathtt{EQ_N}}]\!]$.
3. *If* $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2 \in [\![\widehat{\mathtt{K}}]\!]$ *and* $\Psi \vdash \mathtt{C}_2 = \mathtt{C}_3 \in [\![\widehat{\mathtt{K}}]\!]$ *then* $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_3 \in [\![\widehat{\mathtt{K}}]\!]$.
4. *If* $\Psi \vdash \sigma_1 = \sigma_2 \in [\![\Theta]\!]$ *and* $\Psi \vdash \sigma_2 = \sigma_3 \in [\![\Theta]\!]$ *then* $\Psi \vdash \sigma_1 = \sigma_3 \in [\![\Theta]\!]$.

Proof. First we prove Part 1 by rule induction; then, we prove Part 2 by rule induction using Part 1, Next, we prove Part 3 by induction on the erased kind; finally, we prove Part 4 by induction on the erased context.

1. The proof is by mutual lexicographic induction on the derivations of $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2 \in [\![\widehat{\mathtt{NAT}}]\!]$ and $\Psi \vdash \mathtt{C}_2 = \mathtt{C}_3 \in [\![\widehat{\mathtt{NAT}}]\!]$.

   • Case for

$$\frac{\mathtt{C}_1 \xrightarrow{\text{whr}} \mathtt{C}_1' \quad \Psi \vdash \mathtt{C}_1' = \mathtt{C}_2 \in [\![\widehat{\mathtt{NAT}}]\!]}{\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2 \in [\![\widehat{\mathtt{NAT}}]\!]}\ \texttt{lr-nat-whr-left} \qquad \mathcal{D}_2 \text{ arbitrary.}$$

   By the IH applied to the premise derivation of $\Psi \vdash \mathtt{C}_1' = \mathtt{C}_2 \in [\![\widehat{\mathtt{NAT}}]\!]$ and $\mathcal{D}_2$ (note that one is smaller while the other is the same), $\Psi \vdash \mathtt{C}_1' = \mathtt{C}_3 \in [\![\widehat{\mathtt{NAT}}]\!]$. Then `lr-nat-whr-left` applied to this and the premise reduction derivation gives the result.

   • Case for

$$\mathcal{D}_1 \text{ arbitrary} \qquad \frac{\mathtt{C}_3 \xrightarrow{\text{whr}} \mathtt{C}_3' \quad \Psi \vdash \mathtt{C}_2 = \mathtt{C}_3' \in [\![\widehat{\mathtt{NAT}}]\!]}{\Psi \vdash \mathtt{C}_2 = \mathtt{C}_3 \in [\![\widehat{\mathtt{NAT}}]\!]}\ \texttt{lr-nat-whr-right} \quad .$$

   By the IH applied to the premise derivation of $\Psi \vdash \mathtt{C}_2 = \mathtt{C}_3' \in [\![\widehat{\mathtt{NAT}}]\!]$ and $\mathcal{D}_1$ (note that one is smaller while the other is the same), $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_3' \in [\![\widehat{\mathtt{NAT}}]\!]$. Then `lr-nat-whr-right` applied to this and the premise reduction derivation gives the result.

   • Case for

$$\frac{\mathtt{C}_2 \xrightarrow{\text{whr}} \mathtt{C}_2' \quad \Psi \vdash \mathtt{C}_1 = \mathtt{C}_2' \in [\![\widehat{\mathtt{NAT}}]\!]}{\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2 \in [\![\widehat{\mathtt{NAT}}]\!]}\ \texttt{lr-nat-whr-right}$$

$$\frac{\mathtt{C}_2 \xrightarrow{\text{whr}} \mathtt{C}_2'' \quad \Psi \vdash \mathtt{C}_2'' = \mathtt{C}_3 \in [\![\widehat{\mathtt{NAT}}]\!]}{\Psi \vdash \mathtt{C}_2 = \mathtt{C}_3 \in [\![\widehat{\mathtt{NAT}}]\!]}\ \texttt{lr-nat-whr-left} \quad .$$

   By determinacy of weak head reduction (LEMMA B.15), $\mathtt{C}_2'$ is $\mathtt{C}_2''$, so the RHS premise really derives $\Psi \vdash \mathtt{C}_2' = \mathtt{C}_3 \in [\![\widehat{\mathtt{NAT}}]\!]$. Then the IH on the two premise derivations (note that both are smaller) gives the result.

- Case for

$$\frac{\Psi \vdash \mathtt{C}_1 \longleftrightarrow \mathtt{C}_2 :: \widehat{\mathtt{NAT}}}{\Psi \vdash \mathtt{C}_1 \;=\; \mathtt{C}_2 \;\in\; \widehat{[\![\mathtt{NAT}]\!]}}\ \mathtt{lr\text{-}nat\text{-}neut\text{-}eq}$$

$$\frac{\Psi \vdash \mathtt{C}_2 \longleftrightarrow \mathtt{C}_3 :: \widehat{\mathtt{NAT}}}{\Psi \vdash \mathtt{C}_2 \;=\; \mathtt{C}_3 \;\in\; \widehat{[\![\mathtt{NAT}]\!]}}\ \mathtt{lr\text{-}nat\text{-}neut\text{-}eq} \quad .$$

  Transitivity of algorithmic equality (LEMMA B.20) applied to the premises gives $\Psi \vdash \mathtt{C}_1 \longleftrightarrow \mathtt{C}_3 :: \widehat{\mathtt{NAT}}$, so we can apply `lr-nat-neut-eq` to this derivation to get the result.

- Case when both premises were derived using an application of `lr-nat-z` as the final rule. Apply `lr-nat-z`.

- Case for

$$\frac{\Psi \vdash \mathtt{I}_1 \;=\; \mathtt{I}_2 \;\in\; \widehat{[\![\mathtt{NAT}]\!]}}{\Psi \vdash \mathtt{s}\,\mathtt{I}_1 \;=\; \mathtt{s}\,\mathtt{I}_2 \;\in\; \widehat{[\![\mathtt{NAT}]\!]}}\ \mathtt{lr\text{-}nat\text{-}s} \qquad \frac{\Psi \vdash \mathtt{I}_2 \;=\; \mathtt{I}_3 \;\in\; \widehat{[\![\mathtt{NAT}]\!]}}{\Psi \vdash \mathtt{s}\,\mathtt{I}_2 \;=\; \mathtt{s}\,\mathtt{I}_3 \;\in\; \widehat{[\![\mathtt{NAT}]\!]}}\ \mathtt{lr\text{-}nat\text{-}s} \quad .$$

  By the IH on the premises (note that both are smaller), $\Psi \vdash \mathtt{I}_1 \;=\; \mathtt{I}_3 \;\in\; \widehat{[\![\mathtt{NAT}]\!]}$, so `lr-nat-s` gives the result.

- All other cases are contradictory. So far, we have covered

LHS	RHS
lr-nat-whr-left	_
_	lr-nat-whr-right
lr-nat-whr-right	lr-nat-whr-left
lr-nat-neut-eq	lr-nat-neut-eq
lr-nat-z	lr-nat-z
lr-nat-s	lr-nat-s

  We derive contradictions in each remaining case as follows:

  `lr-whr-nat-right` vs. `lr-nat-z`, `-s`, or `-neut-eq`: The premise of the LHS derivation is that $\mathtt{C}_2 \xrightarrow{\text{whr}} \mathtt{C}_2'$. When the RHS derivation is `lr-nat-z`, $\mathtt{C}_2$ is $\mathtt{z}$; this is contradictory by inversion because no rule derives head reduction for the syntactic form $\mathtt{z}$. When the RHS derivation is `lr-nat-s`, we similarly get a contradiction by inversion because this head reduction derivation is impossible. For `-neut-eq`, by LEMMA B.17 and LEMMA B.18 we get a contradiction.

  `lr-nat-z`, `-s`, or `-neut` vs. `lr-nat-left`: The premise of the RHS derivation is that $\mathtt{C}_2 \xrightarrow{\text{whr}} \mathtt{C}_2'$, so we get the same contradictions as in the above cases.

  This leaves the off-diagonals of `-z`, `-s`, and `-neut-eq`. For `-z` vs. `-s` and `-s` vs. `-z`, we get a contradiction because $\mathtt{C}_2$ cannot syntactically be both $\mathtt{z}$ and $\mathtt{s}\ \mathtt{I}_2$. For `-z` or `-s` vs. `-neut` and the symmetric case, we get a contradiction by inversion because $\mathtt{C}_2$ is $\mathtt{z}$ or $\mathtt{s}\ \mathtt{I}_2$, so we have a derivation of neutral equality where one side is $\mathtt{C}_2$ is $\mathtt{z}$ or $\mathtt{s}\ \mathtt{I}_2$, but no inference rule for neutral equality derives these conclusions.

  Thus, we get the result vacuously in each of these cases.

2. The proof is by mutual lexicographic induction on the derivations of $\Psi \vdash \mathtt{C}_1 \;=\; \mathtt{C}_2 \;\in\; \widehat{[\![\mathtt{EQ}_\mathtt{N}]\!]}$ and $\Psi \vdash \mathtt{C}_2 \;=\; \mathtt{C}_3 \;\in\; \widehat{[\![\mathtt{EQ}_\mathtt{N}]\!]}$.

- Case for

$$\frac{\mathtt{C}_1 \xrightarrow{\text{whr}} \mathtt{C}_1' \quad \Psi \vdash \mathtt{C}_1' = \mathtt{C}_2 \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]}{\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2 \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]}\ \mathtt{lr\text{-}eqn\text{-}whr\text{-}left} \qquad \mathcal{D}_2 \text{ arbitrary.}$$

  By the IH applied to the premise derivation of $\Psi \vdash \mathtt{C}_1' = \mathtt{C}_2 \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]$ and $\mathcal{D}_2$ (note that one is smaller while the other is the same), $\Psi \vdash \mathtt{C}_1' = \mathtt{C}_3 \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]$. Then `lr-eqn-whr-left` applied to this and the premise reduction derivation gives the result.

- Case for

$$\mathcal{D}_1 \text{ arbitrary} \qquad \frac{\mathtt{C}_3 \xrightarrow{\text{whr}} \mathtt{C}_3' \quad \Psi \vdash \mathtt{C}_2 = \mathtt{C}_3' \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]}{\Psi \vdash \mathtt{C}_2 = \mathtt{C}_3 \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]}\ \mathtt{lr\text{-}eqn\text{-}whr\text{-}right} \quad .$$

  By the IH applied to the premise derivation of $\Psi \vdash \mathtt{C}_2 = \mathtt{C}_3' \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]$ and $\mathcal{D}_1$ (note that one is smaller while the other is the same), $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_3' \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]$. Then `lr-eqn-whr-right` applied to this and the premise reduction derivation gives the result.

- Case for

$$\frac{\mathtt{C}_2 \xrightarrow{\text{whr}} \mathtt{C}_2' \quad \Psi \vdash \mathtt{C}_1 = \mathtt{C}_2' \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]}{\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2 \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]}\ \mathtt{lr\text{-}eqn\text{-}whr\text{-}right}$$

$$\frac{\mathtt{C}_2 \xrightarrow{\text{whr}} \mathtt{C}_2'' \quad \Psi \vdash \mathtt{C}_2'' = \mathtt{C}_3 \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]}{\Psi \vdash \mathtt{C}_2 = \mathtt{C}_3 \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]}\ \mathtt{lr\text{-}eqn\text{-}whr\text{-}left} \quad .$$

  By determinacy of weak head reduction (LEMMA B.15), $\mathtt{C}_2'$ is $\mathtt{C}_2''$, so the RHS premise really derives $\Psi \vdash \mathtt{C}_2' = \mathtt{C}_3 \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]$. Then the IH on the two premise derivations (note that both are smaller) gives the result.

- Case for

$$\frac{\Psi \vdash \mathtt{C}_1 \longleftrightarrow \mathtt{C}_2 :: \widehat{\mathtt{EQ}_\mathtt{N}}}{\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2 \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]}\ \mathtt{lr\text{-}nat\text{-}neut\text{-}eq}$$

$$\frac{\Psi \vdash \mathtt{C}_2 \longleftrightarrow \mathtt{C}_3 :: \widehat{\mathtt{EQ}_\mathtt{N}}}{\Psi \vdash \mathtt{C}_2 = \mathtt{C}_3 \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]}\ \mathtt{lr\text{-}nat\text{-}neut\text{-}eq} \quad .$$

  Transitivity of algorithmic equality (LEMMA B.20) applied to the premises gives
  $\Psi \vdash \mathtt{C}_1 \longleftrightarrow \mathtt{C}_3 :: \widehat{\mathtt{EQ}_\mathtt{N}}$, so we can apply `lr-eqn-neut-eq` to this derivation to get the result.

- Case when both premises were derived using an application of `lr-eqn-zz` as the final rule. Apply `lr-eqn-zz`.

- Case for

$$\frac{\Psi \vdash \mathtt{I}_1 = \mathtt{I}_2 \in [\![\widehat{\mathtt{NAT}}]\!] \quad \Psi \vdash \mathtt{J}_1 = \mathtt{J}_2 \in [\![\widehat{\mathtt{NAT}}]\!] \quad \Psi \vdash \mathtt{P}_1 = \mathtt{P}_2 \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]}{\Psi \vdash \mathtt{eqn\_ss}(\mathtt{I}_1, \mathtt{J}_1, \mathtt{P}_1) = \mathtt{eqn\_ss}(\mathtt{I}_2, \mathtt{J}_2, \mathtt{P}_2) \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]}\ \mathtt{lr\text{-}eqn\text{-}ss}$$

$$\frac{\Psi \vdash \mathtt{I}_2 = \mathtt{I}_3 \in [\![\widehat{\mathtt{NAT}}]\!] \quad \Psi \vdash \mathtt{J}_2 = \mathtt{J}_3 \in [\![\widehat{\mathtt{NAT}}]\!] \quad \Psi \vdash \mathtt{P}_2 = \mathtt{P}_3 \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]}{\Psi \vdash \mathtt{eqn\_ss}(\mathtt{I}_2, \mathtt{J}_2, \mathtt{P}_2) = \mathtt{eqn\_ss}(\mathtt{I}_3, \mathtt{J}_3, \mathtt{P}_3) \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]}\ \mathtt{lr\text{-}eqn\text{-}ss} \quad .$$

  By the previous part, $\Psi \vdash \mathtt{I}_1 = \mathtt{I}_3 \in [\![\widehat{\mathtt{NAT}}]\!]$ and $\Psi \vdash \mathtt{J}_1 = \mathtt{J}_3 \in [\![\widehat{\mathtt{NAT}}]\!]$. By the IH on the premises (note that both are smaller), $\Psi \vdash \mathtt{P}_1 = \mathtt{P}_3 \in [\![\widehat{\mathtt{EQ}_\mathtt{N}}]\!]$, so `lr-eqn-ss` gives the result.

- All other cases are contradictory. We derive contradictions in each remaining case as follows:

  `lr-eqn-whr-right` vs. `lr-eqn-zz`, `-s`, or `-neut-eq`: The premise of the LHS derivation is that $\mathtt{C}_2 \xrightarrow{\text{whr}} \mathtt{C}_2'$. When the RHS derivation is `lr-eqn-zz`, $\mathtt{C}_2$ is $\mathtt{eqn\_zz}$; this is contradictory by inversion because no rule derives head reduction for the syntactic form. When the RHS derivation is `lr-eqn-ss`, we similarly get a contradiction by inversion because this head reduction derivation is impossible. For `-neut-eq`, by LEMMA B.17 and LEMMA B.18 we get a contradiction.

  `lr-eqn-zz`, `-ss`, or `-neut` vs. `lr-eqn-whr-left`: The premise of the RHS derivation is that $\mathtt{C}_2 \xrightarrow{\text{whr}} \mathtt{C}_2'$, so we get the same contradictions as in the above case.

  This leaves the off-diagonals of `-zz`, `-ss`, and `-neut-eq`. For `-zz` vs. `-ss` and `-ss` vs. `-zz`, we get a contradiction because $\mathtt{C}_2$ cannot syntactically be both $\mathtt{eqn\_zz}$ and $\mathtt{eqn\_ss}(\mathtt{X},\mathtt{Y},\mathtt{Z})$. For `-zz` or `-ss` vs. `-neut` and the symmetric case, we get a contradiction by inversion because $\mathtt{C}_2$ is $\mathtt{eqn\_zz}$ or $\mathtt{eqn\_ss}(\mathtt{X},\mathtt{Y},\mathtt{Z})$, so we have a derivation of neutral equality where one side is $\mathtt{C}_2$ is $\mathtt{eqn_zz}$or $\mathtt{eqn\_ss}(\mathtt{X},\mathtt{Y},\mathtt{Z})$, but no inference rule for neutral equality derives these conclusions.

  Thus, we get the result vacuously in each of these cases.

3.
   - Case for $\widehat{\mathtt{TYPE}}$. By the definition of $[\![\widehat{\mathtt{TYPE}}]\!]$ on both of the assumptions, we get the two algorithmic equalities. Then transitivity of algorithmic equality (LEMMA B.20) gives $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_3 \in [\![\widehat{\mathtt{TYPE}}]\!]$, so the definition of $[\![\widehat{\mathtt{TYPE}}]\!]$ produces the result.
   - Case for $\widehat{\mathtt{K}}_\mathtt{f} \widehat{\rightarrow}_\mathtt{k} \widehat{\mathtt{K}}_\mathtt{t}$. Assume for "for all" that for arbitrary $\Psi_+ \geq \Psi$ and $\mathtt{C}_\mathtt{f}$, $\mathtt{C}_\mathtt{f}'$, $\Psi_+ \vdash \mathtt{C}_\mathtt{f} = \mathtt{C}_\mathtt{f}' \in [\![\widehat{\mathtt{K}}_\mathtt{f}]\!]$. We must show that $\Psi_+ \vdash \mathtt{C}_1\,\mathtt{C}_\mathtt{f} = \mathtt{C}_3\,\mathtt{C}_\mathtt{f}' \in [\![\widehat{\mathtt{K}}_\mathtt{t}]\!]$ to get the result by the definition of the LR. By our first assumption, $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2 \in [\![\widehat{\mathtt{K}}_\mathtt{f} \widehat{\rightarrow}_\mathtt{k} \widehat{\mathtt{K}}_\mathtt{t}]\!]$, so by the definition of the logical relation applied to this, $\Psi_+ \vdash \mathtt{C}_1\,\mathtt{C}_\mathtt{f} = \mathtt{C}_2\,\mathtt{C}_\mathtt{f}' \in [\![\widehat{\mathtt{K}}_\mathtt{t}]\!]$. By symmetry (LEMMA B.32), $\Psi_+ \vdash \mathtt{C}_\mathtt{f}' = \mathtt{C}_\mathtt{f} \in [\![\widehat{\mathtt{K}}_\mathtt{f}]\!]$, and then by induction applied to $\widehat{\mathtt{K}}_\mathtt{f}$ and these two symmetric statements, $\Psi_+ \vdash \mathtt{C}_\mathtt{f}' = \mathtt{C}_\mathtt{f}' \in [\![\widehat{\mathtt{K}}_\mathtt{f}]\!]$. But then by the definition of the LR applied to the other assumption, $\Psi_+ \vdash \mathtt{C}_2\,\mathtt{C}_\mathtt{f}' = \mathtt{C}_3\,\mathtt{C}_\mathtt{f}' \in [\![\widehat{\mathtt{K}}_\mathtt{t}]\!]$. Then induction applied to $\widehat{\mathtt{K}}_\mathtt{t}$ lets us put these together into what we needed to show.
   - Case for $\widehat{\mathtt{NAT}}$. Apply Part 1.
   - Case for $\widehat{\mathtt{EQ}_\mathtt{N}}$. Apply Part 2.

4.
   - Case for $\cdot$. By assumption, $\Psi \vdash \sigma_1 = \sigma_2 \in [\![\cdot]\!]$ and $\Psi \vdash \sigma_2 = \sigma_3 \in [\![\cdot]\!]$. By definition of the LR applied to the first premise, $\mathtt{C}_1$ is $\cdot$; by the definition of the LR applied to the second premise, $\mathtt{C}_3$ is $\cdot$. The definition of the LR applied to these two facts gives the result.
   - Case for $\Theta, \mathtt{u} :: \widehat{\mathtt{K}}$. By assumption, $\Psi \vdash \sigma_1 = \sigma_2 \in [\![\Theta, \mathtt{u} :: \widehat{\mathtt{K}}]\!]$ and $\Psi \vdash \sigma_2 = \sigma_3 \in [\![\Theta, \mathtt{u} :: \widehat{\mathtt{K}}]\!]$. By the definition of the LR, σ_1 is σ_1',$\mathtt{C}_1/\mathtt{u}$ and σ_2 is σ_2',$\mathtt{C}_2/\mathtt{u}$ where $\Psi \vdash \sigma_1' = \sigma_2' \in [\![\Theta]\!]$ and $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2 \in [\![\widehat{\mathtt{K}}]\!]$; σ_2 is σ_2'',$\mathtt{C}_2'/\mathtt{u}$ and σ_3 is σ_3',$\mathtt{C}_3/\mathtt{u}$ where $\Psi \vdash \sigma_2'' = \sigma_3' \in [\![\Theta]\!]$ and $\Psi \vdash \mathtt{C}_2' = \mathtt{C}_3 \in [\![\widehat{\mathtt{K}}]\!]$. But σ_2', $\mathtt{C}_2/\mathtt{u}$ is σ_2 is σ_2'', $\mathtt{C}_2'/\mathtt{u}$, so, σ_2' is σ_2'' and $\mathtt{C}_2$ is $\mathtt{C}_2'$. Thus, we can apply the IH to $\Psi \vdash \sigma_1' = \sigma_2' \in [\![\Theta]\!]$ and $\Psi \vdash \sigma_2' = \sigma_3' \in [\![\Theta]\!]$ to get $\Psi \vdash \sigma_1' = \sigma_3' \in [\![\Theta]\!]$ and use Part 3 on $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2 \in [\![\widehat{\mathtt{K}}]\!]$ and $\Psi \vdash \mathtt{C}_2 = \mathtt{C}_3 \in [\![\widehat{\mathtt{K}}]\!]$ to get $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_3 \in [\![\widehat{\mathtt{K}}]\!]$. Then the definition of $[\![\Theta, \mathtt{u} :: \widehat{\mathtt{K}}]\!]$ gives the result.

□

LEMMA B.34: LOGICAL RELATION IS CLOSED UNDER HEAD EXPANSION.

1. *If* $\Psi \vdash \mathtt{C}_1' = \mathtt{C}_2 \in [\![\widehat{\mathtt{K}}]\!]$ *and* $\mathtt{C}_1 \xrightarrow{\text{whr}} \mathtt{C}_1'$ *then* $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2 \in [\![\widehat{\mathtt{K}}]\!]$.

2. *If* $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2' \in [\![\widehat{\mathtt{K}}]\!]$ *and* $\mathtt{C}_2 \xrightarrow{\text{whr}} \mathtt{C}_2'$ *then* $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2 \in [\![\widehat{\mathtt{K}}]\!]$.

Proof. The proof in each case is by induction on the classifying erased kind. We first prove Part 1 and then Part 2.

1. - Case for $\widehat{\mathtt{TYPE}}$. By assumption, $\Psi \vdash \mathtt{C}_1' = \mathtt{C}_2 \in [\![\widehat{\mathtt{TYPE}}]\!]$, so by the definition of the LR, $\Psi \vdash \mathtt{C}_1' \iff \mathtt{C}_2 :: \widehat{\mathtt{TYPE}}$. By `norm-eq-cn-whr-left` applied to this and the head reduction derivation (observe that $\widehat{\mathtt{TYPE}}$ is a base kind), we get algorithmic equality; then the definition of $[\![\widehat{\mathtt{TYPE}}]\!]$ gives the result.
   - Case for $\widehat{\mathtt{K}}_\mathtt{f} \widehat{\rightarrow}_\mathtt{k} \widehat{\mathtt{K}}_\mathtt{t}$. We are going to show $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2 \in [\![\widehat{\mathtt{K}}_\mathtt{f} \widehat{\rightarrow}_\mathtt{k} \widehat{\mathtt{K}}_\mathtt{t}]\!]$ using the definition of the LR, so assume for the "for all" arbitrary $\Psi_+ \geq \Psi$, $\mathtt{C}_\mathtt{f}$ and $\mathtt{C}_\mathtt{f}'$ such that $\Psi_+ \vdash \mathtt{C}_\mathtt{f} = \mathtt{C}_\mathtt{f}' \in [\![\widehat{\mathtt{K}}_\mathtt{f}]\!]$. By assumption, $\Psi \vdash \mathtt{C}_1' = \mathtt{C}_2 \in [\![\widehat{\mathtt{K}}_\mathtt{f} \widehat{\rightarrow}_\mathtt{k} \widehat{\mathtt{K}}_\mathtt{t}]\!]$, so by the definition of the LR applied to this, $\Psi_+ \vdash \mathtt{C}_1' \, \mathtt{C}_\mathtt{f} = \mathtt{C}_2 \, \mathtt{C}_\mathtt{f}' \in [\![\widehat{\mathtt{K}}_\mathtt{t}]\!]$. By our other assumption, $\mathtt{C}_1 \xrightarrow{\text{whr}} \mathtt{C}_1'$, so by `whr-app-1` applied to this derivation $\mathtt{C}_1 \, \mathtt{C}_\mathtt{f} \xrightarrow{\text{whr}} \mathtt{C}_1' \, \mathtt{C}_\mathtt{f}$. Then by the IH applied to $\widehat{\mathtt{K}}_\mathtt{t}$, this fact, and $\Psi_+ \vdash \mathtt{C}_1' \, \mathtt{C}_\mathtt{f} = \mathtt{C}_2 \, \mathtt{C}_\mathtt{f}' \in [\![\widehat{\mathtt{K}}_\mathtt{t}]\!]$, we get that $\Psi_+ \vdash \mathtt{C}_1 \, \mathtt{C}_\mathtt{f} = \mathtt{C}_2 \, \mathtt{C}_\mathtt{f}' \in [\![\widehat{\mathtt{K}}_\mathtt{t}]\!]$. This is what we needed to show to satisfy the "for all", so the definition of the LR gives the result.
   - Case for $\widehat{\mathtt{NAT}}$. The assumptions are exactly the premises of `lr-nat-whr-left`, which then derives the conclusion.
   - Case for $\widehat{\mathtt{EQ_N}}$. The assumptions are exactly the premises of `lr-eqn-whr-left`, which then derives the conclusion.

2. This is mostly the same as the previous part.
   - Case for $\widehat{\mathtt{TYPE}}$. By assumption, $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2' \in [\![\widehat{\mathtt{TYPE}}]\!]$, so by the definition of the LR, $\Psi \vdash \mathtt{C}_1 \iff \mathtt{C}_2' :: \widehat{\mathtt{TYPE}}$. By `norm-eq-cn-whr-right` applied to this and the head reduction derivation (observe that $\widehat{\mathtt{TYPE}}$is a base kind), we get algorithmic equality; then the definition of $[\![\widehat{\mathtt{TYPE}}]\!]$ gives the result.
   - Case for $\widehat{\mathtt{K}}_\mathtt{f} \widehat{\rightarrow}_\mathtt{k} \widehat{\mathtt{K}}_\mathtt{t}$. We are going to show $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2 \in [\![\widehat{\mathtt{K}}_\mathtt{f} \widehat{\rightarrow}_\mathtt{k} \widehat{\mathtt{K}}_\mathtt{t}]\!]$ using the definition of the LR, so assume for the "for all" arbitrary $\Psi_+ \geq \Psi$, $\mathtt{C}_\mathtt{f}$ and $\mathtt{C}_\mathtt{f}'$ such that $\Psi_+ \vdash \mathtt{C}_\mathtt{f} = \mathtt{C}_\mathtt{f}' \in [\![\widehat{\mathtt{K}}_\mathtt{f}]\!]$. By assumption, $\Psi \vdash \mathtt{C}_1 = \mathtt{C}_2' \in [\![\widehat{\mathtt{K}}_\mathtt{f} \widehat{\rightarrow}_\mathtt{k} \widehat{\mathtt{K}}_\mathtt{t}]\!]$, so by the definition of the LR applied to this, $\Psi_+ \vdash \mathtt{C}_1 \, \mathtt{C}_\mathtt{f} = \mathtt{C}_2' \, \mathtt{C}_\mathtt{f}' \in [\![\widehat{\mathtt{K}}_\mathtt{t}]\!]$. By our other assumption, $\mathtt{C}_2 \xrightarrow{\text{whr}} \mathtt{C}_2'$, so by `whr-app-1` applied to this derivation $\mathtt{C}_2 \, \mathtt{C}_\mathtt{f}' \xrightarrow{\text{whr}} \mathtt{C}_2' \, \mathtt{C}_\mathtt{f}'$. Then by the IH applied to $\widehat{\mathtt{K}}_\mathtt{t}$, this fact, and $\Psi_+ \vdash \mathtt{C}_1 \, \mathtt{C}_\mathtt{f} = \mathtt{C}_2' \, \mathtt{C}_\mathtt{f}' \in [\![\widehat{\mathtt{K}}_\mathtt{t}]\!]$, we get that $\Psi_+ \vdash \mathtt{C}_1 \, \mathtt{C}_\mathtt{f} = \mathtt{C}_2 \, \mathtt{C}_\mathtt{f}' \in [\![\widehat{\mathtt{K}}_\mathtt{t}]\!]$. This is what we needed to show to satisfy the "for all", so the definition of the LR gives the result.
   - Case for $\widehat{\mathtt{NAT}}$. The assumptions are exactly the premises of `lr-nat-whr-right`, which then derives the conclusion.
   - Case for $\widehat{\mathtt{EQ_N}}$. The assumptions are exactly the premises of `lr-eqn-whr-right`, which then derives the conclusion.

□

One of the primary difficulties in the completeness proof is that the algorithm is not obviously a congruence on the elim forms; the logical relation is designed to give strong enough assumptions to show that it indeed is. Here, we show that this is the case for the inductive kinds.

LEMMA B.35: LOGICAL RELATION IS A CONGRUENCE FOR ELIMS.

1. If

(a) $\Psi, \mathtt{u} :: \widehat{\mathtt{NAT}} \vdash \mathtt{K} \iff \mathtt{K}' \,\mathtt{kind}$

(b) $\Psi \vdash \mathtt{I} = \mathtt{I}' \in [\![\widehat{\mathtt{NAT}}]\!]$,

(c) $\Psi \vdash \mathtt{C_z} = \mathtt{C_z'} \in [\![(\mathtt{K})^-]\!]$

(d) *for all* $\Psi' \geq \Psi$, $\mathtt{J}$, $\mathtt{J}'$, $\mathtt{R}$, *and* $\mathtt{R}'$ *such that* $\Psi' \vdash \mathtt{J} = \mathtt{J}' \in [\![\widehat{\mathtt{NAT}}]\!]$ *and* $\Psi' \vdash \mathtt{R} = \mathtt{R}' \in [\![(\mathtt{K})^-]\!]$, $\Psi' \vdash [\mathtt{R}/\mathtt{r}][\mathtt{J}/\mathtt{i}']\mathtt{C_s} = [\mathtt{R}'/\mathtt{r}][\mathtt{J}'/\mathtt{i}']\mathtt{C_s}' \in [\![(\mathtt{K})^-]\!]$

then $\Psi \vdash \mathtt{NATrec}[\mathtt{u.K}](\mathtt{I}, \mathtt{C_z}, \mathtt{i'.r.C_s}) = \mathtt{NATrec}[\mathtt{u.K}](\mathtt{I}', \mathtt{C_z'}, \mathtt{i'.r.C_s'}) \in [\![(\mathtt{K})^-]\!]$.

2. *If*

(a) $\Psi, \mathtt{i} :: \widehat{\mathtt{NAT}}, \mathtt{j} :: \widehat{\mathtt{NAT}}, \mathtt{p} :: \widehat{\mathtt{EQ_N}} \vdash \mathtt{K} \Longleftrightarrow \mathtt{K}'\,\mathtt{kind}$

(b) $\Psi \vdash \mathtt{Pf} = \mathtt{Pf}' \in [\![\widehat{\mathtt{EQ_N}}]\!]$

(c) $\Psi \vdash \mathtt{C_{zz}} = \mathtt{C_{zz}'} \in [\![(\mathtt{K})^-]\!]$

(d) *for all* $\Psi' \geq \Psi$, $\mathtt{I}$, $\mathtt{I}'$, $\mathtt{J}$, $\mathtt{J}'$, $\mathtt{P}$, $\mathtt{P}'$, $\mathtt{R}$, *and* $\mathtt{R}'$ *such that* $\Psi' \vdash \mathtt{I} = \mathtt{I}' \in [\![\widehat{\mathtt{NAT}}]\!]$, $\Psi' \vdash \mathtt{J} = \mathtt{J}' \in [\![\widehat{\mathtt{NAT}}]\!]$, $\Psi' \vdash \mathtt{P} = \mathtt{P}' \in [\![\widehat{\mathtt{EQ_N}}]\!]$, *and* $\Psi' \vdash \mathtt{R} = \mathtt{R}' \in [\![(\mathtt{K})^-]\!]$, $\Psi' \vdash [\mathtt{R}/\mathtt{r}][\mathtt{P}/\mathtt{p}][\mathtt{J}/\mathtt{j}][\mathtt{I}/\mathtt{i}]\mathtt{C_{ss}} = [\mathtt{R}'/\mathtt{r}][\mathtt{P}'/\mathtt{p}][\mathtt{J}'/\mathtt{j}][\mathtt{I}'/\mathtt{i}]\mathtt{C_{ss}'} \in [\![(\mathtt{K})^-]\!]$

then $\Psi \vdash \mathtt{EQ_Nrec}[\mathtt{i.j.p.K}](\mathtt{P}, \mathtt{C_{zz}}, \mathtt{i.j.p.r.C_{ss}}) = \mathtt{EQ_Nrec}[\mathtt{i.j.p.K}'](\mathtt{P}', \mathtt{C_{zz}'}, \mathtt{i.j.p.r.C_{ss}'}) \in [\![\widehat{\mathtt{K}}]\!]$.

Proof. 1. This part is proven by induction on the derivation of $\Psi \vdash \mathtt{I} = \mathtt{I}' \in [\![\widehat{\mathtt{NAT}}]\!]$.

- Case for

$$\frac{\mathtt{I} \xrightarrow{\text{whr}} \mathtt{I}'' \quad \Psi \vdash \mathtt{I}'' = \mathtt{I}' \in [\![\widehat{\mathtt{NAT}}]\!]}{\Psi \vdash \mathtt{I} = \mathtt{I}' \in [\![\widehat{\mathtt{NAT}}]\!]}\ \mathtt{lr\text{-}nat\text{-}whr\text{-}left}.$$

By the IH applied to the premise derivation and assumptions (a), (c), and (d), $\Psi \vdash \mathtt{NATrec}[\mathtt{u.K}](\mathtt{I}'', \mathtt{C_z}, \mathtt{i'.r.C_s}) = \mathtt{NATrec}[\mathtt{u.K}](\mathtt{I}', \mathtt{C_z'}, \mathtt{i'.r.C_s'}) \in [\![(\mathtt{K})^-]\!]$. By `whr-natrec-num` applied to the premise head reduction derivation, $\mathtt{NATrec}[\mathtt{u.K}](\mathtt{I}, \mathtt{C_z}, \mathtt{i'.r.C_s}) \xrightarrow{\text{whr}} \mathtt{NATrec}[\mathtt{u.K}](\mathtt{I}'', \mathtt{C_z}, \mathtt{i'.r.C_s})$, so closure under head expansion (LEMMA B.34) gives the result.

- Case for

$$\frac{\mathtt{I}' \xrightarrow{\text{whr}} \mathtt{I}'' \quad \Psi \vdash \mathtt{I} = \mathtt{I}'' \in [\![\widehat{\mathtt{NAT}}]\!]}{\Psi \vdash \mathtt{I} = \mathtt{I}' \in [\![\widehat{\mathtt{NAT}}]\!]}\ \mathtt{lr\text{-}nat\text{-}whr\text{-}right}.$$

By the IH applied to the premise derivation and assumptions (a), (c), and (d), $\Psi \vdash \mathtt{NATrec}[\mathtt{u.K}](\mathtt{I}, \mathtt{C_z}, \mathtt{i'.r.C_s}) = \mathtt{NATrec}[\mathtt{u.K}](\mathtt{I}'', \mathtt{C_z'}, \mathtt{i'.r.C_s'}) \in [\![(\mathtt{K})^-]\!]$. By `whr-natrec-num` applied to the premise head reduction derivation, $\mathtt{NATrec}[\mathtt{u.K}](\mathtt{I}', \mathtt{C_z}, \mathtt{i'.r.C_s}) \xrightarrow{\text{whr}} \mathtt{NATrec}[\mathtt{u.K}](\mathtt{I}'', \mathtt{C_z}, \mathtt{i'.r.C_s})$, so closure under head expansion (LEMMA B.34) gives the result.

- Case for

$$\frac{\Psi \vdash \mathtt{I} \longleftrightarrow \mathtt{I}' :: \widehat{\mathtt{NAT}}}{\Psi \vdash \mathtt{I} = \mathtt{I}' \in [\![\widehat{\mathtt{NAT}}]\!]}\ \mathtt{lr\text{-}nat\text{-}neut\text{-}eq}.$$

First, by LEMMA B.29 applied to premise (c), $\Psi \vdash \mathtt{C_z} \Longleftrightarrow \mathtt{C_z'} :: (\mathtt{K})^-$. Second, observe that the context $\Psi, \mathtt{i}' :: \widehat{\mathtt{NAT}}, \mathtt{r} :: (\mathtt{K})^-$ extends Ψ. By `neut-eq-cn-var`, $\Psi, \mathtt{i}' :: \widehat{\mathtt{NAT}}, \mathtt{r} :: (\mathtt{K})^- \vdash \mathtt{i}' \longleftrightarrow \mathtt{i}' :: \widehat{\mathtt{NAT}}$, so by LEMMA B.29 $\Psi, \mathtt{i}' :: \widehat{\mathtt{NAT}}, \mathtt{r} :: (\mathtt{K})^- \vdash \mathtt{i}' = \mathtt{i}' \in [\![\widehat{\mathtt{NAT}}]\!]$; similarly, $\Psi, \mathtt{i}' :: \widehat{\mathtt{NAT}}, \mathtt{r} :: (\mathtt{K})^- \vdash \mathtt{r} = \mathtt{r} \in [\![(\mathtt{K})^-]\!]$. Thus, by premise (d), $\Psi, \mathtt{i}' :: \widehat{\mathtt{NAT}}, \mathtt{r} :: (\mathtt{K})^- \vdash [\mathtt{r}/\mathtt{r}][\mathtt{i}'/\mathtt{i}']\mathtt{C_s} = [\mathtt{r}/\mathtt{r}][\mathtt{i}'/\mathtt{i}']\mathtt{C_s'} \in [\![(\mathtt{K})^-]\!]$, so again by LEMMA B.29,

$\Psi, \mathtt{i}' :: \widehat{\mathtt{NAT}}, \mathtt{r} :: (\mathtt{K})^- \;\vdash\; \mathtt{C_s} \iff \mathtt{C'_s} :: (\mathtt{K})^-$ (where we have dropped the identity substitutions according to the definition of substitution). Then, by `neut-eq-cn-natrec` applied to premise (a), the premise of the rule, and these two facts,
$\Psi \vdash \mathtt{NATrec[u.K](I, C_z, i'.r.C_s)} \longleftrightarrow \mathtt{NATrec[u.K'](I', C'_z, i'.r.C'_s)} :: (\mathtt{K})^-$. Applying LEMMA B.29 to this gives the result.

- Case for

$$\frac{}{\Psi \vdash \mathtt{z} = \mathtt{z} \in [\![\widehat{\mathtt{NAT}}]\!]}\;\mathtt{lr\text{-}nat\text{-}z}.$$

By premise (c), $\Psi \vdash \mathtt{C_z} = \mathtt{C'_z} \in [\![(\mathtt{K})^-]\!]$. By closure under head expansion (LEMMA B.34) applied twice and `whr-natrec-beta-z`,
$\Psi \vdash \mathtt{NATrec[u.K](z, C_z, i'.r.C_s)} = \mathtt{NATrec[u.K'](z, C'_z, i'.r.C'_s)} \in [\![(\mathtt{K})^-]\!]$.

- Case for

$$\frac{\Psi \vdash \mathtt{I} = \mathtt{I}' \in [\![\widehat{\mathtt{NAT}}]\!]}{\Psi \vdash \mathtt{s\;I} = \mathtt{s\;I}' \in [\![\widehat{\mathtt{NAT}}]\!]}\;\mathtt{lr\text{-}nat\text{-}s}.$$

By the IH applied to premises (a), (c), and (d) and the premise of the rule,
$\Psi \vdash \mathtt{NATrec[u.K](I, C_z, i'.r.C_s)} = \mathtt{NATrec[u.K'](I', C'_z, i'.r.C'_s)} \in [\![(\mathtt{K})^-]\!]$. Thus, we can apply premise (d) to show that
$\Psi \vdash [\mathtt{NATrec[u.K](I, C_z, i'.r.C_s)/r}][\mathtt{I/i'}]\mathtt{C_s} = [\mathtt{NATrec[u.K'](I', C'_z, i'.r.C'_s)/r}][\mathtt{I'/i'}]\mathtt{C'_s} \in [\![(\mathtt{K})^-]\!]$.
We can now apply LEMMA B.34 to `whr-natrec-beta-s` twice to prove that
$\Psi \vdash \mathtt{NATrec[u.K](s\;I, C_z, i'.r.C_s)} = \mathtt{NATrec[u.K'](s\;I', C'_z, i'.r.C'_s)} \in [\![(\mathtt{K})^-]\!]$.

2. This part is proven by induction on the derivation of $\Psi \vdash \mathtt{Pf} = \mathtt{Pf}' \in [\![\widehat{\mathtt{EQ_N}}]\!]$.

- Case for

$$\frac{\mathtt{Pf} \xrightarrow{\text{whr}} \mathtt{Pf}'' \quad \Psi \vdash \mathtt{Pf}'' = \mathtt{Pf}' \in [\![\widehat{\mathtt{NAT}}]\!]}{\Psi \vdash \mathtt{Pf} = \mathtt{Pf}' \in [\![\widehat{\mathtt{NAT}}]\!]}\;\mathtt{lr\text{-}eqn\text{-}whr\text{-}left}.$$

By the IH applied to the premise derivation and assumptions (a), (c), and (d),
$\Psi \vdash \mathtt{EQ_Nrec[i.j.p.K](Pf'', C_{zz}, i.j.p.r.C_{ss})} = \mathtt{EQ_Nrec[i.j.p.K'](Pf', C'_{zz}, i.j.p.r.C'_{ss})} \in [\![(\mathtt{K})^-]\!]$.
By `whr-eqn-rec-proof` applied to the premise head reduction derivation,
$\mathtt{EQ_Nrec[i.j.p.K](Pf, C_{zz}, i.j.p.r.C_{ss})} \xrightarrow{\text{whr}} \mathtt{EQ_Nrec[i.j.p.K](Pf'', C_{zz}, i.j.p.r.C_{ss})}$, so closure under head expansion (LEMMA B.34) gives the result.

- Case for `lr-eqn-whr-right`. This case is analogous to the above case.
- Case for `lr-eqn-neut-eq`. By assumption,

$$\frac{\Psi \vdash \mathtt{Pf} \longleftrightarrow \mathtt{Pf}' :: \widehat{\mathtt{EQ_N}}}{\Psi \vdash \mathtt{Pf} = \mathtt{Pf}' \in [\![\widehat{\mathtt{EQ_N}}]\!]}\;\mathtt{lr\text{-}eqn\text{-}neut\text{-}eq}$$

First, by LEMMA B.29 applied to premise (c), $\Psi \vdash \mathtt{C_{zz}} \iff \mathtt{C'_{zz}} :: (\mathtt{K})^-$. Second, observe that the context $\Psi, \mathtt{i} :: \widehat{\mathtt{NAT}}, \mathtt{j} :: \widehat{\mathtt{NAT}}, \mathtt{p} :: \widehat{\mathtt{EQ_N}}, \mathtt{r} :: (\mathtt{K})^-$ extends Ψ. By `neut-eq-cn-var` and LEMMA B.29, i, j, p, and r are logically related to themselves in this extended context. Thus, by premise (d),
$\Psi, \mathtt{i} :: \widehat{\mathtt{NAT}}, \mathtt{j} :: \widehat{\mathtt{NAT}}, \mathtt{p} :: \widehat{\mathtt{EQ_N}}, \mathtt{r} :: (\mathtt{K})^- \vdash [\mathtt{r/r}][\mathtt{p/p}][\mathtt{j/j}][\mathtt{i/i}]\mathtt{C_{ss}} = [\mathtt{r/r}][\mathtt{p/p}][\mathtt{j/j}][\mathtt{i/i}]\mathtt{C'_{ss}} \in [\![(\mathtt{K})^-]\!]$.
Again by LEMMA B.29, $\Psi, , \mathtt{i} :: \widehat{\mathtt{NAT}}, \mathtt{j} :: \widehat{\mathtt{NAT}}, \mathtt{p} :: \widehat{\mathtt{EQ_N}}, \mathtt{r} :: (\mathtt{K})^- \vdash \mathtt{C_{ss}} \iff \mathtt{C'_{ss}} :: (\mathtt{K})^-$ (where we have dropped the identity substitutions according to the definition of substitution). Then, by `neut-eq-cn-natrec` applied to premise (a), the premise of the rule, and these two facts, we get neutral equality of the $\mathtt{EQ_Nrec}$s, and then LEMMA B.29 gives the result.

- Case for `lr-eqn-zz`

$$\frac{}{\Psi \vdash \mathtt{eqn\_zz} = \mathtt{eqn\_zz} \in [\![\widehat{\mathtt{EQ_N}}]\!]}\ \mathtt{lr\text{-}eqn\text{-}zz}\ .$$

  By premise (c), $\Psi \vdash \mathtt{C_{zz}} = \mathtt{C'_{zz}} \in [\![(\mathtt{K})^-]\!]$. By closure under head expansion (applied twice) (LEMMA B.34) and `whr-eqnrec-beta-zz`, we get the result.

- Case for

$$\frac{\Psi \vdash \mathtt{I} = \mathtt{I'} \in [\![\widehat{\mathtt{NAT}}]\!] \quad \Psi \vdash \mathtt{J} = \mathtt{J'} \in [\![\widehat{\mathtt{NAT}}]\!] \quad \Psi \vdash \mathtt{P} = \mathtt{P'} \in [\![\widehat{\mathtt{EQ_N}}]\!]}{\Psi \vdash \mathtt{eqn\_ss}(\mathtt{I},\mathtt{J},\mathtt{P}) = \mathtt{eqn\_ss}(\mathtt{I'},\mathtt{J'},\mathtt{P'}) \in [\![\widehat{\mathtt{EQ_N}}]\!]}\ \mathtt{lr\text{-}eqn\text{-}ss}\ .$$

  By the IH applied to premises (a), (c), and (d) and the $\widehat{\mathtt{EQ_N}}$ premise of the rule,
  $\Psi \vdash \mathtt{EQ_N rec}[\mathtt{i.j.p.K}](\mathtt{P}, \mathtt{C_{zz}}, \mathtt{i.j.p.r.C_{ss}}) = \mathtt{EQ_N rec}[\mathtt{i.j.p.K'}](\mathtt{P'}, \mathtt{C'_{zz}}, \mathtt{i.j.p.r.C'_{ss}}) \in [\![(\mathtt{K})^-]\!]$.
  Thus, we can apply premise (d) to show logical relatedness of the substitutions into $\mathtt{C_{ss}}$ and $\mathtt{C'_{ss}}$. Then we can apply LEMMA B.34 with `whr-eqnrec-beta-ss` to get the result.

□

LEMMA B.36: DEFINITIONAL EQUALS ARE LOGICALLY RELATED.

1. *If* $\Delta \vdash \mathtt{C} \equiv \mathtt{C'} :: \mathtt{K}$ *and* $\Psi \vdash \sigma = \sigma' \in [\![(\Delta)^-]\!]$ *then* $\Psi \vdash \mathtt{C}[\sigma] = \mathtt{C'}[\sigma'] \in [\![(\mathtt{K})^-]\!]$.
2. *If* $\Delta \vdash \mathtt{K} \equiv \mathtt{K'}\,\mathtt{kind}$ *and* $\Psi \vdash \sigma = \sigma' \in [\![(\Delta)^-]\!]$ *then* $\Psi \vdash \mathtt{K}[\sigma] \Longleftrightarrow \mathtt{K'}[\sigma']\,\mathtt{kind}$.

Proof. By mutual induction on the definitional equality derivations. We sometimes silently apply the definitions of erasure and substitution.[9] Note that this theorem statement meets our invariant about only applying substitutions that substitute for all variables in a constructor: when $\Psi \vdash \sigma = \sigma' \in [\![(\Delta)^-]\!]$, σ and σ' substitute for all variables in Δ.

1. - Case for

$$\frac{\Delta \vdash \mathtt{C_2} \equiv \mathtt{C_1} :: \mathtt{K}}{\Delta \vdash \mathtt{C_1} \equiv \mathtt{C_2} :: \mathtt{K}}\ \mathtt{deq\text{-}cn\text{-}sym}\ .$$

   By symmetry of the logical relations (LEMMA B.32), $\Psi \vdash \sigma' = \sigma \in [\![(\Delta)^-]\!]$. By the IH, $\Psi \vdash \mathtt{C_2}[\sigma'] = \mathtt{C_1}[\sigma] \in [\![(\mathtt{K})^-]\!]$, so by symmetry of the logical relations,
   $\Psi \vdash \mathtt{C_1}[\sigma] = \mathtt{C_2}[\sigma'] \in [\![(\mathtt{K})^-]\!]$.

   - Case for

$$\frac{\Delta \vdash \mathtt{C_1} \equiv \mathtt{C_2} :: \mathtt{K} \quad \Delta \vdash \mathtt{C_2} \equiv \mathtt{C_3} :: \mathtt{K}}{\Delta \vdash \mathtt{C_1} \equiv \mathtt{C_3} :: \mathtt{K}}\ \mathtt{deq\text{-}kd\text{-}trans}\ .$$

   By the IH applied to the first premise, $\Psi \vdash \mathtt{C_1}[\sigma] = \mathtt{C_2}[\sigma'] \in [\![(\mathtt{K})^-]\!]$. By symmetry and transitivity of the LR (LEMMA B.32, LEMMA B.33), $\Psi \vdash \sigma' = \sigma' \in [\![(\Delta)^-]\!]$, so by the IH applied to the second premise, $\Psi \vdash \mathtt{C_2}[\sigma'] = \mathtt{C_3}[\sigma'] \in [\![(\mathtt{K})^-]\!]$. Then transitivity of the logical relations gives the result.

   - Case for

$$\frac{\Delta \vdash \mathtt{C} \equiv \mathtt{C'} :: \mathtt{K} \quad \Delta \vdash \mathtt{K} \equiv \mathtt{K'}\,\mathtt{kind}}{\Delta \vdash \mathtt{C} \equiv \mathtt{C'} :: \mathtt{K'}}\ \mathtt{deq\text{-}cn\text{-}deq\text{-}kd}\ .$$

   By the IH applied to the first premise, $\Psi \vdash \mathtt{C}[\sigma] = \mathtt{C'}[\sigma'] \in [\![(\mathtt{K})^-]\!]$. By LEMMA B.11 applied to the second premise, $(\mathtt{K})^-$ is $(\mathtt{K'})^-$, so replacing syntactic equals gives the result.

[9] Derivations respect the definitions of meta-operations such as substitution and erasure: we are just rewriting their subjects according to the definitions of the meta-operations defining them.

- $$\frac{}{\Delta, \mathtt{u} :: \mathtt{K}, \Delta' \vdash \mathtt{u} \equiv \mathtt{u} :: \mathtt{K}}\ \texttt{deq-cn-var}.$$

  By the definition of erasure, $(\Delta)^-$ contains $\mathtt{u} :: (\mathtt{K})^-$. Thus, by the definition of the logical relations, $\mathtt{C}/\mathtt{u}$ is in σ and $\mathtt{C}'/\mathtt{u}$ is in σ', where $\Psi \vdash \mathtt{C} = \mathtt{C}' \in [\![(\mathtt{K})^-]\!]$. By the definition of substitution, $\mathtt{u}[\sigma]$ is $\mathtt{C}$ and $\mathtt{u}[\sigma']$ is $\mathtt{C}'$, so this is the result.

- Case for

  $$\frac{\overset{\mathcal{D}_1}{\Delta \vdash \mathtt{C}_1 \equiv \mathtt{C}_1' :: \mathtt{TYPE}} \quad \overset{\mathcal{D}_2}{\Delta \vdash \mathtt{C}_2 \equiv \mathtt{C}_2' :: \mathtt{TYPE}}}{\Delta \vdash \mathtt{C}_1 \rightarrow \mathtt{C}_2 \equiv \mathtt{C}_1' \rightarrow \mathtt{C}_2' :: \mathtt{TYPE}}\ \texttt{deq-cn-arrow}.$$

  By the IH applied to each premise derivation, $\Psi \vdash \mathtt{C}_1[\sigma] = \mathtt{C}_1'[\sigma'] \in [\![\widehat{\mathtt{TYPE}}]\!]$ and $\Psi \vdash \mathtt{C}_2[\sigma] = \mathtt{C}_2'[\sigma'] \in [\![\widehat{\mathtt{TYPE}}]\!]$. By LEMMA B.29, $\Psi \vdash \mathtt{C}_1[\sigma] \Longleftrightarrow \mathtt{C}_1'[\sigma'] :: \widehat{\mathtt{TYPE}}$ and $\Psi \vdash \mathtt{C}_2[\sigma] \Longleftrightarrow \mathtt{C}_2'[\sigma'] :: \widehat{\mathtt{TYPE}}$. Then we can apply `norm-eq-cn/type-arrow` to these two derivations to get $\Psi \vdash \mathtt{C}_1[\sigma] \rightarrow \mathtt{C}_2[\sigma] \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{C}_1'[\sigma'] \rightarrow \mathtt{C}_2'[\sigma']$, to which we can apply `norm-eq-cn-type` to get normal equality. Then the definition of substitution lets us pull the substitution outside of the `arrow` on each side, and finally the definition of $[\![\widehat{\mathtt{TYPE}}]\!]$ gives the result.

- Case for `deq-cn-prod`. This case is just like the above, except we use `norm-eq-cn/type-prod`.

- Case for `deq-cn-sum`. This case is just like the above, except we use `norm-eq-cn/type-sum`.

- Case for

  $$\frac{\overset{\mathcal{D}_1}{\Delta \vdash \mathtt{K2} \equiv \mathtt{K2}'\,\mathtt{kind}} \quad \overset{\mathcal{D}_2}{\Delta \vdash \mathtt{C} \equiv \mathtt{C}' :: \Pi_{\mathtt{k}}\, \_::\mathtt{K2}.\, \mathtt{TYPE}}}{\Delta \vdash \forall_{\mathtt{K2}}\, \mathtt{C} \equiv \forall_{\mathtt{K2}'}\, \mathtt{C}' :: \mathtt{TYPE}}\ \texttt{deq-cn-all}.$$

  By IH(2) applied to $\mathcal{D}_1$,
  $\Psi \vdash \mathtt{K}_2[\sigma] \Longleftrightarrow \mathtt{K}_2'[\sigma']\,\mathtt{kind}$. By IH(1) applied to $\mathcal{D}_2$, $\Psi \vdash \mathtt{C}[\sigma] = \mathtt{C}'[\sigma'] \in [\![(\Pi_{\mathtt{k}}\, \_::\mathtt{K2}.\, \mathtt{TYPE})^-]\!]$, so by the definition of erasure and LEMMA B.29, $\Psi \vdash \mathtt{C}[\sigma] \Longleftrightarrow \mathtt{C}'[\sigma'] :: (\mathtt{K2})^- \widehat{\rightarrow}_{\mathtt{k}} \widehat{\mathtt{TYPE}}$. Then we can use `norm-eq-cn/type-all` on these derivations to derive
  $\Psi \vdash \forall_{\mathtt{K}_2[\sigma]}\, \mathtt{C}[\sigma] \Longleftrightarrow_{\mathtt{TYPE}} \forall_{\mathtt{K}_2'[\sigma']}\, \mathtt{C}'[\sigma']$, and `norm-eq-cn-type` to get normal equality on that derivation to get normal equality. The definition of substitution lets us pull the substitution outside on each side, and then the definition of the LR gives the result.

- Case for `deq-cn-exists`. This case is just like the above, except we use `norm-eq-cn-exists`.

- Case for `deq-cn-unit`. By `norm-eq-cn/type-unit`, $\Psi \vdash \mathtt{unit} \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{unit}$, and then `norm-eq-cn-type` gives normal equality. Then the definition of $[\![\widehat{\mathtt{TYPE}}]\!]$ and substitution gives the result.

- Case for `deq-cn-void`. This case is just like the above, except we use `norm-eq-cn/type-void`.

- Case for

  $$\frac{\overset{\mathcal{D}}{\Delta \vdash \mathtt{I} \equiv \mathtt{I}' :: \mathtt{NAT}}}{\Delta \vdash \mathtt{nat}\ \mathtt{I} \equiv \mathtt{nat}\ \mathtt{I}' :: \mathtt{TYPE}}\ \texttt{deq-cn-nat}.$$

  By IH applied to $\mathcal{D}$, $\Psi \vdash \mathtt{I}[\sigma] = \mathtt{I}'[\sigma'] \in [\![\widehat{\mathtt{NAT}}]\!]$, so by LEMMA B.29 these are algorithmically equal. Then we apply `norm-eq-cn/type-nat` and `norm-eq-cn-type` and use the definitions of substitution and $[\![\widehat{\mathtt{TYPE}}]\!]$ to get the result.

- Case for `deq-cn-list`. This case is just like the above, except we use `norm-eq-cn/type-list`.

- Case for
$$\frac{\Delta \vdash \mathtt{K}_2 \equiv \mathtt{K}_2' \,\mathtt{kind} \quad \Delta, \mathtt{u} :: \mathtt{K}_2 \vdash \mathtt{C} \equiv \mathtt{C}' :: \mathtt{K}}{\Delta \vdash \lambda_{\mathtt{c}}\, \mathtt{u}{::}\mathtt{K}_2.\, \mathtt{C} \equiv \lambda_{\mathtt{c}}\, \mathtt{u}{::}\mathtt{K}_2'.\, \mathtt{C}' :: \Pi_{\mathtt{k}}\, \mathtt{u}{::}\mathtt{K}_2.\, \mathtt{K}}\ \texttt{deq-cn-fn}\ .$$
We are going to use the definition of the LR for $(\Pi_{\mathtt{k}}\, \mathtt{u}{::}\mathtt{K}_2.\, \mathtt{K})^- = (\mathtt{K}_2)^- \widehat{\rightarrow}_{\mathtt{k}} (\mathtt{K})^-$, so assume for the "for all" that for arbitrary $\mathtt{C}_2$, $\mathtt{C}_2'$, and $\Psi_+ \geq \Psi$, $\Psi_+ \vdash \mathtt{C}_2 = \mathtt{C}_2' \in [\![(\mathtt{K}_2)^-]\!]$. By LEMMA B.31, $\Psi_+ \vdash \sigma = \sigma' \in [\![(\Delta)^-]\!]$, so by the definition of the LR, $\Psi_+ \vdash \sigma, \mathtt{C}_2/\mathtt{u} = \sigma', \mathtt{C}_2'/\mathtt{u} \in [\![(\Delta)^-, \mathtt{u} :: (\mathtt{K}_2)^-]\!]$. By the definition of erasure, this context is $(\Delta, \mathtt{u} :: \mathtt{K}_2)^-$. Thus, by the IH, $\Psi_+ \vdash \mathtt{C}[\sigma, \mathtt{C}_2/\mathtt{u}] = \mathtt{C}'[\sigma', \mathtt{C}_2'/\mathtt{u}] \in [\![(\mathtt{K})^-]\!]$. By LEMMA B.26, $\mathtt{C}[\sigma, \mathtt{C}_2/\mathtt{u}]$ is $[\mathtt{C}_2/\mathtt{u}]\mathtt{C}[\sigma, \mathtt{u}/\mathtt{u}]$ and $\mathtt{C}'[\sigma, \mathtt{C}_2'/\mathtt{u}]$ is $[\mathtt{C}_2'/\mathtt{u}]\mathtt{C}'[\sigma', \mathtt{u}/\mathtt{u}]$. By `whr-app-beta`, $(\lambda_{\mathtt{c}}\, \mathtt{u}{::}\mathtt{K}_2[\sigma].\, \mathtt{C}[\sigma, \mathtt{u}/\mathtt{u}])\ \mathtt{C}_2 \xrightarrow{\text{whr}} [\mathtt{C}_2/\mathtt{u}](\mathtt{C}[\sigma, \mathtt{u}/\mathtt{u}])$ and $(\lambda_{\mathtt{c}}\, \mathtt{u}{::}\mathtt{K}_2'[\sigma'].\, \mathtt{C}'[\sigma', \mathtt{u}/\mathtt{u}])\ \mathtt{C}_2' \xrightarrow{\text{whr}} [\mathtt{C}_2'/\mathtt{u}](\mathtt{C}'[\sigma', \mathtt{u}/\mathtt{u}])$. Thus, by closure under head expansion (LEMMA B.34) applied once to each side, $\Psi_+ \vdash (\lambda_{\mathtt{c}}\, \mathtt{u}{::}\mathtt{K}_2[\sigma].\, \mathtt{C}[\sigma, \mathtt{u}/\mathtt{u}])\ \mathtt{C}_2 = (\lambda_{\mathtt{c}}\, \mathtt{u}{::}\mathtt{K}_2'[\sigma'].\, \mathtt{C}'[\sigma', \mathtt{u}/\mathtt{u}])\ \mathtt{C}_2' \in [\![(\mathtt{K})^-]\!]$. We can pull the substitution outside the λ on each side by the definition of substitution; then the definition of the LR for $(\mathtt{K}_2)^- \widehat{\rightarrow}_{\mathtt{k}} (\mathtt{K})^-$ gives the result.

- Case for
$$\frac{\Delta \vdash \mathtt{C}_1 \equiv \mathtt{C}_1' :: \Pi_{\mathtt{k}}\, \mathtt{u}{::}\mathtt{K}_2.\, \mathtt{K} \quad \Delta \vdash \mathtt{C}_2 \equiv \mathtt{C}_2' :: \mathtt{K}_2}{\Delta \vdash \mathtt{C}_1\ \mathtt{C}_2 \equiv \mathtt{C}_1'\ \mathtt{C}_2' :: [\mathtt{C}_2/\mathtt{u}]\mathtt{K}}\ \texttt{deq-cn-app}\ .$$
By the IH, $\Psi \vdash \mathtt{C}_1[\sigma] = \mathtt{C}_1'[\sigma'] \in [\![(\Pi_{\mathtt{k}}\, \mathtt{u}{::}\mathtt{K}_2.\, \mathtt{K})^-]\!]$ and $\Psi \vdash \mathtt{C}_2[\sigma] = \mathtt{C}_2'[\sigma'] \in [\![(\mathtt{K}_2)^-]\!]$. By the definition of erasure, $(\Pi_{\mathtt{k}}\, \mathtt{u}{::}\mathtt{K}_2.\, \mathtt{K})^-$ is $(\mathtt{K}_2)^- \widehat{\rightarrow}_{\mathtt{k}} (\mathtt{K})^-$. Thus, by the definition of the LR, $\Psi \vdash \mathtt{C}_1[\sigma]\ \mathtt{C}_2[\sigma] = \mathtt{C}_1'[\sigma']\ \mathtt{C}_2'[\sigma'] \in [\![(\mathtt{K})^-]\!]$. Then, by the definition of substitution, we can pull the substitution outside the application on both sides, and, by LEMMA B.11, $(\mathtt{K})^-$ is $([\mathtt{C}_2/\mathtt{u}]\mathtt{K})^-$, so this is the result.

- Case for
$$\frac{\Delta, \mathtt{u} :: \mathtt{K}_2 \vdash \mathtt{C}_1 \equiv \mathtt{C}_1' :: \mathtt{K} \quad \Delta \vdash \mathtt{C}_2 \equiv \mathtt{C}_2' :: \mathtt{K}_2}{\Delta \vdash (\lambda_{\mathtt{c}}\, \mathtt{u}{::}\mathtt{K}_2.\, \mathtt{C}_1)\ \mathtt{C}_2 \equiv [\mathtt{C}_2'/\mathtt{u}]\mathtt{C}_1' :: [\mathtt{C}_2/\mathtt{u}]\mathtt{K}}\ \texttt{deq-cn-app-beta}\ .$$
By induction, $\Psi \vdash \mathtt{C}_2[\sigma] = \mathtt{C}_2'[\sigma'] \in [\![(\mathtt{K}_2)^-]\!]$. By the definition of the LR, $\Psi \vdash \sigma, \mathtt{C}_2/\mathtt{u} = \sigma', \mathtt{C}_2'/\mathtt{u} \in [\![(\Delta)^-, \mathtt{u} :: (\mathtt{K}_2)^-]\!]$, and by the definition of erasure, this context is $(\Delta, \mathtt{u} :: \mathtt{K}_2)^-$. Thus, by the IH, $\Psi \vdash \mathtt{C}_1[\sigma, \mathtt{C}_2[\sigma]/\mathtt{u}] = \mathtt{C}_1'[\sigma', \mathtt{C}_2'[\sigma']/\mathtt{u}] \in [\![(\mathtt{K})^-]\!]$. By LEMMA B.26, $\mathtt{C}_1[\sigma, \mathtt{C}_2[\sigma]/\mathtt{u}]$ is $[\mathtt{C}_2[\sigma]/\mathtt{u}]\mathtt{C}_1[\sigma, \mathtt{u}/\mathtt{u}]$. By `whr-app-beta`, $(\lambda_{\mathtt{c}}\, \mathtt{u}{::}\mathtt{K}_2[\sigma].\, \mathtt{C}_1[\sigma, \mathtt{u}/\mathtt{u}])\ \mathtt{C}_2[\sigma] \xrightarrow{\text{whr}} [\mathtt{C}_2[\sigma]/\mathtt{u}](\mathtt{C}_1[\sigma, \mathtt{u}/\mathtt{u}])$, so by closure under head expansion (LEMMA B.34), $\Psi \vdash (\lambda_{\mathtt{c}}\, \mathtt{u}{::}\mathtt{K}_2[\sigma].\, \mathtt{C}_1[\sigma, \mathtt{u}/\mathtt{u}])\ \mathtt{C}_2[\sigma] = \mathtt{C}_1'[\sigma', \mathtt{C}_2'[\sigma']/\mathtt{u}] \in [\![(\mathtt{K})^-]\!]$. On the left, the definition of substitution allows us to pull the substitution outside the λ and then the application, giving $\Psi \vdash ((\lambda_{\mathtt{c}}\, \mathtt{u}{::}\mathtt{K}_2.\, \mathtt{C}_1)\ \mathtt{C}_2)[\sigma] = \mathtt{C}_1'[\sigma', \mathtt{C}_2'[\sigma']/\mathtt{u}] \in [\![(\mathtt{K})^-]\!]$. Then, by LEMMA B.26, we can rewrite the right-hand side as $([\mathtt{C}_2'/\mathtt{u}]\mathtt{C}_1')[\sigma']$. Finally, LEMMA B.11 shows that $([\mathtt{C}_2/\mathtt{u}]\mathtt{K})^-$ is $(\mathtt{K})^-$, so we have the result.

- Case for `deq-cn-fn-ext`:
$$\frac{\Delta \vdash \mathtt{K}_2\, \mathtt{kind} \quad \Delta \vdash \mathtt{C} :: \Pi_{\mathtt{k}}\, \mathtt{u}{::}\mathtt{K}_2.\, \mathtt{K} \quad \Delta \vdash \mathtt{C}' :: \Pi_{\mathtt{k}}\, \mathtt{u}{::}\mathtt{K}_2.\, \mathtt{K} \quad \Delta, \mathtt{u} :: \mathtt{K}_2 \vdash \mathtt{C}\ \mathtt{u} \equiv \mathtt{C}'\ \mathtt{u} :: \mathtt{K}}{\Delta \vdash \mathtt{C} \equiv \mathtt{C}' :: \Pi_{\mathtt{k}}\, \mathtt{u}{::}\mathtt{K}_2.\, \mathtt{K}}\ .$$
We are going to use the definition of the LR for $(\Pi_{\mathtt{k}}\, \mathtt{u}{::}\mathtt{K}_2.\, \mathtt{K})^- = (\mathtt{K}_2)^- \widehat{\rightarrow}_{\mathtt{k}} (\mathtt{K})^-$, so assume for the "for all" that for arbitrary $\mathtt{C}_2$, $\mathtt{C}_2'$, and $\Psi_+ \geq \Psi$, $\Psi_+ \vdash \mathtt{C}_2 = \mathtt{C}_2' \in [\![(\mathtt{K}_2)^-]\!]$. By LEMMA B.31, $\Psi_+ \vdash \sigma = \sigma' \in [\![(\Delta)^-]\!]$, so by the definition of the LR, $\Psi_+ \vdash \sigma, \mathtt{C}_2/\mathtt{u} = \sigma', \mathtt{C}_2'/\mathtt{u} \in [\![(\Delta)^-, \mathtt{u} :: (\mathtt{K}_2)^-]\!]$. By the definition of erasure, this context is

$(\Delta, \mathtt{u} :: \mathtt{K}_2)^-$. Then, by induction, $\Psi_+ \vdash (\mathtt{C}\,\mathtt{u})[\sigma, \mathtt{C}_2/\mathtt{u}] = (\mathtt{C}'\,\mathtt{u})[\sigma', \mathtt{C}'_2/\mathtt{u}] \in [\![(\mathtt{K})^-]\!]$. The bound variable $\mathtt{u}$ is chosen fresh and it is not free in $\mathtt{C}$ or $\mathtt{C}'$; consequently, rewriting using the definition of substitution gives that $\Psi_+ \vdash \mathtt{C}[\sigma]\,\mathtt{C}_2 = \mathtt{C}'[\sigma']\,\mathtt{C}'_2 \in [\![(\mathtt{K})^-]\!]$. Then the definition of the LR for $(\mathtt{K}_2)^- \widehat{\rightarrow}_{\mathtt{k}} (\mathtt{K})^-$ gives the result.

- Case for `deq-cn-z`. Note that $\mathtt{z}[\sigma]$ is just $\mathtt{z}$, so `lr-nat-z` gives the result.
- Case for

$$\frac{\overset{\mathcal{D}}{\Delta \vdash \mathtt{I} \equiv \mathtt{I}' :: \mathtt{NAT}}}{\Delta \vdash \mathtt{s}\,\mathtt{I} \equiv \mathtt{s}\,\mathtt{I}' :: \mathtt{NAT}}\ \texttt{deq-cn-s}\ .$$

By induction, $\Psi \vdash \mathtt{I}[\sigma] = \mathtt{I}'[\sigma'] \in [\![\widehat{\mathtt{NAT}}]\!]$. Thus, we can apply `lr-nat-s` and then use the definition of substitution to move the substitutions outside the $\mathtt{s}$ on both sides, which gives the result.

- Case for

$$\frac{\begin{array}{c}\mathcal{D}_1\\ \Delta, \mathtt{i} :: \mathtt{N} \vdash \mathtt{K} \equiv \mathtt{K}'\,\mathtt{kind}\\ \mathcal{D}_2\\ \Delta \vdash \mathtt{I} \equiv \mathtt{I}' :: \mathtt{NAT}\\ \mathcal{D}_3\\ \Delta \vdash \mathtt{C}_\mathtt{z} \equiv \mathtt{C}'_\mathtt{z} :: [\mathtt{z}/\mathtt{i}]\mathtt{K}\\ \mathcal{D}_4\\ \Delta, \mathtt{i}' :: \mathtt{N}, \mathtt{r} :: [\mathtt{i}'/\mathtt{i}]\mathtt{K} \vdash \mathtt{C}_\mathtt{s} \equiv \mathtt{C}'_\mathtt{s} :: [\mathtt{s}\,\mathtt{I}'/\mathtt{i}]\mathtt{K}\end{array}}{\Delta \vdash \mathtt{NATrec}[\mathtt{i}.\mathtt{K}](\mathtt{I}, \mathtt{C}_1, \mathtt{i}'.\mathtt{r}.\mathtt{C}_2) \equiv \mathtt{NATrec}[\mathtt{i}.\mathtt{K}'](\mathtt{I}', \mathtt{C}'_1, \mathtt{i}'.\mathtt{r}.\mathtt{C}'_2) :: [\mathtt{I}/\mathtt{i}]\mathtt{K}}\ \texttt{deq-cn-natrec}\ .$$

Note that by LEMMA B.11, the erasure of any substitution into $\mathtt{K}$ is still $(\mathtt{K})^-$ without the substitution; we use this fact silently below.

We are going to use LEMMA B.35, so we must satisfy its assumptions.

(a) By LEMMA B.29, $\Psi, \mathtt{u} :: \widehat{\mathtt{NAT}} \vdash \mathtt{u} = \mathtt{u} \in [\![\widehat{\mathtt{NAT}}]\!]$; thus by LEMMA B.30 and the definition of the LR $\Psi, \mathtt{u} :: \widehat{\mathtt{K}} \vdash \sigma, \mathtt{u}/\mathtt{u} = \sigma', \mathtt{u}/\mathtt{u} \in [\![(\Delta)^-, \mathtt{u} :: \widehat{\mathtt{K}}]\!]$. By the IH(2) applied to $\mathcal{D}_1$, $\Psi, \mathtt{u} :: \widehat{\mathtt{NAT}} \vdash \mathtt{K}[\sigma, \mathtt{u}/\mathtt{u}] \iff \mathtt{K}'[\sigma', \mathtt{u}/\mathtt{u}]\,\mathtt{kind}$.

(b) By the IH(1) applied to $\mathcal{D}_2$, $\Psi \vdash \mathtt{I}[\sigma] = \mathtt{I}'[\sigma'] \in [\![\widehat{\mathtt{NAT}}]\!]$

(c) By the IH(1) applied to $\mathcal{D}_3$, $\Psi \vdash \mathtt{C}_\mathtt{z}[\sigma] = \mathtt{C}'_\mathtt{z}[\sigma'] \in [\![\widehat{\mathtt{K}}]\!]$.

(d) Assume for the "for all" arbitrary $\Psi_+ \geq \Psi$ and $\mathtt{J}$, $\mathtt{J}'$, $\mathtt{R}$, and $\mathtt{R}'$ such that
$\Psi_+ \vdash \mathtt{J} = \mathtt{J}' \in [\![\widehat{\mathtt{NAT}}]\!]$ and $\Psi_+ \vdash \mathtt{R} = \mathtt{R}' \in [\![(\mathtt{K})^-]\!]$. By closure of the LR under context extension (LEMMA B.31), $\Psi_+ \vdash \sigma = \sigma' \in [\![(\Delta)^-]\!]$. Applying the definition of logically related substitutions once gives $\Psi_+ \vdash \sigma, \mathtt{J}/\mathtt{i}' = \sigma', \mathtt{J}/\mathtt{i}' \in [\![\Delta, \mathtt{i}' :: \widehat{\mathtt{NAT}}]\!]$, and applying it gives $\Psi_+ \vdash \sigma, \mathtt{J}/\mathtt{i}', \mathtt{R}/\mathtt{r} = \sigma', \mathtt{J}/\mathtt{i}', \mathtt{R}'/\mathtt{r} \in [\![\Delta, \mathtt{i}' :: \widehat{\mathtt{NAT}}, \mathtt{r} :: (\mathtt{K})^-]\!]$. We can then apply the IH to $\mathcal{D}_4$ and these substitutions to get that
$\Psi_+ \vdash \mathtt{C}_\mathtt{s}[\sigma, \mathtt{J}/\mathtt{i}', \mathtt{R}/\mathtt{r}] = \mathtt{C}'_\mathtt{s}[\sigma', \mathtt{J}'/\mathtt{i}', \mathtt{R}'/\mathtt{r}] \in [\![(\mathtt{K})^-]\!]$. Then LEMMA B.26 gives that $\mathtt{C}_\mathtt{s}[\sigma, \mathtt{J}/\mathtt{i}', \mathtt{R}/\mathtt{r}]$ is $[\mathtt{R}/\mathtt{r}][\mathtt{J}/\mathtt{i}'](\mathtt{C}_\mathtt{s}[\sigma, \mathtt{i}'/\mathtt{i}', \mathtt{r}/\mathtt{r}])$ and the analogous statement for the right-hand side.

Now we can use the fact that the LR is a congruence for the elimination forms (LEMMA B.35) on these facts to show that
$\Psi \vdash \mathtt{NATrec}[\mathtt{u}.\mathtt{K}[\sigma, \mathtt{u}/\mathtt{u}]](\mathtt{I}[\sigma], \mathtt{C}_\mathtt{z}[\sigma], \mathtt{i}'.\mathtt{r}.\mathtt{C}_\mathtt{s}[\sigma, \mathtt{i}'/\mathtt{i}', \mathtt{r}/\mathtt{r}]) =$
$\mathtt{NATrec}[\mathtt{u}.\mathtt{K}'[\sigma', \mathtt{u}/\mathtt{u}]](\mathtt{I}'[\sigma'], \mathtt{C}'_\mathtt{z}[\sigma'], \mathtt{i}'.\mathtt{r}.\mathtt{C}'_\mathtt{s}[\sigma', \mathtt{i}'/\mathtt{i}', \mathtt{r}/\mathtt{r}]) \in [\![(\mathtt{K})^-]\!]$. Then by the definition of substitution we can pull the substitutions outside the `NATrec` on both sides, and we are done.

- Case for `deq-cn-natrec-beta-z`:

$$\frac{\Delta, \mathtt{u} :: \mathtt{N} \vdash \mathtt{K}\,\mathtt{kind} \quad \Delta \vdash \mathtt{C_z} \equiv \mathtt{C'_z} :: [\mathtt{z}/\mathtt{i}]\mathtt{K} \quad \Delta, \mathtt{i'} :: \mathtt{N}, \mathtt{r} :: [\mathtt{i'}/\mathtt{i}]\mathtt{K} \vdash \mathtt{C_s} :: [\mathtt{s}\ \mathtt{I'}/\mathtt{i}]\mathtt{K}}{\Delta \vdash \mathtt{NATrec}[\mathtt{u}.\mathtt{K}](\mathtt{z}, \mathtt{C_z}, \mathtt{i'}.\mathtt{r}.\mathtt{C_s}) \equiv \mathtt{C'_z} :: [\mathtt{z}/\mathtt{i}]\mathtt{K}}.$$

  By the IH, $\Psi \vdash \mathtt{C_z}[\sigma] = \mathtt{C'_z}[\sigma'] \in [\![(\mathtt{K})^-]\!]$. By `whr-natrec-beta-z`, $\mathtt{NATrec}[\mathtt{u}.\mathtt{K}[\sigma, \mathtt{u}/\mathtt{u}]](\mathtt{z}, \mathtt{C_z}[\sigma], \mathtt{i'}.\mathtt{r}.\mathtt{C_s}[\sigma, \mathtt{i'}/\mathtt{i'}, \mathtt{r}/\mathtt{r}]) \xrightarrow{\text{whr}} \mathtt{C_z}[\sigma]$. Then by LEMMA B.34 on the left side, $\Psi \vdash \mathtt{NATrec}[\mathtt{u}.\mathtt{K}[\sigma, \mathtt{u}/\mathtt{u}]](\mathtt{z}, \mathtt{C_z}[\sigma], \mathtt{i'}.\mathtt{r}.\mathtt{C_s}[\sigma, \mathtt{i'}/\mathtt{i'}, \mathtt{r}/\mathtt{r}]) = \mathtt{C'_z}[\sigma'] \in [\![(\mathtt{K})^-]\!]$. By the definition of substitution, `z` is the same as $\mathtt{z}[\sigma]$ and we can pull the substitution outside the `NATrec` to get the result.

- Case for `deq-cn-natrec-beta-s`:

$$\frac{\begin{array}{c}\Delta, \mathtt{u} :: \mathtt{N} \vdash \mathtt{K} \equiv \mathtt{K'}\,\mathtt{kind} \\ \Delta \vdash \mathtt{I} \equiv \mathtt{I'} :: \mathtt{NAT} \\ \Delta \vdash \mathtt{C_z} \equiv \mathtt{C'_z} :: [\mathtt{z}/\mathtt{u}]\mathtt{K} \\ \Delta, \mathtt{i'} :: \mathtt{N}, \mathtt{r} :: [\mathtt{i'}/\mathtt{u}]\mathtt{K} \vdash \mathtt{C_s} \equiv \mathtt{C'_s} :: [\mathtt{s}\ \mathtt{I'}/\mathtt{u}]\mathtt{K}\end{array}}{\Delta \vdash \mathtt{NATrec}[\mathtt{u}.\mathtt{K}](\mathtt{s}\ \mathtt{I}, \mathtt{C_z}, \mathtt{i'}.\mathtt{r}.\mathtt{C_s}) \equiv [\mathtt{NATrec}[\mathtt{u}.\mathtt{K'}](\mathtt{I'}, \mathtt{C'_z}, \mathtt{i'}.\mathtt{r}.\mathtt{C'_s})/\mathtt{r}][\mathtt{I'}/\mathtt{i'}]\mathtt{C'_s} :: [\mathtt{s}\ \mathtt{I}/\mathtt{u}]\mathtt{K}.}$$

  Note that the premises are the same as those of `deq-cn-natrec`, so by the same reasoning as in the first paragraph of that case, we can use the IH to satisfy all of the premises of LEMMA B.35, and then applying the lemma gives
  $\Psi \vdash \mathtt{NATrec}[\mathtt{u}.\mathtt{K}[\sigma, \mathtt{u}/\mathtt{u}]](\mathtt{I}[\sigma], \mathtt{C_z}[\sigma], \mathtt{i'}.\mathtt{r}.\mathtt{C_s}[\sigma, \mathtt{i'}/\mathtt{i'}, \mathtt{r}/\mathtt{r}]) =$
  $\mathtt{NATrec}[\mathtt{u}.\mathtt{K'}[\sigma', \mathtt{u}/\mathtt{u}]](\mathtt{I'}[\sigma'], \mathtt{C'_z}[\sigma'], \mathtt{i'}.\mathtt{r}.\mathtt{C'_s}[\sigma', \mathtt{i'}/\mathtt{i'}, \mathtt{r}/\mathtt{r}]) \in [\![(\mathtt{K})^-]\!]$. Call the left-hand constructor `R` and the right-hand one $\mathtt{R'}$. Then by the definition of the LR applied twice,
  $\Psi \vdash \sigma, \mathtt{I}[\sigma]/\mathtt{i'}, \mathtt{R}/\mathtt{r} = \sigma, \mathtt{I'}[\sigma']/\mathtt{i'}, \mathtt{R'}/\mathtt{r} \in [\![(\Delta)^-, \mathtt{i'} :: \widehat{\mathtt{NAT}}, \mathtt{r} :: (\mathtt{K})^-]\!]$. By the definition of erasure, this matches the context in the final premise, so by the IH
  $\Psi \vdash \mathtt{C_s}[\sigma, \mathtt{I}[\sigma]/\mathtt{i'}, \mathtt{R}/\mathtt{r}] = \mathtt{C_s}[\sigma', \mathtt{I'}[\sigma']/\mathtt{i'}, \mathtt{R}/\mathtt{r}] \in [\![(\mathtt{K})^-]\!]$. By `whr-natrec-beta-s`, $\mathtt{NATrec}[\mathtt{u}.\mathtt{K}[\sigma, \mathtt{u}/\mathtt{u}]](\mathtt{s}\,(\mathtt{I}[\sigma]), \mathtt{C_z}[\sigma], \mathtt{i'}.\mathtt{r}.\mathtt{C_s}[\sigma, \mathtt{i'}/\mathtt{i'}, \mathtt{r}/\mathtt{r}]) \xrightarrow{\text{whr}} [\mathtt{R}/\mathtt{r}][\mathtt{I}[\sigma]/\mathtt{i'}](\mathtt{C_s}[\sigma, \mathtt{i'}/\mathtt{i'}, \mathtt{r}/\mathtt{r}])$, so by closure under head expansion (LEMMA B.34) and LEMMA B.26,
  $\Psi \vdash \mathtt{NATrec}[\mathtt{u}.\mathtt{K}[\sigma, \mathtt{u}/\mathtt{u}]](\mathtt{s}\,(\mathtt{I}[\sigma]), \mathtt{C_z}[\sigma], \mathtt{i'}.\mathtt{r}.\mathtt{C_s}[\sigma, \mathtt{i'}/\mathtt{i'}, \mathtt{r}/\mathtt{r}]) = \mathtt{C_s}[\sigma', \mathtt{I'}[\sigma']/\mathtt{i'}, \mathtt{R'}/\mathtt{r}] \in [\![(\mathtt{K})^-]\!]$.
  The definition of substitution gives that the left-hand side is $\mathtt{NATrec}[\mathtt{u}.\mathtt{K}](\mathtt{s}\ \mathtt{I}, \mathtt{C_z}, \mathtt{i'}.\mathtt{r}.\mathtt{C_s})[\sigma]$ and that $\mathtt{R'}$ is $\mathtt{NATrec}[\mathtt{u}.\mathtt{K'}](\mathtt{s}\ \mathtt{I'}, \mathtt{C'_z}, \mathtt{i'}.\mathtt{r}.\mathtt{C'_s})[\sigma']$ Finally, by LEMMA B.26, the right-hand side is $([\mathtt{NATrec}[\mathtt{u}.\mathtt{K'}](\mathtt{I'}, \mathtt{C'_z}, \mathtt{i'}.\mathtt{r}.\mathtt{C'_s})/\mathtt{r}][\mathtt{I'}/\mathtt{i'}]\mathtt{C_s})[\sigma']$.

- Case for `deq-cn-eqn-zz`. $\mathtt{eqn\_zz}[\sigma]$ is just `eqn_zz`, so `lr-eqn-zz` gives the result.

- Case for `deq-cn-eqn-ss`. By the IH,

$$\begin{array}{c}\Psi \vdash \mathtt{I}[\sigma] = \mathtt{I'}[\sigma'] \in [\![\widehat{\mathtt{NAT}}]\!] \\ \Psi \vdash \mathtt{J}[\sigma] = \mathtt{J'}[\sigma'] \in [\![\widehat{\mathtt{NAT}}]\!] \\ \Psi \vdash \mathtt{Pf}[\sigma] = \mathtt{Pf'}[\sigma'] \in [\![\widehat{\mathtt{EQ_N}}]\!].\end{array}$$

  Thus `lr-eqn-ss` and the definition of substitution give the result.

- Case for `deq-cn-eqn-rec`.
  We are going to use LEMMA B.35, so we must satisfy its premises.
  (a) By LEMMA B.29, LEMMA B.30 and the definition of the LR,
  $\Psi, \mathtt{i} :: \widehat{\mathtt{NAT}}, \mathtt{j} :: \widehat{\mathtt{NAT}}, \mathtt{p} :: \widehat{\mathtt{EQ_N}} \vdash \sigma, \mathtt{i}/\mathtt{i}, \mathtt{j}/\mathtt{j}, \mathtt{p}/\mathtt{p} = \sigma', \mathtt{i}/\mathtt{i}, \mathtt{j}/\mathtt{j}, \mathtt{p}/\mathtt{p} \in [\![(\Delta)^-, \mathtt{i} :: \widehat{\mathtt{NAT}}, \mathtt{j} :: \widehat{\mathtt{NAT}}, \mathtt{p} :: \widehat{\mathtt{EQ_N}}]\!]$.
  Thus, the IH gives that
  $\Psi, \mathtt{i} :: \widehat{\mathtt{NAT}}, \mathtt{j} :: \widehat{\mathtt{NAT}}, \mathtt{p} :: \widehat{\mathtt{EQ_N}} \vdash \mathtt{K}[\sigma, \mathtt{i}/\mathtt{i}, \mathtt{j}/\mathtt{j}, \mathtt{p}/\mathtt{p}] \iff \mathtt{K'}[\sigma', \mathtt{i}/\mathtt{i}, \mathtt{j}/\mathtt{j}, \mathtt{p}/\mathtt{p}]\,\mathtt{kind}$
  (b) By the IH applied to the premise of the rule, $\Psi \vdash \mathtt{Pf}[\sigma] = \mathtt{Pf'}[\sigma'] \in [\![\widehat{\mathtt{EQ_N}}]\!]$.

(c) By the IH applied to the premise of the rule, $\Psi \vdash \mathtt{C}_{\mathtt{zz}}[\sigma] = \mathtt{C}'_{\mathtt{zz}}[\sigma'] \in [\![(\mathtt{K})^-]\!]$.

(d) Assume for the "for all" $\Psi' \geq \Psi$, $\mathtt{I}$, $\mathtt{I}'$, $\mathtt{J}$, $\mathtt{J}'$, $\mathtt{P}$, $\mathtt{P}'$, $\mathtt{R}$, and $\mathtt{R}'$ such that $\Psi' \vdash \mathtt{I} = \mathtt{I}' \in [\![\widehat{\mathtt{NAT}}]\!]$, $\Psi' \vdash \mathtt{J} = \mathtt{J}' \in [\![\widehat{\mathtt{NAT}}]\!]$, $\Psi' \vdash \mathtt{P} = \mathtt{P}' \in [\![\widehat{\mathtt{EQ_N}}]\!]$, and $\Psi' \vdash \mathtt{R} = \mathtt{R}' \in [\![(\mathtt{K})^-]\!]$. By closure under context extension, $\Psi' \vdash \sigma = \sigma' \in [\![(\Delta)^-]\!]$. Then, by the definition of the LR for substitutions, $\Psi \vdash \sigma, \mathtt{I/i}, \mathtt{J/j}, \mathtt{P/p}, \mathtt{R/r} = \sigma', \mathtt{I'/i}, \mathtt{J'/j}, \mathtt{P'/p}, \mathtt{R'/r} \in [\![(\Delta)^-, \mathtt{i} :: \widehat{\mathtt{NAT}}, \mathtt{j} :: \widehat{\mathtt{NAT}}, \mathtt{p} :: \widehat{\mathtt{EQ_N}}, \mathtt{r} :: (\mathtt{K})^-]\!]$. Because this context matches the erasure of the context in the fourth premise of the rule, we can apply the IH to get $\Psi_+ \vdash \mathtt{C}_{\mathtt{ss}}[\sigma, \mathtt{I/i}, \mathtt{J/j}, \mathtt{P/p}, \mathtt{R/r}] = \mathtt{C}'_{\mathtt{ss}}[\sigma', \mathtt{I'/i}, \mathtt{J'/j}, \mathtt{P'/p}, \mathtt{R'/r}] \in [\![(\mathtt{K})^-]\!]$. Then LEMMA B.26 shows that $\mathtt{C}_{\mathtt{ss}}[\sigma, \mathtt{I/i}, \mathtt{J/j}, \mathtt{P/p}, \mathtt{R/r}]$ is $[\mathtt{R/r}][\mathtt{P/p}][\mathtt{J/j}][\mathtt{I/i}](\mathtt{C}_{\mathtt{ss}}[\sigma, \mathtt{i/i}, \mathtt{j/j}, \mathtt{p/p}, \mathtt{r/r}])$ and the analogous fact for the right-hand side. Applying these equalities proves the result.

Then the lemma and the definition of substitution give the result.

- Case for `deq-cn-eqn-rec-beta-zz`. By the IH, $\Psi \vdash \mathtt{C}_{\mathtt{zz}}[\sigma] = \mathtt{C}'_{\mathtt{zz}}[\sigma'] \in [\![(\mathtt{K})^-]\!]$. Then closure under head expansion (LEMMA B.34) with `whr-eqn-rec-beta-zz` on the left-hand side and the definition of substitution give the result.

- Case for `deq-cn-eqn-rec-beta-ss`.

  Observe that the premises here contain all the premises of the congruence rule, so we can satisfy the assumptions of LEMMA B.35 in the same way. This gives that $\Psi \vdash \mathtt{EQ_N rec}[\mathtt{i.j.p.K}[\sigma, \mathtt{i/i}, \mathtt{j/j}, \mathtt{p/p}]](\mathtt{P}[\sigma], \mathtt{C}_{\mathtt{zz}}[\sigma], \mathtt{i.j.p.r.C}_{\mathtt{ss}}[\sigma, \mathtt{i/i}, \mathtt{j/j}, \mathtt{p/p}, \mathtt{r/r}]) =$ $\mathtt{EQ_N rec}[\mathtt{i.j.p.K}'[\sigma', \mathtt{i/i}, \mathtt{j/j}, \mathtt{p/p}]](\mathtt{P}'[\sigma'], \mathtt{C}'_{\mathtt{zz}}[\sigma'], \mathtt{i.j.p.r.C}'_{\mathtt{ss}}[\sigma', \mathtt{i/i}, \mathtt{j/j}, \mathtt{p/p}, \mathtt{r/r}]) \in [\![(\mathtt{K})^-]\!]$. Call the left-hand constructor $\mathtt{R}$ and the right-hand $\mathtt{R}'$. By the IH, $\Psi \vdash \mathtt{I}[\sigma] = \mathtt{I}'[\sigma'] \in [\![\widehat{\mathtt{NAT}}]\!]$, $\Psi \vdash \mathtt{J}[\sigma] = \mathtt{J}'[\sigma'] \in [\![\widehat{\mathtt{NAT}}]\!]$, and $\Psi \vdash \mathtt{P}[\sigma] = \mathtt{P}'[\sigma'] \in [\![\widehat{\mathtt{EQ_N}}]\!]$. Then, by the definition of the LR for substitutions $\Psi \vdash \sigma, \mathtt{I}[\sigma]/\mathtt{i}, \mathtt{J}[\sigma]/\mathtt{j}, \mathtt{P}[\sigma]/\mathtt{p}, \mathtt{R/r} =$ $\sigma, \mathtt{I}'[\sigma']/\mathtt{i}, \mathtt{J}'[\sigma']/\mathtt{j}, \mathtt{P}'[\sigma']/\mathtt{p}, \mathtt{R/r} \in [\![(\Delta)^-, \mathtt{i} :: \widehat{\mathtt{NAT}}, \mathtt{j} :: \widehat{\mathtt{NAT}}, \mathtt{p} :: \widehat{\mathtt{EQ_N}}, \mathtt{r} :: (\mathtt{K})^-]\!]$. Since this matches the context in the $\mathtt{C}_{\mathtt{ss}}$ equality premise, we can apply the IH to get $\Psi \vdash \mathtt{C}_{\mathtt{ss}}[\sigma, \mathtt{I}[\sigma]/\mathtt{i}, \mathtt{J}[\sigma]/\mathtt{j}, \mathtt{P}[\sigma]/\mathtt{p}, \mathtt{R/r}] = \mathtt{C}'_{\mathtt{ss}}[\sigma', \mathtt{I}'[\sigma']/\mathtt{i}, \mathtt{J}'[\sigma']/\mathtt{j}, \mathtt{P}'[\sigma']/\mathtt{p}, \mathtt{R'/r}] \in [\![(\mathtt{K})^-]\!]$. On the left, we then use `whr-eqn-rec-beta-ss`, LEMMA B.26, closure under head expansion (LEMMA B.34), and the definition of substitution to get what we need. On the right, we use the definition of substitution and LEMMA B.26 to give the result.

2.
- Case for
$$\frac{\Delta \vdash \mathtt{K}_2 \equiv \mathtt{K}_1\,\mathtt{kind}}{\Delta \vdash \mathtt{K}_1 \equiv \mathtt{K}_2\,\mathtt{kind}}\ \texttt{deq-kd-sym}.$$
  By symmetry of the LR, $\Psi \vdash \sigma' = \sigma \in [\![(\Delta)^-]\!]$. By the IH, $\Psi \vdash \mathtt{K}_2[\sigma'] \iff \mathtt{K}_1[\sigma]\,\mathtt{kind}$. Then LEMMA B.19 gives the result.

- Case for
$$\frac{\Delta \vdash \mathtt{K}_1 \equiv \mathtt{K}_2\,\mathtt{kind} \quad \Delta \vdash \mathtt{K}_2 \equiv \mathtt{K}_3\,\mathtt{kind}}{\Delta \vdash \mathtt{K}_1 \equiv \mathtt{K}_3\,\mathtt{kind}}\ \texttt{deq-kd-trans}.$$
  By the IH, $\Psi \vdash \mathtt{K}_1[\sigma] \iff \mathtt{K}_2[\sigma']\,\mathtt{kind}$. By symmetry (LEMMA B.32), $\Psi \vdash \sigma' = \sigma \in [\![(\Delta)^-]\!]$, so by transitivity (LEMMA B.33) $\Psi \vdash \sigma' = \sigma' \in [\![(\Delta)^-]\!]$. Then we can apply the IH to the second premise with these substitutions to show $\Psi \vdash \mathtt{K}'_2[\sigma] \iff \mathtt{K}_3[\sigma']\,\mathtt{kind}$. Then LEMMA B.20 gives the result.

- Case for
$$\frac{}{\Delta \vdash \mathtt{TYPE} \equiv \mathtt{TYPE}\,\mathtt{kind}}\ \texttt{deq-kd-type}.$$
  Apply `norm-eq-kd-type`, and then use the definition of substitution to get the result.

- Case for
$$\frac{\Delta \vdash \mathtt{K}_1 \equiv \mathtt{K}_1' \,\mathtt{kind} \quad \Delta, \mathtt{u} :: \mathtt{K}_1 \vdash \mathtt{K}_2 \equiv \mathtt{K}_2' \,\mathtt{kind}}{\Delta \vdash \Pi_{\mathtt{k}}\, \mathtt{u}{::}\mathtt{K}_1.\, \mathtt{K}_2 \equiv \Pi_{\mathtt{k}}\, \mathtt{u}{::}\mathtt{K}_1'.\, \mathtt{K}_2' \,\mathtt{kind}}\ \texttt{deq-kd-pi}.$$

  By the IH applied to the first premise derivation, $\Psi \vdash \mathtt{K}_1[\sigma] \iff \mathtt{K}_1'[\sigma']\,\mathtt{kind}$. By rule, $\Psi, \mathtt{u} :: (\mathtt{K}_1)^- \vdash \mathtt{u} \longleftrightarrow \mathtt{u} :: (\mathtt{K}_1)^-$, so by LEMMA B.29, $\Psi, \mathtt{u} :: (\mathtt{K}_1)^- \vdash \mathtt{u} = \mathtt{u} \in [\![(\mathtt{K}_1)^-]\!]$. By weakening (LEMMA B.30), $\Psi, \mathtt{u} :: (\mathtt{K}_1)^- \vdash \sigma = \sigma' \in [\![(\Delta)^-]\!]$, so by the definition of the LR, $\Psi, \mathtt{u} :: (\mathtt{K}_1)^- \vdash \sigma, \mathtt{u}/\mathtt{u} = \sigma', \mathtt{u}/\mathtt{u} \in [\![(\Delta)^-, \mathtt{u} :: (\mathtt{K}_1)^-]\!]$. By the definition of context erasure, $\Psi, \mathtt{u} :: (\mathtt{K}_1)^- \vdash \sigma, \mathtt{u}/\mathtt{u} = \sigma', \mathtt{u}/\mathtt{u} \in [\![(\Delta, \mathtt{u} :: \mathtt{K}_1)^-]\!]$. Since this matches the context in the second premise, we can apply the IH to get that $\Psi, \mathtt{u} :: (\mathtt{K}_1)^- \vdash \mathtt{K}_2[\sigma, \mathtt{u}/\mathtt{u}] \iff \mathtt{K}_2'[\sigma', \mathtt{u}/\mathtt{u}]\,\mathtt{kind}$. By LEMMA B.11, $(\mathtt{K})^-$ is $(\mathtt{K}[\sigma])^-$. Thus `norm-eq-kd-pi` and the definition of substitution (to pull the substitution outside the Π on each side) give the result.

- Case for
$$\frac{}{\Delta \vdash \mathtt{NAT} \equiv \mathtt{NAT}\,\mathtt{kind}}\ \texttt{deq-kd-nat}.$$

  Apply `norm-eq-kd-nat`, and then use the definition of substitution to get the result.

3. Case for `deq-kd-eqn`. By the IH, $\Psi \vdash \mathtt{I}[\sigma] = \mathtt{I}'[\sigma'] \in [\![\widehat{\mathtt{NAT}}]\!]$ and $\Psi \vdash \mathtt{J}[\sigma] = \mathtt{J}'[\sigma'] \in [\![\widehat{\mathtt{NAT}}]\!]$. By LEMMA B.29, these are algorithmically equal. Then `aeq-kd-eqn` and the definition of substitution give the result.

□

DEFINITION B.37: IDENTITY SUBSTITUTIONS. $\mathtt{id}_{\Psi,\mathtt{u}::\widehat{\mathtt{K}}}$ is $\mathtt{id}_\Psi$,u/u, and $\mathtt{id}_\cdot$ is $\cdot$.

LEMMA B.38: IDENTITY SUBSTITUTIONS ARE LOGICALLY RELATED. $\Psi \vdash \mathtt{id}_\Psi = \mathtt{id}_\Psi \in [\![\Psi]\!]$.

Proof. By induction on the classifying context. The result is immediate by the definition when the context is empty. In the inductive case for $\Psi, \mathtt{u} :: \widehat{\mathtt{K}}$, by `neut-eq-cn-var` $\Psi \vdash \mathtt{u} \longleftrightarrow \mathtt{u} :: \widehat{\mathtt{K}}$, and then LEMMA B.29 gives logical relatedness. This combined with the inductive result gives the result. □

THEOREM B.39: COMPLETENESS OF ALGORITHMIC EQUALITY.

1. *If* $\Delta \vdash \mathtt{C} \equiv \mathtt{C}' :: \mathtt{K}$ *then* $(\Delta)^- \vdash \mathtt{C} \iff \mathtt{C}' :: (\mathtt{K})^-$.

2. *If* $\Delta \vdash \mathtt{K} \equiv \mathtt{K}'\,\mathtt{kind}$ *then* $(\Delta)^- \vdash \mathtt{K} \iff \mathtt{K}'\,\mathtt{kind}$.

Proof. Each part is immediate using LEMMA B.38, LEMMA B.39, LEMMA B.29, and the definition of identity substitutions. □

B.4 Algorithmic Kinding and Typing

Using algorithmic equality, we give a syntax-directed version of the kinding and typing rules. The single kind/type-conversion rule in the declarative judgement is replaced by equality premises on many rules.

Algorithmic kinding and typing are given by three judgements:

- $\Upsilon \vdash \mathtt{K}\ \overrightarrow{\mathtt{kind}}$ Operational interpretation: in the given context, check if K is a well-formed kind.

- $\Upsilon \vdash \mathtt{C}\ \overrightarrow{::}\ \mathtt{K}$ Operational interpretation: in the given context, synthesize a kind for C or fail.

- $\Upsilon; \Xi \vdash \mathtt{E}\ \overrightarrow{:}\ \mathtt{A}$ Operational interpretation: in the given contexts, synthesize a type for E or fail.

Ξ stands for a context containing algorithmic typing ($\mathtt{x} \overset{\rightarrow}{:} \mathtt{A}$) assumptions; Υ stands for a context containing algorithmic kinding ($\mathtt{u} \overset{\rightarrow}{::} \mathtt{K}$) assumptions. $(\Gamma)^+$ and $(\Delta)^+$ are the obvious function from declarative contexts to algorithmic typing ones. $(\Upsilon)^-$ translates algorithmic typing contexts to algorithmic equality ones, so its range is a Ψ. Well-formedness of these new contexts is defined in the usual manner. Because all erased kinds are well-formed, $(\Upsilon)^-$ is well-formed when Υ is.

Soundness and completeness are stated as follows:

THEOREM B.40: SOUNDNESS OF ALGORITHMIC TYPING AND KINDING. *Assume Δ and Γ are well-formed.*

1. *If* $(\Delta)^+ \vdash \mathtt{K} \ \overset{\rightarrow}{\mathtt{kind}}$ *then* $\Delta \vdash \mathtt{K}\,\mathtt{kind}$.
2. *If* $(\Delta)^+ \vdash \mathtt{C} \overset{\rightarrow}{::} \mathtt{K}$ *then* $\Delta \vdash \mathtt{C} :: \mathtt{K}$.
3. *If* $(\Delta)^+; (\Gamma)^+ \vdash \mathtt{E} \overset{\rightarrow}{:} \mathtt{A}$ *then* $\Delta\,;\,\Gamma \vdash \mathtt{E} : \mathtt{A}$.

Proof. In Twelf. □

THEOREM B.41: COMPLETENESS OF ALGORITHMIC TYPING AND KINDING.

1. *If* $\Delta \vdash \mathtt{K}\,\mathtt{kind}$ *then* $(\Delta)^+ \vdash \mathtt{K} \ \overset{\rightarrow}{\mathtt{kind}}$.
2. *If* $\Delta \vdash \mathtt{C} :: \mathtt{K}$ *then* $(\Delta)^+ \vdash \mathtt{C} \overset{\rightarrow}{::} \mathtt{K}'$ *for some* $\mathtt{K}'$ *such that* $\Delta \vdash \mathtt{K}' \equiv \mathtt{K}\,\mathtt{kind}$.
3. *If* $\Delta\,;\,\Gamma \vdash \mathtt{E} : \mathtt{A}$ *then* $(\Delta)^+; (\Gamma)^+ \vdash \mathtt{E} \overset{\rightarrow}{:} \mathtt{A}'$ *for some* $\mathtt{A}'$ *such that* $\Delta \vdash \mathtt{A}' \equiv \mathtt{A} :: \mathtt{TYPE}$

Proof. In Twelf. □

Note that in completeness, we only require that the algorithm synthesize some type in the equivalence class; indeed, LEMMA B.47 shows that our algorithmic judgements synthesize a unique type for a term. Using these theorems, we can show that $(\cdot)^+$ preserves well-formedness of its arguments.

B.5 Type Safety

THEOREM B.42: TYPE SAFETY FOR ALGORITHMIC TYPING. *Assume Δ and Γ are well-formed. If* $(\Delta)^+; (\Gamma)^+ \vdash \mathtt{E} \overset{\rightarrow}{:} \mathtt{A}$ *then for all* $\mathtt{E}'$ *such that* $\mathtt{E} \mapsto^* \mathtt{E}'$

- $(\Delta)^+; (\Gamma)^+ \vdash \mathtt{E}' \overset{\rightarrow}{:} \mathtt{A}'$ *where* $\Delta \vdash \mathtt{A}' \equiv \mathtt{A} :: \mathtt{TYPE}$
- *and either* $\mathtt{E}'$ $\mathtt{value}$ *or* $\mathtt{E}' \mapsto \mathtt{E}''$.

Proof. In Twelf. The proof is by the standard progress and preservation lemmas. The only slightly unusual part is that, for expedience, we show preservation only up to definitional equality; this allows us to prove the necessary substitution lemma directly as a consequence of substitution for the declarative system and equivalence of the algorithmic and declarative presentations. The algorithmic judgements make showing type safety easier in several ways:

- Because all the rules are syntax-directed, the inversion lemmas are proven by inspection; no induction is necessary.
- It is easy to show that equality of types implies equality of subcomponents. Showing this property directly for the declarative system would likely require a logical relations argument.

- In the case of progress for $\mathtt{NATcase}$, it necessary to show that the scrutinized constructor is either weak head reducible, $\mathtt{z}$, or $\mathtt{s\ I}$. Because the constructor is well-typed, it is algorithmically equal to itself, so the definition of algorithmic equality gives the result (because progress only considers closed terms, the constructor in question cannot be neutral). Establishing this property directly for the declarative presentation would be more difficult. Similar reasoning is used for $\mathtt{EQ_N case}$.

□

THEOREM B.43: TYPE SAFETY FOR DECLARATIVE TYPING. *If* $\Delta\,;\,\Gamma \vdash \mathtt{E} : \mathtt{A}$ *then, for all* $\mathtt{E}'$ *such that* $\mathtt{E} \mapsto^* \mathtt{E}'$, $\Delta\,;\,\Gamma \vdash \mathtt{E}' : \mathtt{A}$ *and either* $\mathtt{E}'\ \mathtt{value}$ *or there exists an* $\mathtt{E}''$ *such that* $\mathtt{E}' \mapsto \mathtt{E}''$.

Proof. In Twelf. Type safety is direct using soundness (THEOREM B.40) and completeness (THEOREM B.41) of algorithmic typing and type safety for algorithmic typing (THEOREM B.42). □

B.6 Decidability

LEMMA B.44: DECIDABILITY OF ALGORITHMIC EQUALITY FOR NORMALIZING CONSTRUCTORS AND KINDS. *By "not X", we mean "X implies a contradiction".*

1. *If* $\Psi \vdash \mathtt{K} \Longleftrightarrow \mathtt{K}'\ \mathtt{kind}$ *and* $\Psi \vdash \mathtt{L} \Longleftrightarrow \mathtt{L}'\ \mathtt{kind}$ *then either* $\Psi \vdash \mathtt{K} \Longleftrightarrow \mathtt{L}\ \mathtt{kind}$ *or not* $\Psi \vdash \mathtt{K} \Longleftrightarrow \mathtt{L}\ \mathtt{kind}$.
2. *If* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow \mathtt{C}_1' :: \widehat{\mathtt{K}}$ *and* $\Psi \vdash \mathtt{C}_2 \Longleftrightarrow \mathtt{C}_2' :: \widehat{\mathtt{K}}$ *then either* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow \mathtt{C}_2 :: \widehat{\mathtt{K}}$ *or not* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow \mathtt{C}_2 :: \widehat{\mathtt{K}}$.
3. *If* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow_{\mathtt{NAT}} \mathtt{C}_1'$ *and* $\Psi \vdash \mathtt{C}_2 \Longleftrightarrow_{\mathtt{NAT}} \mathtt{C}_2'$ *then either* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow_{\mathtt{NAT}} \mathtt{C}_2$ *or not* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow_{\mathtt{NAT}} \mathtt{C}_2$.
4. *If* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow_{\mathtt{EQ_N}} \mathtt{C}_1'$ *and* $\Psi \vdash \mathtt{C}_2 \Longleftrightarrow_{\mathtt{EQ_N}} \mathtt{C}_2'$ *then either* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow_{\mathtt{EQ_N}} \mathtt{C}_2$ *or not* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow_{\mathtt{EQ_N}} \mathtt{C}_2$.
5. *If* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{C}_1'$ *and* $\Psi \vdash \mathtt{C}_2 \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{C}_2'$ *then either* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{C}_2$ *or not* $\Psi \vdash \mathtt{C}_1 \Longleftrightarrow_{\mathtt{TYPE}} \mathtt{C}_2$.
6. *If* $\Psi \vdash \mathtt{C}_1 \longleftrightarrow \mathtt{C}_1' :: \widehat{\mathtt{K}}$ *and* $\Psi \vdash \mathtt{C}_2 \longleftrightarrow \mathtt{C}_2' :: \widehat{\mathtt{K}}'$ *then either* $\Psi \vdash \mathtt{C}_1 \longleftrightarrow \mathtt{C}_2 :: \widehat{\mathtt{K}}''$ *for some* $\widehat{\mathtt{K}}''$ *or not.*

Proof. The proof is by mutual lexicographic induction on the given derivations. It uses LEMMA B.13, LEMMA B.17, LEMMA B.15, LEMMA B.16, LEMMA B.19, and LEMMA B.20. □

THEOREM B.45: DECIDABILITY OF ALGORITHMIC EQUALITY.

1. *If* $\Delta \vdash \mathtt{K}\,\mathtt{kind}$ *and* $\Delta \vdash \mathtt{K}'\,\mathtt{kind}$ *then either* $(\Delta)^- \vdash \mathtt{K} \Longleftrightarrow \mathtt{K}'\,\mathtt{kind}$ *or not* $(\Delta)^- \vdash \mathtt{K} \Longleftrightarrow \mathtt{K}\,\mathtt{kind}$.
2. *If* $\Delta \vdash \mathtt{C} :: \mathtt{K}$ *and* $\Delta \vdash \mathtt{C}' :: \mathtt{K}$ *then either* $(\Delta)^- \vdash \mathtt{C} \Longleftrightarrow \mathtt{C}' :: (\mathtt{K})^-$ *or not* $(\Delta)^- \vdash \mathtt{C} \Longleftrightarrow \mathtt{C}' :: (\mathtt{K})^-$.

Proof. In each part, reflexivity (LEMMA B.5), completeness of algorithmic equality (THEOREM B.39), and decidability for normalizing kinds (LEMMA B.44) give the result. □

LEMMA B.46: CONSTRUCTORS HAVE UNIQUE WEAK HEAD NORMAL FORMS. *If* $\mathtt{C} \overset{\text{whr}}{\longrightarrow}{}^* \mathtt{C}'$ *and* $\mathtt{C} \overset{\text{whr}}{\longrightarrow}{}^* \mathtt{C}''$ *where* $\mathtt{C}'\ \mathtt{whnorm}$ *and* $\mathtt{C}''\ \mathtt{whnorm}$ *then* $\mathtt{C}'$ *is* $\mathtt{C}''$.

Proof. By induction on the first derivation. When one side ends in `whrrt-whr` and the other in `whrrt-refl`, either $\mathtt{C}'$ or $\mathtt{C}''$ is $\mathtt{C}$, so the premise derivation of $\mathtt{C} \xrightarrow{\text{whr}} \mathtt{X}$ combined with the derivation of $\mathtt{C}$ $\mathtt{whnorm}$ give a contradiction by LEMMA B.18; then we get the result vacuously. When both derivations are `whrrt-refl`, both $\mathtt{C}'$ and $\mathtt{C}''$ are $\mathtt{C}$. When bother derivations end in `whrrt-whr`, determinacy of weak head reduction (LEMMA B.15) and the IH give the result. □

LEMMA B.47: ALGORITHMS SYNTHESIZE UNIQUE KINDS AND TYPES.

1. *If* $\Upsilon \vdash \mathtt{C} \overset{\rightarrow}{::} \mathtt{K}$ *and* $\Upsilon \vdash \mathtt{C} \overset{\rightarrow}{::} \mathtt{K}'$ *then* $\mathtt{K}$ *is* $\mathtt{K}'$.
2. *If* $\Upsilon; \Xi \vdash \mathtt{E} \overset{\rightarrow}{:} \mathtt{A}$ *and* $\Upsilon; \Xi \vdash \mathtt{E} \overset{\rightarrow}{:} \mathtt{A}'$ *then* $\mathtt{A}$ *is* $\mathtt{A}'$.

Proof. The algorithmic typing and kinding rules are syntax-directed, so in each case the final rules of both derivations must be the same. Then, in each case, the result follows from the available inductive hypotheses, using LEMMA B.46 and simple properties of syntactic equality (reflexivity, symmetry, transitivity, congruence, and equality of subcomponents). □

LEMMA B.48: SOUNDNESS OF MANY-STEP WEAK HEAD REDUCTION.
If $\Delta \vdash \mathtt{C} :: \mathtt{K}$ *and* $\mathtt{C} \xrightarrow{\text{whr}}{}^{*} \mathtt{C}'$, *then* $\Delta \vdash \mathtt{C} \equiv \mathtt{C}' :: \mathtt{K}$.

Proof. In Twelf. □

LEMMA B.49: NORMALIZING TERMS OF KIND $\widehat{\text{TYPE}}$ HAVE WEAK HEAD NORMAL FORMS.
If $\Psi \vdash \mathtt{C}_1 \iff \mathtt{C}_2 :: \widehat{\text{TYPE}}$ *then there exists a* $\mathtt{C}_3$ *such that* $\mathtt{C}_3$ $\mathtt{whnorm}$ *and* $\mathtt{C}_1 \xrightarrow{\text{whr}}{}^{*} \mathtt{C}_3$.

Proof. By induction on the given derivation. In the case for `norm-eq-cn-whr-left`, the IH gives that there exists a $\mathtt{C}_3$ such that $\mathtt{C}_1' \xrightarrow{\text{whr}}{}^{*} \mathtt{C}_3$, and by premise $\mathtt{C}_1 \xrightarrow{\text{whr}} \mathtt{C}_1'$, so `whrrt-whr` gives the result. In the case for `norm-eq-cn-whr-right`, the result is immediate by the IH. In the case for `norm-eq-cn-type`, LEMMA B.17 applied to the premise and `whrrt-refl` give the result. No other rules derive a conclusion with the correct kind. □

THEOREM B.50: DECIDABILITY OF ALGORITHMIC TYPING AND KINDING.

1. *Given a context* Υ *and a kind* $\mathtt{K}$, *either* $\Upsilon \vdash \mathtt{K}\ \overrightarrow{\mathtt{kind}}$ *or not* $\Upsilon \vdash \mathtt{K}\ \overrightarrow{\mathtt{kind}}$.
2. *Given a context* Υ *and a constructor* $\mathtt{C}$, *either* $\Upsilon \vdash \mathtt{C} \overset{\rightarrow}{::} \mathtt{K}$ *for some* $\mathtt{K}$ *or not.*
3. *Given contexts* Υ *and* Ξ *and a term* $\mathtt{E}$, *either* $\Upsilon; \Xi \vdash \mathtt{E} \overset{\rightarrow}{:} \mathtt{A}$ *for some* $\mathtt{A}$ *or not.*

Proof. The first two parts are by mutual induction over the given kind and constructor; the third is by induction on the given term. The proof uses LEMMA B.47, LEMMA B.40, LEMMA B.4, LEMMA B.9, LEMMA B.10, LEMMA B.48, LEMMA B.46, and LEMMA B.49. □

THEOREM B.51: DECIDABILITY OF DECLARITIVE JUDGEMENTS.

1. *Given* Δ *and* $\mathtt{K}$, *either* $\Delta \vdash \mathtt{K}\ \mathtt{kind}$ *or not* $\Delta \vdash \mathtt{K}\ \mathtt{kind}$.
2. *Given* Δ, $\mathtt{K}$, *and* $\mathtt{K}'$, *either* $\Delta \vdash \mathtt{K} \equiv \mathtt{K}'\ \mathtt{kind}$ *or not* $\Delta \vdash \mathtt{K} \equiv \mathtt{K}'\ \mathtt{kind}$.
3. *Given* Δ, $\mathtt{C}$, *and* $\mathtt{K}$, *either* $\Delta \vdash \mathtt{C} :: \mathtt{K}$ *or not* $\Delta \vdash \mathtt{C} :: \mathtt{K}$.
4. *Given* Δ, $\mathtt{C}$, $\mathtt{C}'$, *and* $\mathtt{K}$, *either* $\Delta \vdash \mathtt{C} \equiv \mathtt{C}' :: \mathtt{K}$ *or not* $\Delta \vdash \mathtt{C} \equiv \mathtt{C}' :: \mathtt{K}$.

5. *Given* Δ, Γ, $\mathtt{E}$, *and* $\mathtt{A}$, *either* $\Delta\,;\,\Gamma \vdash \mathtt{E} : \mathtt{A}$ *or not* $\Delta\,;\,\Gamma \vdash \mathtt{E} : \mathtt{A}$.

Proof. The proof of each part is direct using various lemmas and the previous parts.

1. This part uses decidability, soundness, and completeness of algorithmic equality (THEOREM B.50, THEOREM B.40, and THEOREM B.41).
2. This part uses the previous part to establish well-formedness of the kinds in question, as the algorithm is only sound for well-formed kinds. It also uses regularity (LEMMA B.9) and decidability, soundness, and completeness for algorithmic kinding (THEOREM B.45, THEOREM B.22, and THEOREM B.39).
3. This part uses the previous part and synthesis of unique kinds (LEMMA B.47), as well as decidability, soundness, and completeness of algorithmic equality (THEOREM B.50, THEOREM B.40, and THEOREM B.41).
4. This part uses the previous part to establish well-kindness of the constructors in question. It also uses uses regularity (LEMMA B.9) and decidability, soundness, and completeness for algorithmic kinding (THEOREM B.45, THEOREM B.22, and THEOREM B.39).
5. This part uses the previous part and synthesis of unique kinds (LEMMA B.47), as well as decidability, soundness, and completeness of algorithmic equality (THEOREM B.50, THEOREM B.40, and THEOREM B.41).

$\square$