ELABORATING INDUCTIVE DEFINITIONS IN THE CALCULUS OF DEPENDENT LAMBDA ELIMINATIONS

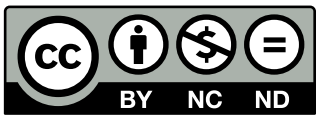
by

Christa Jenkins

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

August 2023

Thesis Committee: Aaron Stump, Thesis Supervisor
Brandon Myers
Cesare Tinelli
Garrett Morris
Omar Chowdhury



ACKNOWLEDGEMENTS

Many thanks to my advisor, Aaron Stump, for you support, encouragement, and advice throughout my time at Iowa. Your careful reading and detailed feedback, including for this dissertation, has been invaluable both in improving my research and myself as a researcher.

I am grateful to my committee — Brandon Myers, Cesare Tinelli, Garrett Morris, and Omar Chowdhury — for all your feedback during the proposal. You have been essential in helping me narratively focus and clarify this work, improving it as a work of scientific communication.

More broadly, I thank all the CLC members I have met. The seminars, lunches, coffee meetups, and other gatherings have always been friendly, welcoming, and invariably filled with stimulating discussion. Especially, thanks to Andrew, Alex, Tony, Larry, and Denis.

To my parents, James and Joyce, and my partner, Cheri, I give thanks for your love, understanding, and emotional support. To my close friend Valeri, I thank you for your late night speech that galvanized me to apply to graduate school. And finally, a heartfelt thanks to Nick for introducing me to dependent type theory. I doubt very much I would be where I am now without all of you.

ABSTRACT

In programming languages, algebraic datatypes along with the associated features of pattern matching and recursion provide notational convenience for problem-solving. This convenience helps programmers abstract away from low-level details and focus on the problem at hand, thereby helping them avoid software bugs. Because of this expressivity, however, high-level programming languages with these features are complex software artifacts, and their implementations may have subtle and undesirable interactions. For general-purpose languages these interactions may result in unintuitive edge cases, run-time crashes, unintended non-termination, or unexpected behavior. For programming languages that serve also as interactive theorem provers (ITPs), the consequences of bugs in such systems may be as dire as losing trust in proofs carried out within them.

While many lines of evidence can increase trust in software, such as testing, the most robust is formal, machine-checked verification. Unfortunately, formal verification of a fully-fledged programming language is a *Herculean* task. To facilitate this endeavor, language designers can turn to *elaboration*, a term of art describing the type-guided translation of a language into a simpler kernel language. Such a translation can then be proven correct, with the kernel language being easier to test or formally verify.

This dissertation concerns the elaboration of inductive datatypes into the *calculus of dependent lambda eliminations* (CDLE), a dependent type theory designed to

be both expressive and tiny. Unlike other type theories, which must bring datatypes and their associated induction principles into the kernel language, in CDLE it is possible to *derive* datatypes and their induction principles from a relatively small set of language features. This dissertation demonstrates how higher-level syntax for many desirable features — inductive datatype declarations, pattern matching, recursion and induction with a form of typed-based termination checking — can be soundly translated into CDLE.

PUBLIC ABSTRACT

Software is everywhere in modern life, to the extent that sometimes our lives depend upon it! But, how can we trust this software? Famously, computer scientist Edsger Dijkstra once said "Program testing can be used to show the presence of bugs, but never to show their absence!" Machine-checked verification offers the strongest guarantee that software is without bugs, and interactive theorem provers (ITPs) are a popular tool for this purpose. However, this then raises the question: who checks the checker? This is especially vexing as ITPs are themselves complicated pieces of software, with features being added to aid humans in verifying ever more complex software.

One of the ways to handle complexity like this is to translate it into simpler terms. Unlike the natural languages used by people, when dealing with programs and rigorous proof it is vital to ensure that no details are lost in translation. That is the focus of this dissertation: taking the near-ubiquitous feature of *inductive definitions,* which play a vital role in helping us write programs and reason about them while at the same time introducing complexity into the ITPs that implement them, and translating them in a meaning-preserving way into the much simpler theory of the Calculus of Dependent Lambda Eliminations.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

CHAPTER 1
INTRODUCTION

> The utility of a language as a tool of thought increases with the range of topics it can treat, but decreases with the amount of vocabulary and the complexity of grammatical rules which the user must keep in mind. Economy of notation is therefore important.
>
> Kenneth Iverson, *Notation as a Tool of Thought*

> The limits of my language means the limits of my world.
>
> Ludwig Wittgenstein, *Tractatus Logico-Philosophicus*

Many modern functional programming languages allow users to give inductive definitions of datatypes and functions. Programming languages are tools of thought, and considered as such inductive definitions are powerful in their expressiveness both to specify and reason about program behavior. This is all the more true for interactive theorem provers (ITPs) based on dependently typed functional languages, where users expect to carry out such reasoning in the *same language* as that used to write programs, using the Curry-Howard correspondence [75] to interpret types as *propositions* and programs as *proofs*. For ITPs, inductively defined datatypes are expected to also come with induction principles for reasoning.

However, there is inherent tension between powerful programming language features and *trust* in the correct implementation of these features, as that same power in the former can entail difficulties in the latter. This is, again, even more true of ITPs, as implementation bugs threaten the logical consistency of the system and therefore undermine trust in proofs carried out within the prover.

To underscore this point, consider some notable examples of unexpected behaviors arising from feature interactions in some well-known ITPs.

- **Dependent pattern matching in Agda**

  Dependent pattern matching is a popular feature that brings the familiar facilities of pattern matching over algebraic datatypes to dependent type theories. Since datatypes may be indexed in such settings (e.g., $Vec \cdot A\ n$ is the type of lists containing n elements of type A), case branches in pattern matching may reveal information about indices that should be propagated in the right-hand side of a defining equation (e.g., in the empty list case, we know $n = 0$). The original proposal for dependent pattern matching by [24], and the default form present Agda, implies Streicher's [77] **K** axiom for propositional equality.

  While unproblematic on its own, axiom **K** is inconsistent with *homotopy type theory* (**HoTT**) [89]. Agda's development team introduced an optional flag `--without-K` to make dependent pattern matching safe for investigations into **HoTT**. Unfortunately, the first implementation of `--without-K` [68] was without formal justification, and Cockx [21] demonstrated it to be incompatible with **HoTT**. Cockx [22] put dependent pattern matching without **K** back on solid theoretical foundations by showing how it may be soundly translated (using *elaboration*) to a core theory without **K**.

- **Coinductive types in Coq**

  While inductive types serve to structure the definitions of functions that consume necessarily finite data, *coinductive types* structure the definitions of func-

tions that *generate possibly infinite* data. For example, the coinductive type $\mathit{Stream} \cdot A$ is the type of infinite lists containing elements of type A. Prior to version 8.5, the Coq proof assistant [86] only supported coinductive definitions in terms of constructors (e.g., $\mathit{cons} : A \to \mathit{Stream} \cdot A \to \mathit{Stream} \cdot A$). However, semantically coinductive types are properly understood as being specified by their *destructors* (e.g. $\mathit{head} : \mathit{Stream} \cdot A \to A$ and $\mathit{tail} : \mathit{Stream} \cdot A \to \mathit{Stream} \cdot A$). This mismatch of semantics and implementation resulted in a loss of *subject reduction*, the property that running a well-typed program produces a well-typed result [86] With version 8.5, Coq supports definitions of coinductive types aligned with their semantics, and as of version 8.9, the earlier constructor-based formulation is considered deprecated.

- **Impredicativity and proof-irrelevance in Lean**

  Impredicativity and proof-irrelevance are two powerful (and orthogonal) features in ITPs. Impredicative quantification in a type T is quantification over a collection of types that includes T itself. One common definition of proof irrelevance for a type T is that any two terms of that type are equal. The Lean theorem prover [30] supports an impredicative *Prop* sort with a strong form of proof-irrelevance. In his master's thesis, Cerneiro [18] left open the question of whether strong normalization (the property that all well-typed programs terminate) held for Lean; Abel and Coquand [2] gave a counter-example, showing in particular that these two features were responsible for its failure. Although this result does not appear to impact the logical consistency of Lean, it was

nonetheless an undesired and unexpected consequence of feature interaction.

1.1 The Rôle of Elaboration

> The proof assistant itself is 'just another program', so its correctness can be verified. To do this, one... has to formalize the rules of the logic. Then one would prove that the proof assistant can prove a theorem ϕ if and only if ϕ is derivable in the logic.
>
> Herman Geuvers, *Proof assistants: History, ideas, and future*

To avoid scenarios such as those given above, it is desirable to prove that a given type theory satisfies certain meta-theoretic properties. However, this raises a new difficulty: each time language implementers add or extend a feature, must they re-do all meta-theoretic results for the entire system? For mature ITPs, this can be monumental to perform even *once*, let alone for every iteration of the development of the language. Fortunately, there is an alternative: for some features, language designers may choose *definitional* language extensions justified by *elaboration*. Informally, elaboration is a term of art describing:

1. the designation of a high-level **source language** with convenient features for users;
2. the designation of a low-level **target language**, usually a small core (sub)language, which is the object of meta-theoretic study;
3. a **translation** from the source language to the target language; and
4. a **soundness proof** showing that this translation preserves the meaning (static and dynamic) of the source language in the target language.

Elaboration can thus be seen as giving *translational semantics* for source-language features, and usually requires that such features be definitional or schematic exten-

sions to the core theory.

In this setup, language designers desire two characteristics for the target language. First, while the target language is not expected to be ergonomic to program in directly, it is important that it be *logically expressive* so that the many complex features desired of the source language are derivable within it. Second, the target language should be *small*. This is to facilitate trust in the core theory, as reducing the number and complexity of constructs in the target language means both that the theory is easier to reason about and also that the size of implementations of the target language can be reduced. This second point in turn means it is easier to formally verify that an implementation of the target language correct, or to have multiple independent checkers to establish trust by consensus.

1.2 CDLE as a Target Language

In this dissertation, we will consider one candidate in particular to serve as our target language: the *Calculus of Dependent Lambda Eliminations* (CDLE). CDLE was designed to be both formally tiny and logically expressive: the core theory is described concisely in ∼20 typing rules by Stump [81], and can be implemented in ∼1K lines of Haskell code; within the theory, one can derive induction principles for lambda encoded datatypes [36], monotone recursive types [47], rich recursion schemes [84], simulated large eliminations [45], and zero-cost program and proof reuse [31]. The surface language, Cedille, has been and continues to be designed bottom-up from CDLE, with its features guided by the results obtained within CDLE.

We give now a brief justification of our choice of CDLE by comparing to other

likely candidates.

- **The Calculus of Constructions (CC)**

  CDLE contains CC as a subsystem, and so of course the latter theory is formally simpler. However, concerning logical expressivity CC suffers a significant deficiency: lack of induction for datatypes. Geuvers [37] showed that induction is not derivable for lambda-encoded data in $\lambda P2$ (a subsystem of CC) by constructing a counter-model, and the addition of higher-order type constructors in CC does not appear to change the situation.

  A secondary issue is the lack of efficient data accessors. Specializing to the case of lambda encoded natural numbers, this is the problem of defining an efficient predecessor function. While this is not known to be impossible, there are negative results that suggest as much: Parigot [70] showed there is no efficient predecessor for Church-encoded naturals, and Spławski and Urzyczyn [76] show that System F with positive recursive types (which is sufficient for giving efficient data accessors) cannot have a dynamically sound translation to System F (i.e., a reduction step in the source language does not correspond to a bounded number of reductions in the target language). Since the terms and types of $\lambda P2$ can be efficiently translated to System F by erasing dependencies, it follows there is no efficient translation of these positive recursive types into $\lambda P2$ either.

- **Theories with general inductive datatypes**

  To address the above difficulties with CC, Werner [95] formulated the *Calculus*

of Inductive Constructions (CIC), which extends CC with type inference and computation rules for inductive definitions. CIC underlies ITPs such as Coq and Lean, making it a time-proven choice for an expressive target language. However, with this expressivity comes a cost: datatypes are formally complex entities, requiring for example a positivity criteria to ensure logical consistency and normalization. Thus, bringing datatypes into the target language significantly increases its complexity. Furthermore, the situation is not guaranteed to be stable, as in the future new classes of datatypes that fall outside the form axiomatized in the target language, such as coinductive, nonstricly positive, inductive-inductive, and inductive-recursive, may be desired, requiring rework of the meta-theory.

- **Theories with a closed universe of strictly positive datatypes**

  Dagand and McBride [28] demonstrated that a class of inductive definitions similar to those supported in CIC can be translated into a modest extension of Martin-Löf type theory (MLTT) containing a cumulative universe hierarchy, a universe of labels, large eliminations, and a closed universe of inductive type formers. Compared to theories with general inductive definitions, this reduces significantly the complexity of the core theory. However, the problem of future extensibility remains, as only strictly positive inductive types are supported.

In fairness, we note that there are other criteria language implementers may have that would cause them to reject CDLE as a foundational logic. One may reject the presence of impredicativity or CDLE's commitment to extrinsic typing on

philosophical grounds, or its relatively weak termination guarantee on grounds both practical and philosophical. However, in assessing the suitability of a constructive logic to serve as a core language for an ITP using the criteria of logical consistency, expressivity, and formal simplicity, it is our opinion that CDLE improves significantly upon the candidates discussed above.

1.3 How to Read This Dissertation

We now explain the organization of the remainder of this dissertation. Throughout, we use authorial first-person plural pronouns unless the emphasis of the prose is original work, in which case first person singular pronouns will be used. Each of the subsequent chapters will begin with a short preamble followed by three pieces of meta-text.

Prospectus. Paragraphs beginning with **Prospectus** provide summary of the remaining sections of the chapter.

Reading Guide. Paragraphs beginning with *Reading Guide* assign reading priorities to the chapter material, indicating what we believe is important for understanding the dissertation as a whole and what might be prudent for most readers to skim on a first pass.

Original Contributions. In paragraphs beginning with *Original Contributions*, I identify the material presented in this dissertation that is my original work and that which is due to other authors.

The material in this dissertation is at times quite technical and intricate. To help orient the reader, we provide here in Figure 1.1 a high-level overview of the

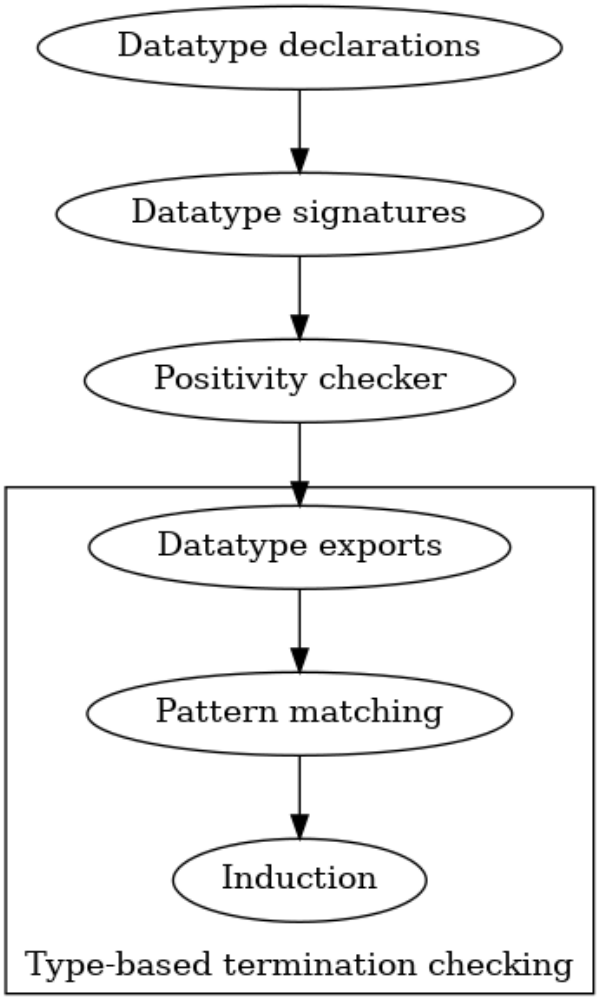


Figure 1.1: High-level outline of surface language datatype system

components of Cedille's datatype system. For now, we only focus on the process from the point of view of the surface language; subsequent chapters elaborate on the rôle of the target language for each of these components.

1. The user writes a datatype declaration, which includes the name of the datatype and the type of its constructors. These constructors are type-checked.
2. We elaborate the *signature* of the declared datatype, which is the part of the datatype that describes its unique shape (number and arities of its constructors).
3. A *positivity checker* for the signature ensures that admitting the declared datatype would not introduce logical inconsistency or violate CDLE's termination guarantees.
4. The datatype is admitted, and we export both the expected identifiers (the datatype name and its constructors) and some unexpected ones, used for Cedille's type-based termination checking.
5. The datatype exports enable a novel form of pattern matching we call *course-of-values pattern matching*, a crucial component of Cedille's approach to termination checking. For the *cognoscenti*, note that this is not a form of *dependent pattern matching*, but more akin to case tree notation for dependent eliminators.
6. Datatypes in Cedille come with induction principles. When induction in Cedille is combined with course-of-values pattern matching, this yields *course-of-values induction*, an expressive induction scheme in which the induction hypothesis can be invoked on dynamically computed subdata.

1.3.1 Contributions

With the overview of Cedille's datatype system given, the original contributions of this dissertation are as follows (results I deem of greater significance are proceeded by a star):

- some meta-theoretic results for CDLE, including subject reduction and normalization for types constructors and kinds under call-by-name operational semantics (Chapter 3);
- ⋆ the evidence producing positivity checker and its soundness proof (Chapter 4);
- ⋆ the elaboration of signatures from datatype declarations and its soundness proof (Chapter 4);
- the design of the surface-language constructs that facilitate course-of-values pattern matching and induction (Chapter 5);
- ⋆ the elaborating type inference rules for these constructs and the proof that the judgments elaborating Cedille to CDLE are sound (Chapter 5).

Of course, each of the above-mentioned results are described in greater detail in their respective chapters.

1.3.2 Overivew

The remainder of this dissertation is organized as follows.

- To motivate the results presented in this dissertation, Chapter 2 provides an extended primer on inductive definitions in ITPs: the intuition behind them, their uses, and the concerns raised when introducing them into a type theory.

In particular, this chapter covers several recursion schemes (each of which are reducible to course-of-values induction) to highlight the expressivity provided by Cedille's type-based termination checking. These recursion schemes are discussed first informally, then more rigorously, and the theoretical foundation of Cedille's type based termination checking is explained (the "Mendler-style" of coding recursion).

- As we mentioned earlier, Cedille is designed bottom-up from derivations carried out within CDLE; furthermore, CDLE is a sublanguage of Cedille. We therefore, in Chapter 3, detail the theory and meta-theory of CDLE, and describe some derivations carried out within it that will be used by the datatype system. This culminates in a specification of a combinator for course-of-values induction in CDLE.
- Chapter 4 begins the process outlined in Figure 1.1 for Cedille's treatment of datatypes by describing the elaboration of *signatures* from datatype declarations and an evidence-producing positivity checker for them. The only surface language syntax treated here is the declaration of datatypes.
- In Chapter 5, we finish our formal presentation of the Cedille surface language. In particular, we describe and elaborate the identifiers that admitted datatype declarations export to the user, and present the elaborating type inference rules for the syntactic constructs for course-of-values pattern matching and induction.

CHAPTER 2
INDUCTIVE DEFINITIONS

> Inductive definitions occur in two different roles, and we call them *fundamental* and *non-fundamental* accordingly . . . [T]he definition of 'entity' is the fundamental inductive definition. It establishes the domain of objects for the arithmetic. Then, the non-fundamental inductive definitions. . . apply to objects already known in their status as entities . . . Thus the fundamental inductive definition establishes the range of a variable, over which one may subsequently define predicates by non-fundamental inductive definitions.
>
> Stephen Kleene, *Introduction to Metamathematics*

Inductive datatypes, also known as *algebraic datatypes* or *ADTs*, are a popular feature of functional languages that combine a scheme for introducing new types with a mechanism for defining functions over them by pattern matching and recursion. This popularity extends to implementations of type theory, in particular proof assistants based on dependent type theories, wherein proofs are written using these same features. These features of inductive types — datatype declarations, pattern matching, and recursion — are the primary focus of this dissertation, and we will collectively refer to them as forms of *inductive definition*.

The appeal of inductive datatypes lies in the concise way in which the programmer may describe how to *produce* and *consume* elements of a datatype. Production is determined by the list of constructors given in the datatype declaration, where we understand that the set of elements of the type is the *least* set freely generated by these constructors. This understanding justifies a form of definition called *pattern matching*, in which only the constructor forms need be considered when defining a function consuming a datatype. In addition, programmers may define such functions

recursively without fear of nontermination, provided that recursive calls are restricted to progressively smaller elements of the datatype.

This last point introduces the main wrinkle in bringing inductive definitions to proof assistants from general-purpose functional programming languages. Unlike general-purpose functional languages where general recursion is the norm, in ITPs recursion must be carefully controlled to preserve termination guarantees and logical consistency. The orthodox approach to dealing with this issue consists of two parts: first, ensure that the datatype is *well-founded* by requiring its declaration be *positive*; second, ensure that recursive calls respect the well-founded ordering inherent to positive inductive types. This chapter serves as a gentle introduction to these issues and how they are safeguarded in the surface-language syntax of Cedille.

2.0.1 About This Chapter

Prospectus. We informally review inductive definitions in type theory in Section 2.1, presenting examples of the kinds of inductive definition we wish to support for their expressive capabilities, and the non-examples of non-well-founded definitions we wish to prohibit due to their threat to the language's logical consistency and termination guarantees. Because we cannot accept all terminating definitions while excluding all nonterminating ones, we look to formally describe a particular family of inductive definitions to support in the surface language. For datatype declarations, we shall require that they satisfy a certain *positivity* criterion, to be formalized later in Chapter 3. For recursive functions, the formalization is given as a terminating recursive combinator, which we slowly build up to in Section 2.2. Throughout, we give informal

justifications for the ways in which termination and logical consistency are guaranteed; the formal justification, which we postpone for now, comes from the meta-theoretic properties of the CDLE target language (Chapter 3) and the proof that elaboration of Cedille to CDLE is sound (Chapter 5). To bridge the gap between the material presented in this chapter and the formal description of surface language in Chapter 5, in Section 2.2 we use this combinator to reimplement the desirable examples shown in Section 2.1 and discuss the mechanisms by which non-examples are prohibited.

Reading Guide. Readers who are familiar with use of structured recursion schemes in general, and Mendler-style combinators for recursion in particular, to control nontermination may skim Sections 2.1 to familiarize themselves with aspects of the surface language syntax and read Section 2.2.6 for the details of the Mendler-style recursive combinator upon which is founded type-based termination checking in Cedille before proceeding to the next chapter. For all other readers, it is **strongly recommended** to read this chapter carefully.

Original Contributions. This chapter consists mostly of review of background material. The surface language syntax for Cedille, which in this chapter is used for informal exposition, is my original work (see Jenkins et al. [43]).

2.1 Informal Overview

This section aims to develop the reader's informal intuitions about inductive definitions in type theory. Our starting point is the difficulties in perserving logical consistency and termination guarantees when importing ADTs and recursive definitions from functional languages to ITPs. Thus, the reader is assumed to have some

```
data Bool : ⋆
= true  : Bool
| false : Bool .

data Nat : ⋆
= zero : Nat
| succ : Nat → Nat .
```

Figure 2.1: Booleans and natural numbers as inductive datatypes

familiarity with ADTs in general-purpose functional languages. We treat each of the features, and the concerns they raise, as outlined in the introduction of this chapter: Section 2.1.1 covers datatype declarations, Section 2.1.2 covers pattern matching and the need for datatypes to be positive, and Section 2.1.3 covers recursion and termination checking. To facilitate reading the Cedille syntax used throughout this section, listings of Cedille code include pseudocode using syntax in the style of Agda and Haskell.

2.1.1 Inductive Datatype Declarations

Figure 2.1 shows an example of the proposed Cedille surface language syntax for datatype declarations. Modulo small differences in punctuation, the syntax is similar to that found in languages with support for inductive datatypes such as Agda, Coq, Haskell, or Idris. Beginning with the `data` keyword, we give the name of the datatype being declared (*Bool* or *Nat*), its kind ($\star$, the kind of types), followed by a list of constructors separated by a vertical bar and terminated by a period — datatypes *Bool* and *Nat* have two constructors each.

Datatype *Bool* is non-recursive, and so only has two closed values: *true* and

```
if : ∀ X: ⋆. Bool → X → X → X
= Λ X. λ b. λ t. λ f.
  σ b {
  | true → t
  | false → f
  } .

pred : Nat → Nat
= λ n. σ n {
  | zero → n
  | succ n' → n'
  } .

{- Pseudocode
if : ∀ X: ⋆. Bool → X → X → X
if true  t f = t
if false t f = f

pred : Nat → Nat
pred n@zero = n
pred (succ n') = n'
-}
```

Figure 2.2: If-then-else and predecessor using pattern matching

false. Natural numbers are the go-to example for an inductive definition, as the constructor for successor (*succ* in the figure) recursively refers to the datatype we are defining in the type of its argument. Thanks to this recursion, there are infinitely many closed values of type *Nat*: *zero*, *succ zero*, *succ* (*succ zero*), and so on.

In the discussion to follow, we will be concerned not only with pattern matching and recursion as language features, but also with the measures taken in Cedille to ensure that these may be soundly translated to CDLE.

2.1.2 Pattern Matching

We now turn our attention to pattern matching. Examples of the Cedille surface language syntax for this feature are given in Figure 2.2, accompanied by equivalent Agda-style pseudocode for the benefit of the reader (in Cedille, text enclosed in {- and -} is a multi-line comment). The first, *if*, is the *if-then-else* conditional expression for Booleans. It is polymorphic in the type of the *then* and *else* branches, indicated by universal quantification over the type variable X. The argument $b : \mathit{Bool}$ is scrutinized by the case analysis operator σ; we therefore call b the "scrutinee" of case analysis.

In the σ expression, the programmer is obligated to handle exactly two cases, one each for the two constructors. Within the enclosing curly braces of the σ-expression, we refer to *true* and *false* as the *patterns*. In Cedille's syntax, patterns occur to left of the arrow, and to the right of the arrow is the *case body*. If a value given for b matches the pattern *true*, the case body for *true* (in this situation, t) is the result of the σ-expression; if it matches *false*, then that case body is the result.

For the inductive datatype *Nat*, whose definition uses recursion, pattern matching gives access to the predecessor in the *succ* case. This is illustrated with *pred* in the figure. When the original argument n to *pred* is not *zero*, its predecessor is bound as $n' : \mathit{Nat}$. This illustrates that pattern matching not only gives us control flow, but also gives the programmer access to the underlying data (in this situation, the predecessor) that we are guaranteed is present when a branch is selected.

Remark 2.1. The reader may wonder why *pred* returns n in our example, and not simply *zero*, when pattern matching reveals that n is *zero*. This is addressed in

Section 2.2.6.

We will give a formal semantics to σ expressions in Chapter 5; for now, understand that the syntax $\sigma\ t\ \{\ |\ p_1 \to t_1\ |\ \ldots\ |\ p_n \to t_n\ \}$ is an expression computed by cases over a datatype element t, where $(p_i)_{i=1\ldots n}$ give all n constructor forms and $(t_i)_{i=1\ldots n}$ are the case branches. Within each t_i, we are able to refer to constructor arguments given in the pattern p_i (such as n' in the successor case of *pred*).

2.1.2.1 Computing by Cases

To understand pattern matching, we need to know not only how to write programs using this feature but also what those programs *mean*, i.e., how they compute. In type theory, computation occurs when an elimination form of a type acts on an introduction form of that type. For example, the elimination form for functions is application, $t_1\ t_2$, the introduction form is abstraction (that is, a lambda expression) $\lambda\, x.\, t$, and computation (in the form of a β-reduction) occurs when a function abstraction is applied to an argument: $(\lambda\, x.\, t)\ t_2 \dashrightarrow t[t_2/x]$, where $t[t_2/x]$ denotes the capture-avoiding substitution of the free occurences of x in t with t_2. For pattern matching in Cedille, the eliminator is σ-expressions and the introduction form is a constructor form. When these meet, the appropriate case branch is selected with any variables bound in the pattern for that branch substituted with the constructor arguments.

Figure 2.3 shows an example program execution using *pred* and *if* (defined in Figure 2.2). For ease of reference, each step of execution is numbered, given in the left column of the figure. In each row, the subexpression to which we apply a

$$
\begin{array}{lll}
0. & & \mathit{pred}\ (\mathit{if}\ \mathit{true}\ (\mathit{succ}\ \mathit{zero})\ \mathit{zero}) \\
1. & =_{\mathbf{df}} & \mathit{pred}\ (\underline{(\lambda\, b.\, \lambda\, t.\, \lambda\, f.\, \sigma\ b\ \{\ |\mathit{true} \to t \mid \mathit{false} \to f\ \})\ \mathit{true}}\ (\mathit{succ}\ \mathit{zero})\ \mathit{zero}) \\
2. & \dashrightarrow & \mathit{pred}\ (\underline{(\lambda\, t.\, \lambda\, f.\, \sigma\ \mathit{true}\ \{\ |\mathit{true} \to t \mid \mathit{false} \to f\ \})\ (\mathit{succ}\ \mathit{zero})}\ \mathit{zero}) \\
3. & \dashrightarrow & \mathit{pred}\ \underline{(\lambda\, f.\, \sigma\ \mathit{true}\ \{\ |\mathit{true} \to \mathit{succ}\ \mathit{zero} \mid \mathit{false} \to f\ \})\ \mathit{zero}} \\
4. & \dashrightarrow & \mathit{pred}\ \underline{(\sigma\ \mathit{true}\ \{\ |\mathit{true} \to \mathit{succ}\ \mathit{zero} \mid \mathit{false} \to \mathit{zero}\ \})} \\
5. & \dashrightarrow & \underline{\mathit{pred}}\ (\mathit{succ}\ \mathit{zero}) \\
6. & =_{\mathbf{df}} & \underline{(\lambda\, n.\, \sigma\ n\ \{\ |\mathit{zero} \to n \mid \mathit{succ}\ n' \to n'\ \})}\ (\mathit{succ}\ \mathit{zero}) \\
7. & \dashrightarrow & \underline{\sigma\ (\mathit{succ}\ \mathit{zero})\ \{\ |\mathit{zero} \to \mathit{succ}\ \mathit{zero} \mid \mathit{succ}\ n' \to n'\ \}} \\
8. & \dashrightarrow & \mathit{zero}
\end{array}
$$

Figure 2.3: Example computation of *if* and *pred*

computation rule is shown underlined; these are known as *reducible expressions*, often shortened to *redexes*. In the middle column, replacement of a named term with its definition is indicated with $=_{\mathbf{df}}$, and a step of computation is indicated with $\dashrightarrow$.

After replacing *if* with its definition in line 1 and applying the lambda abstraction to its three arguments in lines 2, 3, and 4, we obtain a redex in which the scrutinee of the σ expression is the constructor *true* for *Bool*. This leads to line 5, where the σ expression is replaced by the body of the *true* case branch. Then, we replace *pred* with its definition in line 6 and reduce the resulting function application in line 7, leaving us with a redex involving σ and the value *succ zero*. We obtain line 8 by rewriting the redex to the body of the successor branch, where the bound variable n' is substituted with *zero*.

Remark 2.2. In Figure 2.3, we used the call-by-value reduction strategy for ease of readability. However, it should be noted that Cedille uses call-by-name operational semantics.

```
not : Bool → Bool
= λ b. if b false true .

notInvolutive : Π b: Bool. { not (not b) ≃ b }
= λ b. σ b {
  | true  → refl true
  | false → refl false
  } .

{- Pseudocode
notInvolutive : Π b: Bool. { not (not b) ≃ b }
notInvolutive true  = refl true
notInvolutive false = refl false
-}
```

Figure 2.4: Proof `not` is involutive

2.1.2.2 Proving by Cases

In ITPs, we do not only care about computing with inductive types; we also care about using them to carry out inductive proofs. We have not yet discussed the features that enable us to express a proper induction principle for inductive types such as *Nat*, as we have not introduced a mechanism for recursion. However, we *do* expect pattern matching to give us (at least) a form of proof definition that we may think of as "proof by cases": to prove that a predicate holds for all inhabitants of a datatype, it suffices to show it holds for each of the constructor forms.

We illustrate this method of reasoning in Figure 2.4, using as an example a proof that Boolean negation in involutive. First, the negation operator *not* is implemented using *if*. Next, for our proof *notInvolutive* we use Π to construct a dependent function type. Read the type of *notInvolutive* as saying that for all Booleans

```
data Bad (X: ⋆) : ⋆
= bad : (Bad → X) → Bad .

selfApp : ∀ X: ⋆. Bad ·X → X
selfApp (bad f) = f (bad f)

diag : ∀ X: ⋆. X
diag = selfApp (bad selfApp)
```

Figure 2.5: Nontermination with pattern matching and nonpositive datatypes (pseudocode)

b, *not* (*not* b) is equal to b. The proof proceeds by cases on the argument b using operator σ. In the case that b is *true*, all occurences of b in the return type are rewritten to *true*. So, the subexpression *not* b of the return type becomes *not true*, which computes to *false*. With the left-hand side of the equation now being *not false*, it can compute further to *true*. Now, the return type is $\{true \simeq true\}$, which is a reflexive equation and therefore provable with *refl true* : $\{true \simeq true\}$ (see Chapter 3 for the details of Cedille's equality type).

2.1.2.3 Contravariance and Nontermination

Without suitable restrictions, it is possible use datatype declarations and pattern matching *alone* [63] to construct closed terms that are nonterminating and render the underlying logic inconsistent, a disastrous situation for any ITP. Figure 2.5 shows an example of this using Agda-style pseudocode.

The parameterized datatype *Bad* is defined having a single constructor, *bad*, whose sole argument is a function of type $Bad \cdot X \to X$, where $X : \star$ is a parameter to the datatype declaration. Note that in Cedille, the types of constructors for a

parameterized like *Bad* omit references to the type parameter, and an application whose argument is a type or type constructor is denoted with a center dot. Unlike *Nat*, the type of the constructor for *Bad* makes a recursive reference to the datatype in a *contravariant* (or *negative*) position, as it is the domain of the function type $Bad \to X$. This makes the argument to *bad* act as a *consumer* of the datatype, rather than as a *producer*.

The contravariant occurence of the datatype in its constructor's argument type can be combined with pattern matching to define a form of self-application (*selfApp*), wherein a function $f : Bad \cdot X \to X$ that is used in the construction of the value given to *selfApp* (that is, *bad f*) is applied *to* that value. When we apply *selfApp* to itself in this way, we get the looping term *diag* whose type tells us all types are inhabited — the definition of logical inconsistency.

$$diag =_{\mathbf{df}} selfApp\ (bad\ selfApp) \dashrightarrow selfApp\ (bad\ selfApp) \dashrightarrow \ldots$$

The most common (though not the only) mechanism for prohibiting self application in this manner is to require that the types of arguments for datatype constructors satisfy a *positivity criterion* : recursive occurrences to the datatype being defined may only appear *covariantly*, preventing arguments to datatype constructors from being able to act as consumers of the datatype. This will be the restriction enforced by Cedille's datatype system; for more details, see Chapter 4.

2.1.3 Recursion

In general-purpose functional programming languages, *general recursion* is a mechanism by which programmers may write recursive term definitions without any

restrictions on the recursive occurences of the term being defined. General recursion is a powerful language feature — too powerful, in fact, to be included in ITPs without caveat. In the presence of general recursion, we can show that every type T is inhabited, rendering the underlying logic of the system inconsistent.

```
loop : T
loop = loop
```

Though it is possible to admit general recursion in an ITP by separating the logical and programming fragments of the language (such as in Zombie [17]), most ITPs do not take this approach. Instead, they usually restrict recursive function definitions to unproblematic forms by tying them to (co)inductive datatypes. When a recursive function is defined in terms of an inductive datatype such that it respects the well-founded ordering of that type, we say such a function is *inductively defined*.

The question arises: by what mechanism shall we check that inductive definitions of functions respect the well-founded ordering of an inductive type? Put another way, *how will termination checking be implemented?* For designers of ITPs, the goal is to expand the class of recursive definitions readily accepted by the termination checker without introducing logical inconsistency. The ITPs Agda, Coq, and Idris implement *syntactic* termination checking, under which recursive calls can be made only on (application spines whose heads are) pattern variables [40]. This regime of termination checking suffers a number of well-known drawbacks (see Barthe et al. [11]).

- It's *non-compositional:* refactoring can cause a previously accepted function to be flagged by the termination checker, even if the two versions are equivalent.

This can to some extent be addressed by devices such as pragmas for inlining functions, though this means users must become aware of this "bag of tricks" to use the system more effectively.

- It's *complicated:* syntactic termination checkers ensure that a given definition satisfies a predicate that is *global* to a particular definition, such as whether a variable is bound in a constructor pattern. As language designers expand the supported class of inductive definitions, such as those that swap arguments, rely on a lexicographic ordering, or are mutually recursive, it becomes more difficult to correctly specify and implement this predicate.

An alternative, compositional approach to syntactic termination checking is *type-based termination checking,* in which the type system itself is used to guarantee inductive definitions of functions are well-founded. The most popular incarnation of type-based termination checking is *size-indexed* types [11], a form of which is implemented in Agda [1]. Though size-indexed types (often shortened to simply *sized types*) avoid the problems inherent to syntactic termination checkers, they can introduce new difficulties, as an existing type theory that wishes to support them must be extended with new typing primitives. As an example of such a difficulty, the existing formulation of sized types in Agda has been shown to be unsound [85].

Cedille's surface language uses a form of type-based termination checking. We postpone discussion of the particular formulation Cedille uses, and comparisons to other extant termination checking methods, to Sections 2.2 and 2.3. Instead, for the remainder of this section we build an informal intuition for the class of inductive

```
minus : Nat → Nat → Nat
= λ m. λ n. μ m-minus. n {
  | zero → m
  | (succ n') → pred (m-minus n')
}

{- Pseudocode
minus : Nat → Nat → Nat
minus m n = m-minus n
  where
  m-minus : Nat → Nat
  m-minus zero = m
  m-minus (succ n') = pred (m-minus n')
-}
```

Figure 2.6: Subtraction and the less-than operator (iteration)

definitions we will support in the surface language, discussing the relative trade-offs between syntactic and type-based termination checking for supporting this class as we proceed.

2.1.3.1 Iteration

The first variety of inductive definition we consider, *iteration*, is the simplest, being easily captured by both syntactic and type-based termination checking. An iterative function definition may make recursive calls *only* on the immediate predecessors of the function's input; furthermore, these predecessors may *only* be used as arguments to recursive calls.

Figure 2.6 shows an example of an iterative function, *minus*, that implements subtraction for natural numbers. In the implementation of *minus*, we define an auxiliary recursive function *m-minus* that subtracts its argument from m, the first

argument to *minus*. In the successor case, the revealed predecessor n' is used as the argument to the recursive call *m-minus* n'.

The code listing of the figure contains the first use of the μ operator of the surface language in this dissertation. We will see a formal description of this operator in Chapter 5; for now, it suffices to understand that $\mu\, f.\ t\ \{\ |\ p_1 \to t_1\ |\ \ldots\ |p_n \to t_n\ \}$ is an expression that defines a term inductively over an inhabitant t of a datatype. The case tree (occuring within the curly braces) is understood similarly to those used in σ-expressions (discussed in Section 2.1.2); unlike σ, however, within these case branches the recursive function being defined (e.g., *m-minus*) may be referenced.

2.1.3.1.1 Computing by Iteration

Like with σ expressions, to understand how μ expressions enable inductive definitions we need to know not only how to write programs with them, but also see how these programs compute. We show an example of this in Figure 2.7, which gives some of the computation steps for subtracting the quantity 1 (*succ zero*) from itself using *minus*.

After replacing *minus* with its definition in line 1 and applying the resulting lambda abstraction to its two arguments in lines 2 and 3, the redex we are left with is a μ expression whose scrutinee is a constructor form, *succ zero*. The entire μ expression is replaced with the body of the successor branch, with the pattern variable n' replaced with *zero*. Additionally, all occurrences of the recursive function *m-minus* are replaced by a lambda abstraction whose body is a μ expression with the same case tree we started with — this is the mechanism by which recursion is supported.

$$
\begin{array}{lll}
0. & & \underline{\mathit{minus}}\ (\mathit{succ}\ \mathit{zero})\ (\mathit{succ}\ \mathit{zero}) \\
1. & =_{\mathbf{df}} & \underline{(\lambda\, m.\, \lambda\, n.\, \mu\ \mathit{m\text{-}minus}.\ n\ \{\ |\mathit{zero} \to m \mid \mathit{succ}\ n' \to \mathit{pred}\ (\mathit{m\text{-}minus}\ n')\ \})\ \underline{(\mathit{succ}\ \mathit{zero})}}\ (\mathit{succ}\ \mathit{zero}) \\
2. & \dashrightarrow & \underline{(\lambda\, n.\, \mu\ \mathit{m\text{-}minus}.\ n\ \{\ |\mathit{zero} \to \mathit{succ}\ \mathit{zero} \mid \mathit{succ}\ n' \to \mathit{pred}\ (\mathit{m\text{-}minus}\ n')\ \})\ \underline{(\mathit{succ}\ \mathit{zero})}} \\
3. & \dashrightarrow & \underline{\mu\ \mathit{m\text{-}minus}.\ (\mathit{succ}\ \mathit{zero})\ \{\ |\mathit{zero} \to \mathit{succ}\ \mathit{zero} \mid \mathit{succ}\ n' \to \mathit{pred}\ (\mathit{m\text{-}minus}\ n')\}} \\
4. & \dashrightarrow & \mathit{pred}\ (\underline{(\lambda\, n.\, \mu\ \mathit{m\text{-}minus}.\ n\ \{|\mathit{zero} \to \mathit{succ}\ \mathit{zero} \mid \mathit{succ}\ n' \to \mathit{pred}\ (\mathit{m\text{-}minus}\ n')\})\ \underline{\mathit{zero}}}) \\
5. & \dashrightarrow & \mathit{pred}\ (\underline{\mu\ \mathit{m\text{-}minus}.\ \mathit{zero}\ \{|\mathit{zero} \to \mathit{succ}\ \mathit{zero} \mid \mathit{succ}\ n' \to \mathit{pred}\ (\mathit{m\text{-}minus}\ n')\}}) \\
6. & \dashrightarrow & \mathit{pred}\ (\underline{\mathit{succ}\ \mathit{zero}})
\end{array}
$$

Figure 2.7: Example computation of *minus*

This brings us to line 4. After applying the lambda abstraction to its argument *zero* to obtain line 5, we again have a redex in the form of a μ expression applied to a constructor form (this time, *zero*). We now take the branch for *zero*, whose body is simply *succ zero*.

2.1.3.1.2 Proving by Iteration

When we make iteration dependent, we obtain an induction principle in which the induction hypothesis holds for the immediate predecessors. We show an example of a proof in this style written in Cedille in Figure 2.8, where the term *minusZeroLeft* is a proof that *minus zero n* is equal to *zero* for all n.

Like the example proof by cases with σ that we saw in Section 2.1.2.2, within each case body of the μ expression the type we are expected to inhabit has all occurrences of the scrutinee, n, replaced by the constructor form for that branch. So, in the *zero* case, the expected return type is $\{\mathit{minus}\ \mathit{zero}\ \mathit{zero} \simeq \mathit{zero}\}$, which computes to the reflexive equation $\{\mathit{zero} \simeq \mathit{zero}\}$. In the successor case, the goal type

```
minusZeroLeft : Π n: Nat. { minus zero n ≃ zero }
= λ n. μ minusZeroLeft. n {
  | zero → refl zero
  | succ n' →
    subst ·(λ x: Nat. { pred x ≃ zero }) (minus zero n') zero
      (minusZeroLeft n') (refl zero)
  } .

{- Pseudocode
minusZeroLeft : Π n: Nat. { minus zero n ≃ zero }
minusZeroLeft zero = refl zero
minusZeroLeft (succ n') =
  subst ·(λ x: Nat. { pred x ≃ zero}) (minus zero n') zero
    (minusZeroLeft n') (refl zero)
-}
```

Figure 2.8: Proof that subtracting anything from 0 is 0

is $\{minus\ zero\ (succ\ n') \simeq zero\}$, which is *convertible* with $\{pred\ (minus\ zero\ n') \simeq zero\}$ (both expressions can be reduced by computation to the same term). The induction hypothesis *minusZeroLeft* n' gives us a proof that *minus zero* n' is equal to *zero*, so we use *subst* to substitute the former for the latter in the left-hand side of the equation. The last argument to *subst* is a proof of expected type after this rewriting has been performed — $\{pred\ zero \simeq zero\}$. This computes to the reflexive equation $\{zero \simeq zero\}$, which we prove with *refl zero*.

2.1.3.2 Primitive Recursion

An immediate improvement to the iteration scheme is to allow direct predecessors to play roles in recursive definitions other than as the argument to recursive calls. This relaxation is often simply called *recursion*, but as we are interested in many forms of recursion in this disseration, to avoid confusion we will refer to it as

```
add : Nat → Nat → Nat
= λ m. μ add. m {
  | zero → λ n. n
  | succ m' → λ n. succ (add m' n)
  } .

sumFrom : Nat → Nat
= λ n. μ sumFrom. m {
  | zero → zero
  | succ n' → succ (add n' (sumFrom n'))
  } .

{- Pseudocode
add : Nat → Nat → Nat
add zero n = n
add (succ m') n = succ (add m' n)

sumFrom : Nat → Nat
sumFrom zero = zero
sumFrom (succ n') = succ (add n' (sumFrom n'))
-}
```

Figure 2.9: Sum of numbers from 0 to n (primitive recursion)

primitive recursion due to its historical use in defining the class of primitive recursive functions [51]. Our adopted terminology is not fully satisfactory, since in the presence of higher-order functions this scheme can be used to define more than just this class of functions.

Remark 2.3. For the remainder of this section, our examples of recursion schemes will involve only program construction. The μ operator supports both primitive recursion and course-of-values iteration, and computes as illustrated in Section 2.1.3.1.1.

Figure 2.9 shows an example use of the primitive recursion scheme to define *sumFrom*, which computes the sum of the natural numbers from 0 to n for any $n : Nat$. In the successor case of its definition, the predecessor n' is used *both* as the

argument to a recursive call to *sumFrom* and as the first argument to *add*.

Like with the iteration scheme, syntactic termination checkers can easily support the primitive recursion scheme since they place restrictions on how recursive calls are made, not on other uses of predecessors. However, this can be a somewhat nuanced point in the design of type-based termination checking, as predecessors may have types subtly different from the "real" datatype. We postpone discussion of this to Section 2.2.

Remark 2.4. The code listing of *sumFrom* in the figure is not a well-typed surface language program, as it is missing a necessary type coercion to make n' a valid first argument to *add*. We will discuss this issue in Section 2.2.4.

2.1.3.3 Course-Of-Values Iteration

Another improvement to the iteration scheme is to relax the restriction on recursive calls to *arbitrary* predecessors of the input, not just the immediate predecessors. When we make only this relaxation, we call the scheme *course-of-values iteration*; when we combine this with the generalization that yields primitive recursion, we call it *course-of-values recursion*.

The Fibonnaci Sequence and Nested Pattern Matching

Course-of-values iteration is not amenable to analysis by syntactic termination checking, but *is* amenable to type-based termination checking. To see why, it is useful to identify the subset of the course-of-values recursion scheme that mature syntactic termination checkers usually *do* support: making recursive calls on predecessors of a *statically computed* depth through nested pattern matching.

Figure 2.10 shows a (naive) implementation of the function *fib* which computes

```
fib : Nat → Nat
= λ n. μ fib. n {
  | zero → succ zero
  | succ n' →
    σ n' {
    | zero → succ zero
    | succ n'' → add (fib n') (fib n'')
    }
  }

{- Pseudocode
fib : Nat → Nat
fib zero = succ zero
fib (succ zero) = succ zero
fib (succ n'@(succ n'')) = add (fib n') (fib n'')
-}
```

Figure 2.10: Fibonacci (course-of-values iteration, static)

the nth entry of the Fibonacci sequence. Function *fib* is an example of course-of-values iteration on predecessors of the input at a static maximum depth. In the case where the argument is greater than one, two recursive calls are made: one each for the predecessor, n', and *its* predecessor, n'' (for the pseudocode in the listing, we use an @-pattern to bind the depth-1 predecessor of the argument as n'). The maximum depth that *fib* inspects its argument in a single step is evident by the syntax of its definition, as it arises from nested pattern matching on that argument. This makes it amenable to analysis by syntactic termination checkers.

Euclidean Division and Full Course-of-Values Iteration

In contrast to this, Figure 2.11 shows an example in which recursive calls are made on *dynamically* computed predecessors of the argument. In the figure,

```
lt : Nat → Nat → Bool
= λ m. λ n.
  σ (minus (succ m) n) {
  | zero → true
  | succ l → false
  } .

div : Nat → Nat → Nat
= λ m. λ n. μ div-n. m {
  | zero → zero
  | succ m' →
    if (lt (succ m') n) zero
      (succ (div-n (minus m' (pred n))))
  } .

{- Pseudocode
lt : Nat → Nat → Bool
lt m n = sigma (minus (succ m) n)
  where
  sigma : Nat → Bool
  sigma zero = true
  sigma (succ l) = false

div : Nat → Nat → Nat
div m n = div-n m
  where
  div-n : Nat → Nat
  div-n zero = zero
  div-n (succ m') =
    if (lt (succ m') n) true
      (succ (div-n (minus m' (pred n))))
-}
```

Figure 2.11: Division (course-of-values recursion, dynamic)

div implements natural number division through repeated subtraction of the divisor. To simplify the example, division by zero is handled by (eventually) returning zero. Proceeding by recursion on the dividend m, in the case that the argument to the recursive function *div-n* is greater than or equal to the divisor, we compute the difference between its predecessor, m', and the predecessor of the divisor and make a recursive call on the result. Thus, the argument to the recursive call to *div-n* is a *dynamically* computed predecessor, as the depth at which we access a predecessor depends on the value of the divisor n.

Unlike the situation for *fib*, in *div* it is not syntactically apparent that the recursive call respects the well-founded ordering of *Nat*: the expression we are making a recursive call on, *minus m* (*pred n*), is not a pattern variable (or an application spine whose head is a pattern variable). That *minus* only ever returns values no greater than its first argument is a *semantic* property. Type-based termination checking allows us to reflect this property into the type system, and so the task of the programmer is to implement a version of *minus* which makes the type of the result it returns suitable for use as an argument to the recursive call.

Remark 2.5. As with *sumFrom* from Figure 2.9, the implementation of *div* is not a well-typed surface language program due to the omission of type coercions; see Remark 2.4.

2.2 Recursion Schemes à la Mendler

In the previous section, we looked at examples of inductive definitions for which we desire surface language support, informally gesturing towards the class of recursion scheme they are meant to represent. In this section, we give formal descriptions of

these schemes, which form the basis of type-based termination checking in Cedille. This formalization comes in the form of combinators for structured recursion, which are presented with laws for typing, computation, and extensionality principles.

For the vast majority of literature on terminating structured recursion combinators, each combinators can be placed in one of two "heritages." The first and most widely known heritage is *Squiggol-style* combinators, as popularized by Bird and de Moor [13]. These combinators arise from the semantics of datatypes in category theory as initial algebras. The second heritage is *Mendler-style* combinators, as popularized by Uustalu and Vene [91], which arise from an alternative (and isomorphic) categorical semantics of inductive types as initial Mendler algebras [90]. Cedille's method of type-based termination checking follows this second heritage.

Squiggol style

Throughout this section, we discuss the Mendler style of coding recursion at length, and in doing so make frequent comparisons to the Squiggol style. For readers who are not familiar with the Squiggol style of coding recursion (or perhaps just unfamiliar with the term itself), we discuss a brief example before proceeding.

Example 2.6 (List *foldr*)**.** *Figure 2.12 shows the definition of the Squiggol-style recursive combinator foldr for lists in Agda-like pseudocode. Type parameter A is for the type of the elements of the list, and X is for the type of the result we wish to compute. The first two term arguents, of type X and $A \to X \to X$, are used respectively to specify how to handle the cases for constructors nil and cons.*

In foldr, parameter $c : A \to X \to X$ is given as its second argument this recursively computed result. When using Squiggol-style combinators such as foldl, we think in terms of the results *of recursive calls on predecessors, rather than thinking about the handle for recursive calls separately.*

To illustrate this further, consider the two equivalent implementations of a function to sum the entries of a list of numbers below:

```
sum1 : List ·Nat → Nat
```

```
data List (A: ⋆) : ⋆
= nil : List
| cons : A → List → List .

foldr : ∀ A: ⋆. ∀ X: ⋆. X → (A → X → X) → List ·A → X
foldr n c nil = n
foldr n c (cons h t) = c h (foldlr n c t)
```

Figure 2.12: List *foldl*

```
sum1 = foldr zero add

sum2 : List ·Nat → Nat
sum2 nil = zero
sum2 (c h t) = add h (sum2 t)
```

In sum2, the recursive call to compute the sum of the tail of the list is explicit, whereas in sum1, foldr takes care of this for us.

2.2.0.1 Why Code Recursion à la Mendler?

Compared to the Squiggol style, Mendler-style combinators are arguably more pleasant for programmers to work with. Mendler-style combinators compute more similarly to the general fixedpoint combinator that underpins general recursion, to which programmers of general-purpose functional languages are accustomed, than do their Squiggol-style counterparts. Additionally, in the Squiggol style more advanced forms of recursion (such as course-of-values iteration) require the explicit introduction of intermediate structures, whereas in the Mendler style these structures are implicit (see Uustalu and Vene [91], Sections 3 and 4). Finally, Mendler-style combinators generalize to mixed-variant datatypes (those with recursive occurrences occurring both covariantly and contravariantly) without risk to logical consistency or termination [4, 90]. Though this last feature is beyond the scope of this dissertation, in the view

of the author it is desirable that this avenue of future work is not foreclosed to Cedille.

2.2.1 Datatypes, Generically

Before we proceed with our discussion of recursive combinators, we review some necessary background on the semantics of inductive datatypes. We draw on the categorical semantics of datatypes as initial algebras [58, 42], however, no prior knowledge of category theory is assumed of the reader.

Inductive datatypes can be separated into two components. The first of these is the *signature*, which describes the non-recursive structure particular to a given datatype. We will show what this looks like for the particular example of *Nat*, but before this we need to introduce *coproducts* and the *unitary type*.

Definition 2.7 (*Unit*)**.** The type *Unit* has a single inhabitant, *unit*. In the surface language syntax, we can declare it as a nonrecursive datatype.

```
data Unit : ⋆ = unit : Unit .
```

We think of *Unit* as a type for "uninteresting values". When used to express signatures of inductive datatype, its role is to be the type of the argument for a constructor like *zero* which, in the surface language, would ordinarily be declared to take no arguments. This useage is justified by a natural type isomorphism $T \cong Unit \to T$ (for all T).

Definition 2.8 (*Sum*)**.** For all types T_1 and T_2, the type $Sum \cdot T_1 \cdot T_2$ is their disjoint union (or *coproduct*), with $in_l : T_1 \to Sum \cdot T_1 \cdot T_2$ and $in_r : T_2 \to Sum \cdot T_1 \cdot T_2$ giving the left and right injections, respectively. The center dot denotes type application where the argument is a type or type constructor. In the surface language syntax, we can declare *Sum* as a nonrecursive, parameterized datatype.

```
data Sum (T1: ⋆) (T2: ⋆) : ⋆
= inl : T1 → Sum
| inr : T2 → Sum .
```

Along with the constructors, *Sum* comes also with an *eliminator*, which we will call *elimSum*, which can be seen as a generalization of *if-then-else* conditional for Booleans as it chooses one of two branches to execute, depending on which injection was used to construct the inhabitant of the coproduct. In the surface language syntax, we can define *elimSum* as follows.

```
elimSum
: ∀ T1: ⋆. ∀ T2: ⋆. Sum ·T1 ·T2 → ∀ X: ⋆. (T1 → X) → (T2 → X) → X
= Λ T1. Λ T2. λ x. Λ X. λ l. λ r.
  σ x {
  | inl x1 → l x1
  | inr x2 → r x2
  } .
```

With the definitions of *Sum* and *Unit* understood, we can give the signature for natural numbers as $\lambda X{:}\star.\, \mathit{Sum} \cdot \mathit{Unit} \cdot X$, where X is a bound type variable used as a placeholder for the recursive occurrence of *Nat*.

The second component is a generic least fixedpoint operation μ for these signatures, which grants them their inductive structure. If $F : \star \to \star$ is positive, then we say μF is an inductive datatype with signature F. Note that we abuse notation, using μ for both datatype formation at the level of types and for recursive term definitions at the level of terms. The type μF is isomorphic to its one-step unfolding $F \cdot \mu F$, and this isomorphism is witnessed by a pair of functions $\mathit{in} : F \cdot \mu F \to \mu F$ and $\mathit{out} : \mu F \to F \cdot \mu F$.

We understand *in* to be the *generic* datatype constructor, since it introduces a value of type μF from an "F-collection" of predecessors; *out* exposes these predecessors, making it a generic datatype *destructor*. As an example, we can recover the usual constructors for *Nat* using *in*, coproducts, and the unitary type.

$$\begin{array}{rcl} \mathit{zero} & = & \mathit{in}\ (\mathit{in}_l\ \mathit{unit}) \\ \mathit{succ} & = & \lambda\, x{:}\mathit{Nat}.\, \mathit{in}\ (\mathit{in}_r\ x) \end{array}$$

Using *out* and *elimSum*, we can implement the predecessor function.

$$\mathit{pred} = \lambda\, n.\, \mathit{elimSum}\ (\mathit{out}\ n)\ (\lambda\, u.\, \mathit{zero})\ (\lambda\, n'.\, n')$$

2.2.2 Case Distinction Scheme

The first scheme we discuss, the case distinction scheme, does not feature any recursion. The typing, computation, and extension law are shown in Figure 2.13, which we now describe.

Reading Guide. For readers who are unfamiliar with the notation for inferences used in the figure, we give a brief primer. The notation $\Gamma \vdash t : T$ is a *typing judgment*, a three-place relation between a typing context Γ, a term t, and a type T, which we read as "under context Γ, term t has type T." The other judgment, $\Gamma \vdash t_1 \simeq t_2$, we read as "under context Γ, terms t_1 and t_2 are equal" (the precise notion of equality in use here is left under-specified for now). We denote appending a (fresh) variable x with type T to a context by $\Gamma, x\!:\!T$.

We affirm judgments through the use of inference rules, which are written with a horizontal line with the *premises* above the line and the singular *conclusion* below the line. The premises are sufficient conditions for showing the conclusion, and to affirm them one uses other applicable inference rules, forming a proof tree. Figure 2.13 and the others like it in this section should be understood as part of a full type theory that includes variables, functions, and polymorphism. We will see a complete listing of inference rules for a type theory in Chapter 3, when we overview the CDLE target language.

The typing rule for case distinction is given in the top left of the figure. It says that *case* takes two terms, a function t_1 of type $F \cdot \mu F \to T$ and a datatype element $t_2 : \mu F$. In the top right of the figure we have the computation rule, saying how case

$$\frac{\Gamma \vdash t_1 : F \cdot \mu F \to T \quad \Gamma \vdash t_2 : \mu F}{\Gamma \vdash \mathit{case}\ t_1\ t_2 : T} \qquad \mathit{case}\ t_1\ (\mathit{in}\ t_2') \dashrightarrow t_1\ t_2'$$

$$\frac{\Gamma \vdash t_1 : F \cdot \mu F \to T \quad \Gamma \vdash t_2 : \mu F \quad \Gamma \vdash h : \mu F \to T \quad \Gamma, x{:}F \cdot \mu F \vdash h\ (\mathit{in}\ x) \simeq t_1\ x}{\Gamma \vdash h\ t_2 \simeq \mathit{case}\ t_1\ t_2}$$

Figure 2.13: Case distinction: typing, computation, extension

computes when given a datatype value constructed with *in* — it just applies its first argument to the collection of predecessors.

The extensionality law for case distinction, located at the bottom of the figure, requires more explanation than the others. In Section 2.1.2, we mentioned that definitions by pattern matching are justified because the *only* (closed) values of a datatype are those built using the constructors. The extensionality law expresses exactly this idea: if a function $h : \mu F \to T$ obeys the same computation law as *case* t_1 (for some t_1), then for all $t_2 : \mu F$, the expressions $h\ t_2$ and *case* $t_1\ t_2$ are equal. That is to say, the behavior of the combinator for case distinction is *entirely* determined by its action on values constructed with *in*.

Example 2.9 (Predecessor)**.** *With case, we can implement the predecessor function pred for natural numbers as follows.*

```
pred : Nat → Nat
= λ n. case (λ x. elimSum x (λ u. n) (λ n'. n')) n .
```

In the above, we use the generic formulation of Nat as $\mu\ (\lambda X{:}\star.\ Sum \cdot Unit \cdot X)$*. The first argument to the combinator is a function of type* $Sum \cdot Unit \cdot Nat \to Nat$ *(so, the domain of this function is the unrolled recursive type). Eliminator elimSum enables us to specify how to handle the coproduct injections, corresponding to the two case branches in the definition of pred in Figure 2.3 for zero and succ. For the zero case, we ignore the "uninteresting" constructor argument* $u : Unit$ *and just return* n*; for the succ case, we return the bound predecessor* n'*.*

$$\frac{\Gamma \vdash t_1 : \forall\, R{:}\star.\, (R \to T) \to F \cdot R \to T \quad \Gamma \vdash t_2 : \mu F}{\Gamma \vdash \mathit{miter}\ t_1\ t_2 : T}$$

$$\mathit{miter}\ t_1\ (\mathit{in}\ t_2') \dashrightarrow t_1 \cdot \mu F\ (\mathit{miter}\ t_1)\ t_2'$$

$$\frac{\Gamma \vdash t_1 : \forall\, R{:}\star.\, (R \to T) \to F \cdot R \to T \quad \Gamma \vdash t_2 : \mu F \quad \Gamma \vdash h : \mu F \to T \quad \Gamma, x{:}F \cdot \mu F \vdash h\ (\mathit{in}\ x) \simeq t_1 \cdot \mu F\ h\ x}{\Gamma \vdash h\ t_2 \simeq \mathit{miter}\ t_1\ t_2}$$

Figure 2.14: Iteration: typing, computation, extension

2.2.3 Mendler-Style Iteration

Figure 2.14 gives the typing, computation, and extensionality laws for *Mendler-style iteration*. This scheme, particularly its extension to course-of-values recursion (Section 2.2.6), is the foundation on which Cedille implements its type-based termination checking. Combinator *miter* introduces the idea of using *parametric polymorphism* to ensure termination.

Remark 2.10. Though the typing rules for recursive combinators that we consider in this chapter show the combinators applied to a fixed set of arguments, in this semi-formal treatment we permit them to be partially applied, such as in the computation rule above, to de-clutter our presentation.

2.2.3.1 Typing

Like the case distinction scheme before it, the combinator for Mendler iteration takes two arguments. The first of these, t_1, gives the action to take for a single step of recursion, and the second, t_2, is an inhabitant of the datatype on which we wish to operate. We further unpack the type of t_1.

1. The type variable $R{:}\star$ serves as a "stand-in" for the datatype μF, in particular for the subset of values on which it is safe to make recursive calls. As type

polymorphism is parameteric in Cedille, there is no way for t_1 to inspect this type argument to determine whether it was instantiated with μF; this is a "theorem for free" in the sense of Wadler [92]. Instead, this type argument is opaque: the only values of this type, as well as the only allowable operations on these values, are those given as part of an argument to t_1.

2. The function argument of type $R \to T$ is a handle for making recursive calls. Its domain is the type R, so we can understand this handle for recursion as being restricted to only those values smaller than t_2 (note that by typing, t_2 is not a legal argument to this handle).

3. The argument of type $F \cdot R$ is an F-collection of predecessors of type R, revealed by pattern matching. By parametric polymorphism, t_1 can do nothing with these predecessors other than apply to them the handle for recursion.

Example 2.11 (Addition)**.** *We can use combinator miter to implement addition for natural numbers; c.f. Figure 2.9.*

```
add : Nat → Nat → Nat
= λ m.
  miter
    (Λ R. λ add. λ x.
     elimSum x (λ u. λ n. n) (λ m'. λ n. succ (add m' n)))
    m .
```

In the first argument to miter we have bound variables $R : \star$, $add : R \to Nat \to Nat$, and $x : Sum \cdot Unit \cdot R$. We inspect x with elimSum, and in the successor case (the third argument to elimSum) the recursive call is add m' n, and we return the successor of the result. Observe that add m n is not type correct, nor is add (succ m') n; allowing either of these would result in nontermination.

2.2.3.2 Computation

The computation law of *miter* expresses how the combinator acts over values built with constructor *in*. Because of the polymorphic typing of t_1, it can be a little

trickier to understand than the computation law for case distinction. To the right of the arrow, we instantiate the type paramater R of t_1 with μF, the datatype. This means that for the remaining two arguments, we are obligated to provide a term of type $\mu F \to T$ and one of type $F \cdot \mu F$. For the first of these, we use the combinator again with t_1 itself as the handle for recursive calls, and for the second we provide the collection of predecessors $t_2' : F \cdot \mu F$.

Unlike the Squiggol combinators, Mendler combinators for recursion have the pleasant property that they compute similarly to the general recursion combinator *fix* (recall that *fix* f $\dashrightarrow$ f (*fix* f)). Aside from the typing, another important difference between *miter* and *fix* is that reduction of the former only occurs in the presence of (that is to say, *is guarded by*) the constructor *in*, ensuring that the second argument to t_1 consists of the predecessors of the given datatype value, making them safe for recursive calls.

2.2.3.3 Extension

Finally, we come to the extensionality law for *miter*. Like the extensionality law for case distinction, extensionality for *miter* means that if any $h : \mu F \to T$ behaves like *miter* t_1 (for some t_1) for those values constructed with *in*, then h behaves like *miter* t_1 on all values of the datatype. Again, we may put this another way: the behavior of *miter* t_1 (for all t_1) is completely determined by its action over the constructor *in*.

2.2.4 Mendler-Style Primitive Recursion

We transitioned from iteration to primitive recursion in Section 2.1.3.2 by allowing predecessors to serve in roles other than arguments to recursion, with the example *sumFrom* showing the predecessor of a natural number being used as the argument to the previously defined function *add*. Examining the typing law for *miter*, we see that the opaque type variable R prevents us from giving a similar definition of *sumFrom* with the combinator, as the predecessors are given to us only at an opaque type. To support primitive recursion, we need a way to "reveal" that the given predecessors also have type μF.

Of course, in the informal explanation we have been relying on to build intuition for coding recursion à la Mendler, it is important that the type variable R *not* be identifiable with the datatype μF for terms playing the role of t_1 in the combinator laws. The trick is to observe that *only going from* μF *to* R *is unsound*, as there is no problem in using the predecessors to construct a new, larger datatype expression, so long as we are unable to make recursive calls upon it. We can accomplish this by equipping t_1 with an additional argument for coercing terms of type R to type μF.

The result of this, shown in Figure 2.15, is the *Mendler-style primitive recursion scheme*. In the typing rule, t_1 takes an additional argument of type $R \to \mu F$, serving as the type coercion. In the computation rule, this argument is instantiated to the identity function (type argument R is instantiated to μF, so our obligation is a function of type $\mu F \to \mu F$ and thus our use of the identity function is well-typed). We pass over the extensionality law without further comment, as its reading for *mrec*

$$\frac{\Gamma \vdash t_1 : \forall\, R{:}\star.\, (R \to T) \to (R \to \mu F) \to F \cdot R \to T \quad t_2 : \mu F}{\Gamma \vdash \mathit{mrec}\ t_1\ t_2 : T}$$

$$\mathit{mrec}\ t_1\ (\mathit{in}\ t_2') \dashrightarrow t_1 \cdot \mu F\ (\mathit{mrec}\ t_1)\ (\lambda\, x.\, x)\ t_2'$$

$$\frac{\Gamma \vdash t_1 : \forall\, R{:}\star.\, (R \to T) \to (R \to \mu F) \to F \cdot R \to T \quad \Gamma \vdash t_2 : \mu F \quad \Gamma \vdash h : \mu F \to T \quad \Gamma, x{:}F \cdot \mu F \vdash h\ (\mathit{in}\ x) \simeq t_1 \cdot \mu F\ h\ (\lambda\, x.\, x)\ x}{\Gamma \vdash h\ t_2 \simeq \mathit{mrec}\ t_1\ t_2}$$

Figure 2.15: Primitive Recursion: typing, computation, extension

is essentially the same as for the other schemes discussed so far.

Example 2.12 (Sum From)**.** *With combinator mrec, we implement the function sumFrom in a way that's faithful to the earlier presentation in Figure 2.9.*

```
sumFrom : Nat → Nat
= λ n.
  mrec (Λ R. λ sumFrom. λ c. λ x.
        elimSum x (λ u. zero) (λ n'. succ (add (c n') (sumFrom n'))))
    n .
```

Aside from the superficial differences between the two implementations (use of the combinator mrec instead of the μ operator, and Nat implemented generically instead of being a declared datatype), there is one discrepancy requiring further explanation: the use of the bound variable $c : R \to Nat$ to coerce the type of predecessor $n' : R$ so that it can be used as the first argument to add. There is a corresponding type coercion in the surface language that was omitted from the successor case of sumFrom in Figure 2.9 (see Remark 2.4). Restoring the surface language coercion, the body of the successor case looks as follows.

```
succ (add (to/Nat -isType/sumFrom n') (sumFrom n'))
```

We will explain this syntax in greater detail in Chapter 5.

2.2.5 Mendler-Style Course-Of-Values Iteration

The penultimate recursion scheme we consider is *Mendler-style course-of-values iteration*. The laws for the combinator *mcov* for this scheme are shown in Figure 2.16. In Section 2.1.3.3, we remarked that to support this recursion scheme

$$\frac{\Gamma \vdash t_1 : \forall\, R{:}\star.\, (R \to T) \to (R \to F \cdot R) \to F \cdot R \to T \quad t_2 : \mu F}{\Gamma \vdash \mathit{mcov}\ t_1\ t_2 : T}$$

$$\mathit{mcov}\ t_1\ (\mathit{in}\ t_2') \dashrightarrow t_1 \cdot \mu F\ (\mathit{mcov}\ t_1)\ \mathit{out}\ t_2'$$

$$\frac{\Gamma \vdash t_1 : \forall\, R{:}\star.\, (R \to T) \to (R \to F \cdot R) \to F \cdot R \to T \quad \Gamma \vdash t_2 : \mu F \quad \Gamma \vdash h : \mu F \to T \quad \Gamma, x{:}F \cdot \mu F \vdash h\ (\mathit{in}\ x) \simeq t_1 \cdot \mu F\ h\ \mathit{out}\ x}{\Gamma \vdash h\ t_2 \simeq \mathit{mcov}\ t_1\ t_2}$$

Figure 2.16: Course-of-Values Iteration: typing, computation, extension

in the surface language we needed to be able to make recursive calls on *arbitrary* predecessors of the given datatype value, not just the immediate ones. In terms of coding recursion á la Mendler, this means we need to make terms playing the role of t_1 in the typing law accept as an additional argument a version of the destructor $\mathit{out} : \mu F \to F \cdot \mu F$, one that preserves the opaque type argument in the revealed predecessors. That is to say, the combinator needs to take an argument of type $R \to F \cdot R$ that is instantiated to *out* in the computation law.

Example 2.13 (Fibonacci)**.** *We show how to use the mcov combinator to implement the Fibonacci function from Figure 2.10. Recall that this example demonstrates the ability to make recursive calls on predecessors of a* static *depth using course-of-values iteration. We postpone the example of division (which requires course-of-values* recursion*) to Section 2.2.6.*

```
fib : Nat → Nat
= λ n.
  mcov (Λ R. λ fib. λ o. λ x.
        elimSum x (λ u. succ zero)
          (λ n'. elimSum (o n') (λ u. succ zero)
                   (λ n''. add (fib n') (fib n'')))) n .
```

Kicking off the recursive definition with mcov, the outermost elimSum acts on $x : Sum \cdot Unit \cdot R$. In the successor branch (the third argument to elimSum), the predecessor is bound as $n' : R$, and we analyze this further with another use of elimSum on the expression $o\ n'$, which has type $Sum \cdot Unit \cdot R$. In the successor branch of this inner case expression, we now have the predecessor of n' as $n'' : R$, and we add the result of two recursive calls, one each on n' and n''.

$$\frac{\Gamma \vdash t_2 : \mu F \qquad \Gamma \vdash t_1 : \forall\, R{:}\star.\, (R \to T) \to (R \to \mu F) \to (R \to F \cdot R) \to F \cdot R \to T}{\Gamma \vdash \mathit{mcovr}\ t_1\ t_2 : T}$$

$$\mathit{mcovr}\ t_1\ (\mathit{in}\ t_2') \dashrightarrow t_1 \cdot \mu F\ (\mathit{mcovr}\ t_1)\ (\lambda\, x.\, x)\ \mathit{out}\ t_2'$$

$$\frac{\Gamma \vdash t_1 : \forall\, R{:}\star.\, (R \to T) \to (R \to \mu F) \to (R \to F \cdot R) \to F \cdot R \to T \quad \Gamma \vdash t_2 : \mu F \qquad \Gamma \vdash h : \mu F \to T \quad \Gamma, x{:}F \cdot \mu F \vdash h\ (\mathit{in}\ x) \simeq t_1 \cdot \mu F\ h\ (\lambda\, x.\, x)\ \mathit{out}\ x}{\Gamma \vdash h\ t_2 \simeq \mathit{mcovr}\ t_1\ t_2}$$

Figure 2.17: Course-of-values recursion: typing, computation, extension

2.2.6 Mendler-Style Course-Of-Values Recursion

For completeness, we conclude our discussion of combinators for structured recursion by presenting the full set of laws for the derived scheme *course-of-values recursion*. This is shown in Figure 2.17 as the typing, computation, and extensionality laws for combinator *mcovr*. This combinator, which is the foundation upon which we build Cedille's type-based termination checker, is the result of combining the Mendler-style combinators for primitive recursion and course-of-values iteration: two additional arguments for the two capabilities we provide the programmer.

Example 2.14 (Division)**.** *With the combinator mcovr, we can give an implementation for division for natural numbers in a style faithful to the listing for div in Figure 2.11. This is shown in Figure 2.18. This is the most complex example we have seen yet, so we present it step by step.*

- *Our first step is generalizing the predecessor function to operate on the opaque types introduced through the use of mcovr. This is predCoV in the figure. In place of the destructor out : Nat → Sum · Unit · Nat for datatypes, we require the caller to provide a type R together with function of type R → Sum · Unit · R — an "abstract out". The argument from which we are to compute the predecessor, n, must have the same type.*
- *To show that predCoV is a proper generalization of pred, we next implement the*

```
predCoV : ∀ R: ⋆. (R → Sum ·Unit ·R) → R → R
= Λ R. λ o. λ n. elimSum (o n) (λ u. n) (λ n'. n') .

pred : Nat → Nat
= predCoV ·Nat out .

minusCoV : ∀ R: ⋆. (R → Sum ·Unit ·R) → R → Nat → R
= Λ R. λ o. λ m. λ n.
  mcovr
    (Λ R'. λ m-minus. λ c. λ o. λ x.
     elimSum x (λ u. m) (λ n'. predCoV ·R o (m-minus n'))) n .

div : Nat → Nat → Nat
= λ m. λ n.
  mcovr
    (Λ R. λ div-n. λ c. λ o. λ x.
     elimSum x (λ u. zero)
       (λ m'. if (lt (succ (c m')) n) zero
                 (succ (div-n (minusCoV ·R o m' (pred n))))))
    m .
```

Figure 2.18: Division with *mcovr*

latter in terms of the former: the ordinary predecessor function is an instance of predCoV where the type is Nat and the destructor is out.

- *Next, in minusCoV we generalize subtraction by replacing the usual predecessor function with predCoV. This is where we compute a predecessor at a* dynamic *depth that will be suitable for recursion by some caller, because* the return type of *minusCoV* is the same as the type of the minuend.
- *Finally, in div we implement division by recursion over the dividend. Inspecting $x : Sum \cdot Unit \cdot R$ with elimSum, in the successor case we test whether the successor of m' is strictly less than the divisor, using the coercion $c : R \to Nat$ to cast m' to the appropriate type. If the test fails, we make a recursive call on $minusCoV \cdot R\ o\ m'\ (pred\ n) : R$, where $o : R \to Sum \cdot Unit \cdot R$ is the "abstract out" function.*

In Remark 2.5, we remarked that the code listing for div in Figure 2.11 was not type correct, as it omitted necessary typing annotations. In an idiomatic Cedille implementation of division with all needed typing annotations present, the recursive call would look like the following.

```
div-n (minusCoV -isType/div-n m' (pred n))
```

We will explain this syntax in greater detail in Chapter 5.

Remark 2.15. In Remark 2.1, we noted the somewhat peculiar definition of *pred* in Figure 2.2, in particular returning the original argument n when pattern matching reveals it to be *zero*, and not simply returning *zero* itself. The definition of *predCoV* in Figure 2.18 illustrates that the reason for this is *the typing:* n has type R, whereas *zero* does not. Note that in Cedille, it is possible in such situations to "retype" *zero* to type R using the knowledge that $n : R$ is equal to *zero*; such retypings are discussed in Chapter 3.

2.3 Chapter Conclusion

In this chapter, we discussed inductive type declarations and well-founded recursive functions in type theory, known collectively as *inductive definitions*, and showed examples of both in the surface language syntax of Cedille. Our goal was to identify the class of inductive definitions supported by the surface language. For inductive datatypes, we postponed to Chapter 4 the positivity criterion that determines the class of declarations supported in the surface language. For well-founded recursive functions, we formalized the non-dependent class of supported definitions

with a Mendler-style combinator for *course-of-values recursion*, postponing its generalization to dependent types (that is, a combinator for induction) to Section 3.4. Of course, at this stage the connection between this combinator and the surface language is still informal; we make it formal via elaborating type inference rules in Chapter 5.

The remainder of this section summarizes relevant work on inductive definitions in type theory.

Inductive Types

Inductive datatype declarations, in the form of algebraic datatypes, made their debut in the HOPE programming language [15] and have since become a popular language feature, appearing in modern languages such as Haskell, ML, Racket, and Scala, to name only a few. For dependent type theories, concurrent developments have seen user-defined inductive types realized as W-types [67], computed from codes in a closed universe of types [12, 28], axiomatized [95], and even derived from impredicative encodings [79]. Malcom [58] and Hagino [42] showed how datatype declarations could be interpreted into the categorical semantics of inductive types as initial algebras, placing them on firm semantic foundations.

Positivity

As noted in Section 2.1.2.3, Mendler [63] is credited with the observation that pattern matching over mixed-variant types leads to nontermination and inconsistency. This result can be seen as a variation on an observation by Morris [66] that recursive types without a positivity restriction can be used to construct a general fixedpoint combinator; both can be understood as a syntactic application of Scott's domain-

theoretic semantics of the untyped lambda calculus [74].

Interestingly, it is also the Mendler style of coding recursion that allows one to use mixed-variant types in total type theory by denying them the usual case distinction scheme. This result, established by Uustalu and Vene [90], is leveraged by Ahn and Sheard [5], who show how to safely define practical functions over mixed-variant types, such as a pretty printer for a higher-order abstract syntax (HOAS) representation of a toy language. Ahn later used the Mendler style of coding recursion as the basis of the Nax programming language [4], which also supports mixed-variant datatypes.

Positivity is usually presented as a syntactic criterion which involves the counting of arrows to the left of which a recursive occurrence occurs (see for example Geuvers [39], Section 3). We have intentially omitted discussion of syntactic characterizations of positivity, as we believe the *producer-consumer* narrative provides better intuition for the problem of mixed-variant datatypes. Moreover, this narrative more closely resembles Cedille's formal positivity requirement, a semantically motivated notion of *monotonicity*, than does the syntactic account.

Termination Checking

It is no coincidence that in Section 2.1.3 we discussed two kinds of termination checking regimes, syntactic and type-based, and in Section 2.2 we discussed two heritages of recursive combinator, Squiggol and Mendler. In their advocacy for coding recursion á la Mendler, Uustalu and Vene [91] linked syntactic criteria for termination with the equations of the Squiggol-style recursive definitions, referring to

it as the *conventional style*. Barthe et al. [11] explicitly credit Mendler [64] as inspiration for their approach to type-based termination using size-indexed types. More recent publications by Stump et al. [84] and Abreu et al. [3] have used the Mendler style of coding recursion to implement ever more complex recursive functions, such as those using nested recursion and divide-and-conquer strategies, using the terminology *recursion universe* to build intuition for programming against an expressive polymorphic interface.

Termination of recursive combinators can be established via elaboration to a total type theory. Matthes [60] undertook this endeavor for both Squiggol-style and Mendler-style combinators, showing how they can be realized in an extension of System F with positive recursive types, provided that the datatype signature can be shown to be covariant. We will have more to say on this in Chapter 4, particularly as we embark on our own Matthesian endeavor to elaborate monotonicity witnesses from datatype declarations.

CHAPTER 3
CALCULUS OF DEPENDENT LAMBDA ELIMINATIONS: THEORY AND META-THEORY

> But, eventually, you will of course have to have a language which is not given meaning by translating it into another language but has to be given meaning in some other way, and this is the language of the most primitive notions that you are dealing with, because that they are primitive means precisely that they cannot be defined in terms of any other notions.
>
> ---
>
> Per Martin-Löf
> *Truth of a Proposition, Evidence of a Judgement, Validity of a Proof*

Before we elaborate inductive definitions in Cedille to its target language, the *Calculus of Dependent Lambda Eliminations* (CDLE), we must first understand CDLE and in particular what features make it an appealing choice as target language. CDLE is a logically consistent constructive type theory that contains as a subsystem the impredicative and extrinsically typed *Calculus of Constructions* (CC) [25]. Its *raison d'être* is to be a kernel theory upon which ITPs with inductive types can be based. To that end, CDLE has two primary design goals.

- **Be expressive.** Every feature available in the ITP *must* be expressible in some fashion in the kernel theory, though the form it takes there may not be nearly as convenient to use. The contrapositive of this is that if a feature cannot be expressed in the kernel, it cannot be supported in the surface language.
- **Be compact.** As ITPs are themselves complex software artifacts, to facilitate trust in the proofs carried out within them they should satisfy the *de Bruijn criterion* [38]: proofs should be translated into the kernel theory so that a separate, easier to verify implementation of the kernel can check them.

Concerning the first of these goals, this dissertation is itself evidence of the expressiveness of CDLE; one can also look to the works of Firsov and Stump [36] on generically deriving induction for lambda encodings, Diehl et al [31] on deriving zero-cost program and proof reuse, Jenkins et al. [45] on deriving a simulation of large eliminations, and Jenkins and Stump [48] on deriving monotone recursive types. For the second, Cedille translates well-typed programs and proofs into a minimal specification of CDLE called *Cedille Core* [81], which has been implemented in ∼1K lines of Haskell code. It is for these reasons and more that CDLE has, at least in the view of the author, succeeded in achieving its design goals.

However, it must be understood that this success comes at the expense of other characteristics that one might find desirable of an ITP, such as a strong guarantee of termination or axiomatic neutrality. If these relative shortcomings can be tolerated, then CDLE provides an excellent setting for pragmatists concerned with using dependent types to express and reason about programs and with trusting the ITP within which such reasoning is carried out. We will also see later in this chapter that the causes for CDLE's qualified termination guarantee are directly responsible for one of its most powerful reasoning capabilities — the ability to internalize the typing judgement.

3.0.1 About This Chapter

Prospectus. We overview CDLE, the target language for the elaboration of Cedille's inductive definitions, with Sections 3.1.1 and 3.1.2 providing guidance on how to read a type theory presented as a collection of algorithmic inference rules and Sections 3.1.3

and 3.1.4 covering the type formation, introduction, elimination, and erasure rules of CDLE. As it is the target language that gives meaning (via translation) to the surface language within the framework of elaboration, we also overview the essential meta-theoretic properties of CDLE and discuss the implications these have for Cedille in Section 3.2. Finally, in Section 3.3 we introduce some constructions that are derivable within CDLE. These constructions are used in this chapter to state a CDLE-idiomatic induction principle for course-of-values recursion (Section 3.4) which will be used throughout the later chapters describing the process of elaboration.

Reading Guide. All readers, save those intimately familiar with CDLE, are **strongly recommended** to read the following portions of this chapter: Section 3.1 up to and including 3.1.2, which summarizes basic conventions for reading inference rules, meta-variable conventions for this dissertation, and the all-important notion of *erasure*; Section 3.2.2, which formally states CDLE's termination guarantee; and the entirety of Section 3.3, which introduces the derived constructs *Cast* and *View* and shows their use to state a CDLE-idiomatic course-of-values induction principle.

The reader may choose to skim the presentation of the typing constructs of CDLE in Sections 3.1.4, particularly implicit products, dependent intersections, and the equality type, referring back to these as needed when proceeding with the subsequent chapters of this dissertation.

Original Contributions. Sections 3.1 and 3.2 are adapted from Stump and Jenkins [83] and Jenkins and Stump [48], to which I contributed technical tweaks to the theory as well as the meta-theoretic results of subject reduction for type constructors and

kinds, judgement validity, and strong normalization for type constructors and kinds under a certain reduction relation (Propositions 3.25, 3.26, and 3.20).

The original formulation of the course-of-values induction scheme in Section 3.4 is due to Firsov et al. [35] (for which all credit lies with the first two authors, Firsov and Diehl). I have modified this scheme to make it more suitable as a target for elaborating surface language μ- and σ-expressions.

Finally, the derived constructs *Cast* and *View* were first published in Jenkins and Stump [48]. All credit lies with Stump for the original CDLE derivation for these; the axiomatic presentation and prose surrounding it are mine.

3.1 Typing Judgements, Conversion, and Erasure

Before immersing ourselves in the formalism of CDLE, it is helpful to understand its *philosophy*, in particular its ontology regarding types. The computational substrate of CDLE is (just) the *untyped* lambda calculus, with types playing the role of helping us to reason about the behavior of untyped terms. CDLE is therefore an *extrinsically typed* (aka *Curry-style*) theory, in which types are external to terms.

Traditionally, the externality of types in a Curry-style theory is immediately apparent from its formalization in a *type-assignment system*. As an example, consider the type assignment rule below for type constructor instantiation ("type constructor" refers to types and type-level functions).

$$\frac{\Gamma \vdash t : \forall\, X : \kappa.\, T \quad \Gamma \vdash T' : \kappa' \quad \kappa \cong \kappa'}{\Gamma \vdash t : T[T'/X]}$$

The challenge for a machine implementation of such a system is immediate: because of this rule, the typing judgment is not *syntax directed*; i.e., there is nothing about

the syntax of the term t itself, the subject of typing in the rule, that indicates how we should choose the type constructor T' for our instantiation.

To avoid this difficulty, CDLE uses a system of typing annotations for terms, similar in appearance to intrinsically typed languages. However, such annotations play no role whatsoever in computation. All annotations are erased, leaving behind only terms of the untyped lambda calculus. Using type constructor instantiation as an example again, the single type-assignment rule above would instead be presented in two parts: a type inference rule for annotated terms and an annotation erasure rule.

$$\frac{\Gamma \vdash t : \forall\, X : \kappa.\, T \quad \Gamma \vdash T' : \kappa' \quad \kappa \cong \kappa'}{\Gamma \vdash t \cdot T' : T[T'/X]} \qquad |t \cdot T'| = |t|$$

The notation for the erasure of some term t is $|t|$, and the erasure function is defined inductively over the syntax of type-annotated terms. Note that in this setup, we still speak of *assigning types to terms*. Understand, however, that the term to which we are assigning a type is **not** t, but $|t|$. Also keep in mind that the term-level operational semantics is **only** for untyped terms. In CDLE, this is just the untyped lambda calculus; when we present the operational semantics for μ- and σ-expressions in Cedille, it will be for untyped versions of these constructs.

3.1.1 Preliminaries

We begin with a brief summary of the terminology and conventions of notation used throughout this chapter. The reader may choose to skip ahead to Section 3.1.3, referring back to this section as needed.

Notation 3.1 (Meta-variables)**.**

- t, t', t_1, s etc denote terms.
- x, y, z, x', x_1 etc denote term variables.
- T, T', T_1, S etc denote type constructors. We also use F, G (and subscripted and primed variants of these) for datatype signatures, that is, type constructors of kind $\star \to \star$.
- X, Y, Z, R, X', X_1 etc denote type constructor variables.
- $\kappa, \kappa_1, \kappa'$ etc denote kinds.

Definition 3.2 (Type constructors, kinds)**.**

- By *type constructors*, we refer to all expressions at the type level. This includes the familiar types, like *Nat* and *Bool*, and expressions that take as arguments terms and types and return type constructors, such as $\lambda\, X : \star.\, X \to X$ or $\lambda\, x : \mathit{Nat}.\, \{x \simeq \mathit{zero}\}$.
- *Kinds* are to type constructors what types are to terms, i.e., kinds classify type constructors. For example, the kind of types is $\star$, and the kind of type constructors serving as predicates on *Nat* (such as $\lambda\, x : \mathit{Nat}.\, \{x \simeq \mathit{zero}\}$) is $\mathit{Nat} \to \star$.

Notation 3.3 (Free and declared variables)**.** The meta-language notations $FV(t)$, $FV(T)$, and $FV(\kappa)$ denote the set of free variables of term t, type constructor T, and kind κ, resp. $DV(\Gamma)$ denotes the set of declared variables of the context Γ, (that is, it returns an unordered set from an ordered context).

Notation 3.4 (Capture-avoiding substitution)**.** The capture-avoiding substitution of variable x for term t is written $[t/x]$, and the capture-avoiding substitution of type constructor variable X for type constructor T is written $[T/X]$. We sometimes need an additional type ascription to a term which we are substituting in for a variable. This is written $[t/x]^T$ and is an abbreviation for $[(\chi\ T\ \text{-}\ t)/x]$, where $\chi\ T\ \text{-}\ t$ is the annotation for ascribing the type T to term t (see Figure 3.7). We discuss why this is sometimes needed in Remark 3.9.

Remark 3.5 (Freshness of variables, α-equivalence classes of terms)*.* Whenever we append a variable to a typing context, written $\Gamma, X : \kappa$ for type variables and $\Gamma, x : T$ for term variables, we implicitly assume that variable is *fresh* with respect to the declared variables of Γ, i.e., $X \notin DV(\Gamma)$ and $x \notin DV(\Gamma)$. We adopt the common practice of working with α-equivalence classes of terms, meaning that we assume we may freely rename variables bound by abstractions (λ and λ) or quantifiers (Π, $\forall$, and ι) in terms, types, and kinds, so that they are fresh with respect to a given context. This is to avoid cluttering the presentation of inference rules with details about renaming and lookup of shadowed variables.

Notation 3.6 (Reduction and convertibility)**.** We use the dashed arrow $\dashrightarrow$ for call-by-name reduction of terms and $\dashrightarrow_\beta$ for full $\beta\eta$ reduction of terms. For type constructors and kinds, we use $\dashrightarrow$ for reduction to weak head normal form and $\dashrightarrow_\beta$ for term-free reduction; these are defined in Section 3.1.3. Since we have stratified

the language into term-, type-, and kind-level, no ambiguity should arise by the reuse of the symbols $\dashrightarrow$ and $\dashrightarrow_\beta$. We use a slash through the symbol to indicate that a term, type constructor, or kind is *normal* (does not reduce), e.g., $T \not\dashrightarrow$.

For a reduction relation R, we write R^+ for its transitive closure and R^* for its reflexive transitive closure. For untyped terms, we write $=_{\beta\eta}$ for $\beta\eta$-convertibility (the reflexive, symmetric, transitive closure of full $\beta\eta$-reduction).

For definitions of call-by-name reduction and full $\beta\eta$-reduction, see an introductory text on programming languages such as Stump's "Programming Language Foundations" [78] (Section 5.4) or Pierce's "Types and Programming Languages" [72] (Section 5.1).

3.1.2 Algorithmic Inference Rules

Throughout this chapter and the remainder of this dissertation, we will present and discuss *judgements* and *inference rules* for those judgements. For the benefit of readers who are unfamiliar with this convention for describing type theories, we will now explain how these should be understood. A judgement is a relation that is inductively defined by inference rules. We have seen already judgements for typing having the form $\Gamma \vdash t : T$ in Section 2.2. Recall that this is a three-part relation between a typing context Γ, a term t, and a type T, which we read as *"under context Γ, term t has type T"*. When we speak of an inference rule for a judgement, we mean that that inference rule has that judgement occurring in the *conclusion* (below the horizontal line); in order to use a particular rule to affirm a judgement, we must affirm all the judgements in its premises (given above the horizontal line) through the use of other inference rules. The judgements in the premises need not be the same as that of the conclusion.

Type constructor instantiation serves as a useful example.

$$\frac{\Gamma \vdash t : \forall\, X : \kappa.\, T \quad \Gamma \vdash T' : \kappa' \quad \kappa \cong \kappa'}{\Gamma \vdash t \cdot T' : T[T'/X]}$$

The above is an inference rule for the typing judgement $\Gamma \vdash t : T$, because this is the judgement mentioned in the conclusion. In the premises, however, there is a reference to a different judgement, the *kinding* judgement $\Gamma \vdash T : \kappa$. As kinding also makes reference to typing, these judgements should be considered *mutually inductively defined*.

For this dissertation, we are not concerned merely with giving a specification of a type theory, but describing an *elaboration semi-algorithm* — only a semi-algorithm, as type inference in CDLE is undecidable due to the presence of non-terminating terms (see Section 3.2.2). Even though we do no elaboration in this chapter, we shall introduce here the conventions for formulating inference rules algorithmically. Thus, what will follow in the remainder of this chapter is not just a presentation of CDLE, but in fact a semi-algorithm for its type inference. We now explain the properties that make the inference rules for a judgment algorithmic.

3.1.2.1 Moded Judgements

For each judgement, we decide which of the related parts are to be considered *inputs* and which are considered *outputs*. As an example, for the kinding judgement $\Gamma \vdash T : \kappa$, the context Γ and type constructor T are the inputs, and the kind κ is an output; as another, the judgement for well-formed kinds, $\Gamma \vdash \kappa$, has only inputs. In figures that introduce a judgement, we convey this reading with supserscript $+$ for outpus and $-$ for inputs, e.g., $\Gamma^- \vdash T^- : \kappa^+$ and $\Gamma^- \vdash \kappa^-$. This introduces a wrinkle

for the typing judgment $\Gamma \vdash t : T$, as it is desirable to have the type as an *input* sometimes, and other times as an *output*. We will return to this matter shortly.

3.1.2.2 Syntax-Directedness

Given a judgement that we wish to assert, which of its inference rules should we try to apply? If multiple rules apply, then any algorithm we might wish to implement for the type theory will have to perform some kind of search, possibly with backtracking. Furthermore, since the parts of the judgement considered to be outputs are not predetermined, if two different strategies produce two different sets of outputs, which should we choose?

We avoid these unfortunate outcomes if instead the type system *syntax directed*, meaning that the we know which rule must apply by the form (syntax) of the inputs. This means that *at most one rule* is applicable for any concrete instantiation of the inputs of the judgement. For example, there is only one applicable rule for typing polymorphic instantiation, $t \cdot T'$.

Remark 3.7. When we consider the elaboration judgments in Chapters 4.1 and 4, we will sometimes define these with rules that make use side conditions or premises asserting or negating that some syntactic property holds, such as syntactic equality of identifiers. Under the strictest definition of the term, judgments defined with such rules are not syntax-directed. We will however adopt a more liberal usage in this dissertation, and call a judgment syntax-directed when the applicable rule is uniquely determined by a syntactic criteria for the inputs of the judgment.

3.1.2.3 Mode-Correct Rules

Each inference rule must respect the modalities we have assigned to the components of judgements to which it refers, called *mode correctness* [71, 93]. The recipe for mode correctness is as follows.

1. **Assume** that the inputs of the judgement in the conclusion are available.
2. **Show** that the inputs of the judgement in the premise can be constructed.
3. **Assume** that the outputs of the judgement in the premise are given.

   If there are multiple premises, read them left to right, where we are allowed to use the outputs to construct the inputs of premises further to the right.
4. **Show** that the output of the judgement in the conclusion can be constructed.

In order to make the inference rules for typing mode-correct, we will need to break apart the judgement $\Gamma \vdash t : T$ into two judgements, depending on whether we wish to treat the type as an input or an output: one for *type synthesis*, $\Gamma^- \vdash t^- \in T^+$, and one for *type checking*, $\Gamma^- \vdash T^- \ni t^-$ This is called *bidirectional type inference*, and it is a well-known technique for formulating a type theory such that a type inference algorithm can be easily extracted from the presentation. See Dunfield and Krishnaswami [32] for a historical survey of the technique.

Example 3.8. *Consider the type* synthesis *rule for type constructor instantiation.*

$$\frac{\Gamma \vdash t \in \forall\, X : \kappa.\, T \quad \Gamma \vdash T' : \kappa' \quad \kappa \cong \kappa'}{\Gamma \vdash t \cdot T' \in T[T'/X]}$$

By moding, Γ, t, T' are given as inputs. We can therefore construct the inputs to the first two premises of the rule, obtaining $\forall\, X : \kappa.\, T$ and κ' as their outputs. This allows us to construct the inputs to the final premise (κ and κ') and the output to the conclusion ($T[T'/X]$).

Remark 3.9 (Annotated substitutions). In Notation 3.4, we introduced the notation $[t/x]^T$ that abbreviates $[\chi\ T\ \text{-}\ t/x]$. Now that we have seen the bidirectional typing judgements, we can explain why this is needed. Term variables may have their types *synthesized* (since they are declared in typing contexts in which their types are associated with them), but the terms t with which we are substituting them are often only assumed to be able to have their types *checked*. The type ascription enables us to ensure that a well-typed term is still well-typed after substitution, which we need to show that reduction preserves the kinding of type constructors (Theorem 3.25).

3.1.3 Kinds and Type Constructors

We now proceed with a detailed description of CDLE, beginning with its classifiers. There are two sorts of classifiers in CDLE: kinds κ classify type constructors, and types (type constructors of kind $\star$) classify terms. Figure 3.2 gives the inference rules for the judgement $\Gamma \vdash \kappa$ that kind κ is well-formed in context Γ, and for judgement $\Gamma \vdash T : \kappa$ that type constructor T is well-formed and has kind κ under Γ. The full grammar for CDLE kinds, type constructors, and terms is given in Figure 3.1.

Convertibility of kinds and type constructors is notated with $\cong$, and the inference rules that comprise type constructor convertibility are given in Figure 3.6 (the rules for kinds are omitted, as they consist entirely of congruence rules). In those rules, we reduce type constructors to weak head normal form with the reduction relation $\dashrightarrow$ (Figure 3.4), before checking convertibility with the auxiliary relation $\cong^{\mathrm{w}}$. When terms are encountered during this process, they are checked for $\beta\eta$-equivalence modulo erasure (see Figure 3.8 for the erasure rules). Figure 3.5 gives an expanded reduction relation $\dashrightarrow_\beta$ for type constructors and kinds that acts on everything except terms. It is for this relation, which subsumes reduction to weak head normal form, that we prove certain meta-theoretic properties of interest for CDLE's language of type constructors and kinds (see Section 3.2).

3.1.3.1 Subsystem CC

As mentioned earlier, CDLE has as a subsystem the impredicative, extrinsically typed *Calculus of Constructions* [25] (CC). We now briefly review this subsystem. The kind formers of CDLE are the same as those in CC: $\star$ classifies types (such

kinds	$\kappa, \kappa', \kappa_1, \ldots$	$::=$	
type classifier			$\star$
term quantification			$\mid \Pi\, x\!:\!T.\, \kappa$
type constructor quantification			$\mid \Pi\, X\!:\!\kappa_1.\, \kappa_2$
type constructors	$S, T, F, T', T_1, \ldots$	$::=$	
variable			X
polymorphic type			$\mid \forall\, X\!:\!\kappa.\, T$
implicit product type			$\mid \forall\, x\!:\!S.\, T$
product type			$\mid \Pi\, x\!:\!S.\, T$
term abstraction			$\mid \lambda\, x\!:\!S.\, T$
type constructor abstraction			$\mid \lambda\, X\!:\!\kappa.\, T$
term application			$\mid T\ t$
type constructor application			$\mid T \cdot T'$
dependent intersection type			$\mid \iota\, x\!:\!S.\, T$
equality type			$\mid \{t_1 \simeq t_2\}$
terms	$s, t, t', t_1, \ldots$	$::=$	
variable			x
term abstraction			$\mid \lambda\, x.\, t$
term application			$\mid t\ t_1$
polymorphic abstraction			$\mid \Lambda\, X.\, t$
polymorphic instantiation			$\mid t \cdot T$
erased term abstraction			$\mid \Lambda\, x.\, t$
erased term application			$\mid t\ \text{-}t_1$
intersection introduction			$\mid [t_1, t_2]$
intersection projection			$\mid t.1 \mid t.2$
reflexive equalities			$\mid \beta\{t'\}$
δ axiom			$\mid \delta\ \text{-}\ t$
equality substitution			$\mid \rho\ t\ @x\langle t_2\rangle.T'\ \text{-}\ t'$
φ axiom			$\mid \varphi\ t\ \text{-}\ t'\ \{t''\}$
type ascription			$\mid \chi\ T\ \text{-}\ t$
let expressions			$\mid [x : T_1 = t_1]\ \text{-}\ t_2$

Figure 3.1: Grammar of CDLE

$$\boxed{\Gamma^- \vdash \kappa^-}$$

$$\frac{}{\Gamma \vdash \star} \qquad \frac{\Gamma \vdash T : \star \quad \Gamma, x{:}T \vdash \kappa}{\Gamma \vdash \Pi\, x{:}T.\, \kappa} \qquad \frac{\Gamma \vdash \kappa' \quad \Gamma, X{:}\kappa' \vdash \kappa}{\Gamma \vdash \Pi\, X{:}\kappa'.\, \kappa}$$

$$\boxed{\Gamma^- \vdash T^- : \kappa^+}$$

$$\frac{(X : \kappa) \in \Gamma}{\Gamma \vdash X : \kappa} \qquad \frac{\Gamma \vdash \kappa \quad \Gamma, X{:}\kappa \vdash T : \star}{\Gamma \vdash \forall\, X{:}\kappa.\, T : \star}$$

$$\frac{\Gamma \vdash T : \star \quad \Gamma, x{:}T \vdash T' : \star}{\Gamma \vdash \forall\, x{:}T.\, T' : \star} \qquad \frac{\Gamma \vdash T : \star \quad \Gamma, x{:}T \vdash T' : \star}{\Gamma \vdash \Pi\, x{:}T.\, T' : \star}$$

$$\frac{\Gamma \vdash T : \star \quad \Gamma, x{:}T \vdash T' : \kappa}{\Gamma \vdash \lambda\, x{:}T.\, T' : \Pi\, x{:}T.\, \kappa} \qquad \frac{\Gamma \vdash \kappa \quad \Gamma, X{:}\kappa \vdash T' : \kappa'}{\Gamma \vdash \lambda\, X{:}\kappa.\, T' : \Pi\, X{:}\kappa.\, \kappa'}$$

$$\frac{\Gamma \vdash T : \Pi\, x{:}T'.\, \kappa \quad \Gamma \vdash T' \ni t}{\Gamma \vdash T\ t : \kappa[t/x]^{T'}} \qquad \frac{\Gamma \vdash T_1 : \Pi\, X{:}\kappa_2.\, \kappa_1 \quad \Gamma \vdash T_2 : \kappa_2' \quad \kappa_2 \cong \kappa_2'}{\Gamma \vdash T_1 \cdot T_2 : \kappa_1[T_2/X]}$$

$$\frac{\Gamma \vdash T : \star \quad \Gamma, x{:}T \vdash T' : \star}{\Gamma \vdash \iota\, x{:}T.\, T' : \star} \qquad \frac{FV(t\ t') \subseteq DV(\Gamma)}{\Gamma \vdash \{t \simeq t'\} : \star}$$

Figure 3.2: Kind formation and kinding of type constructors in CDLE

$$\boxed{\vdash \Gamma^-}$$

$$\frac{}{\vdash \emptyset} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash T : \star}{\vdash \Gamma, x{:}T} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash \kappa}{\vdash \Gamma, X{:}\kappa}$$

Figure 3.3: Rules for checking a context is well-formed

$$\boxed{T^- \dashrightarrow T'^+}$$

$$\frac{}{(\lambda\, x{:}T_1.\, T_2)\ t \dashrightarrow T_2[t/x]^{T_1}} \qquad \frac{}{(\Lambda\, X{:}\kappa.\, T_2) \cdot T_1 \dashrightarrow T_2[T_1/X]}$$

$$\frac{T \dashrightarrow T'}{T\ t \dashrightarrow T'\ t} \qquad \frac{T_1 \dashrightarrow T_1'}{T_1 \cdot T_2 \dashrightarrow T_1' \cdot T_2}$$

Figure 3.4: Reduction to weak head normal form for types

$$\boxed{T^- \dashrightarrow_\beta T'^+} \qquad \boxed{\kappa^- \dashrightarrow_\beta \kappa'^+}$$

$$\frac{}{(\lambda\, x{:}T_1.\, T_2)\ t \dashrightarrow_\beta T_2[t/x]^{T_1}} \qquad \frac{}{(\lambda\, X{:}\kappa.\, T)\cdot T' \dashrightarrow_\beta T[T'/X]}$$

$$\frac{T_1 \dashrightarrow_\beta T_1'}{T_1\ t \dashrightarrow_\beta T_1'\ t}$$

$$\frac{T_1 \dashrightarrow_\beta T_1'}{T_1 \cdot T_2 \dashrightarrow_\beta T_1' \cdot T_2} \qquad \frac{T_2 \dashrightarrow_\beta T_2'}{T_1 \cdot T_2 \dashrightarrow_\beta T_1 \cdot T_2'}$$

$$\frac{T_1 \dashrightarrow_\beta T_1'}{\Gamma \vdash \lambda\, x{:}T_1.\, T_2 \dashrightarrow_\beta \lambda\, x{:}T_1'.\, T_2} \qquad \frac{T_2 \dashrightarrow_\beta T_2'}{\lambda\, x{:}T_1.\, T_2 \dashrightarrow_\beta \lambda\, x{:}T_1.\, T_2'}$$

$$\frac{\kappa \dashrightarrow_\beta \kappa'}{\lambda\, X{:}\kappa.\, T \dashrightarrow_\beta \lambda\, X{:}\kappa'.\, T} \qquad \frac{T \dashrightarrow_\beta T'}{\lambda\, X{:}\kappa.\, T \dashrightarrow_\beta \lambda\, X{:}\kappa.\, T'}$$

$$\frac{T_1 \dashrightarrow_\beta T_1' \quad \mathcal{Q} \in \{\Pi, \forall, \iota\}}{\mathcal{Q}\, x{:}T_1.\, T_2 \dashrightarrow_\beta \mathcal{Q}\, x{:}T_1'.\, T_2} \qquad \frac{T_2 \dashrightarrow_\beta T_2' \quad \mathcal{Q} \in \{\Pi, \forall, \iota\}}{\mathcal{Q}\, x{:}T_1.\, T_2 \dashrightarrow_\beta \mathcal{Q}\, x{:}T_1.\, T_2'}$$

$$\frac{\kappa \dashrightarrow_\beta \kappa'}{\forall\, X{:}\kappa.\, T \dashrightarrow_\beta \forall\, X{:}\kappa'.\, T} \qquad \frac{T \dashrightarrow_\beta T'}{\forall\, X{:}\kappa.\, T \dashrightarrow_\beta \forall\, X{:}\kappa.\, T'}$$

$$\frac{T \dashrightarrow_\beta T'}{\Pi\, x{:}T.\, \kappa \dashrightarrow_\beta \Pi\, x{:}T'.\, \kappa} \qquad \frac{\kappa \dashrightarrow_\beta \kappa'}{\Pi\, x{:}T.\, \kappa \dashrightarrow_\beta \Pi\, x{:}T.\, \kappa'}$$

$$\frac{\kappa_1 \dashrightarrow_\beta \kappa_1'}{\Pi\, X{:}\kappa_1.\, \kappa_2 \dashrightarrow_\beta \Pi\, X{:}\kappa_1'.\, \kappa_2} \qquad \frac{\kappa_2 \dashrightarrow_\beta \kappa_2'}{\Pi\, X{:}\kappa_1.\, \kappa_2 \dashrightarrow_\beta \Pi\, X{:}\kappa_1.\, \kappa_2'}$$

Figure 3.5: Term-free reduction of types and kinds

$$\boxed{T_1^- \cong T_2^-} \qquad \boxed{T_1^- \cong^{\mathrm{w}} T_2^-}$$

$$\frac{T_1 \dashrightarrow^* T_1' \not\dashrightarrow \quad T_2 \dashrightarrow^* T_2' \not\dashrightarrow \quad T_1' \cong^{\mathrm{w}} T_2'}{T_1 \cong T_2}$$

$$\frac{}{X \cong^{\mathrm{w}} X} \qquad \frac{|t_1| =_{\beta\eta} |t_1'| \quad |t_2| =_{\beta\eta} |t_2'|}{\{t_1 \simeq t_2\} \cong^{\mathrm{w}} \{t_1' \simeq t_2'\}}$$

$$\frac{\kappa \cong \kappa' \quad T \cong T'}{\forall\, X{:}\kappa.\, T \cong^{\mathrm{w}} \forall\, X{:}\kappa'.\, T'} \qquad \frac{T_1 \cong T_1' \quad T_2 \cong T_2' \quad \mathcal{Q} \in \{\Pi, \forall, \iota\}}{\mathcal{Q}\, x{:}T_1.\, T_2 \cong^{\mathrm{w}} \mathcal{Q}\, x{:}T_1'.\, T_2'}$$

$$\frac{T_1 \cong T_1' \quad T_2 \cong T_2'}{\lambda\, x{:}T_1.\, T_2 \cong^{\mathrm{w}} \lambda\, x{:}T_1'.\, T_2'} \qquad \frac{\kappa \cong \kappa' \quad T \cong T'}{\lambda\, X{:}\kappa.\, T \cong^{\mathrm{w}} \lambda\, X{:}\kappa'.\, T'}$$

$$\frac{T_1 \cong^{\mathrm{w}} T_1' \quad T_2 \cong T_2'}{T_1 \cdot T_2 \cong^{\mathrm{w}} T_1' \cdot T_2'} \qquad \frac{T \cong^{\mathrm{w}} T' \quad |t| =_{\beta\eta} |t'|}{T\ t \cong^{\mathrm{w}} T'\ t'}$$

Figure 3.6: Type constructor convertibility

as *Nat* or *Bool*), $\Pi\, X : \kappa_1 .\, \kappa_2$ classifies type-level functions that abstract over type constructors of kind κ_1 and return a type constructor of kind κ_2 (e.g., type constructor $\lambda\, X : \star .\, X \to X$ has kind $\Pi\, X : \star .\, \star$), and $\Pi\, x : T .\, \kappa$ classifies type-level function that abstracts over terms of type T and return type constructors of kind κ (e.g., type constructor $\lambda\, x : \mathit{Nat} .\, \{x \simeq \mathit{zero}\}$ has kind $\Pi\, x : \mathit{Nat} .\, \star$).

Notation 3.10 (Non-dependent function kinds)**.** For kinds of the form $\Pi\, X : \kappa_1 .\, \kappa_2$ (resp. $\Pi\, x : T .\, \kappa$), if $X \notin FV(\kappa_2)$ (resp. $x \notin FV(\kappa)$) then we may write $\kappa_1 \to \kappa_2$ (resp. $T \to \kappa$) as abbreviation.

The type formers that CDLE inherits from CC are: type variables; impredicative type constructor quantification $\forall\, X : \kappa .\, T$, used to type polymorphic terms; dependent product types $\Pi\, x : T .\, T'$, used to type functions; abstractions over terms $\lambda\, x : T .\, T'$ and over type constructors $\lambda\, X : \kappa .\, T$; and applications of type constructors to terms $T\ t$ and other type constructors $T_1 \cdot T_2$.

3.1.3.2 Beyond CC

CDLE has three additional type formers beyond those of its CC subsystem. There are:

- *implicit dependent product* types $\forall\, x : T_1 .\, T_2$, used to type functions whose argument exists solely for typing (that is, the argument is not used in computation);
- *dependent intersection* types $\iota\, x : T_1 .\, T_2$, used to assign two types at once to a term, and where the second type assignment may refer to the first; and
- *equality* types $\{t_1 \simeq t_2\}$.

For now, we are only concerned with the rules which say when these types are well-kinded; their meaning will become clear in Section 3.1.4.

Checking well-kindedness for implicit dependent products and dependent intersection follows the same recipe as does kinding of dependent product types, e.g., $\forall\, x : T.\, T'$ has kind $\star$ if T has kind $\star$ and if T' has kind $\star$ under a typing context extended by the assumption $x : T$. For equality types, the only requirement for the type $\{t \simeq t'\}$ to be well-formed is that the free variables of both t and t' (written $FV(t\ t')$) are declared in the typing context. Thus, the equality is *untyped*, as neither t nor t' need be typable.

Notation 3.11 (Non-dependent function types.)**.**

- For types of the form $\Pi\, x : T_1.\, T_2$, if $x \notin FV(T_2)$ then we may write $T_1 \to T_2$.
- For types of the form $\forall\, x : T_1.\, T_2$, if $x \notin FV(T_2)$ then we may write $T_1 \Rightarrow T_2$.

3.1.4 Bidirectional Typing of Terms and Erasure

Figure 3.7 gives the type inference rules for the term constructs of CDLE. These type inference rules are *bidirectional* [32]: judgement $\Gamma \vdash t \in T$ indicates term t *synthesizes* type T under typing context Γ and judgement $\Gamma \vdash T \ni t$ indicates t can be *checked* against type T. These rules are to be read bottom-up as an algorithm for type inference, with Γ and t considered inputs in both judgements and the type T an output in the synthesis judgement and input in the checking judgement. We write $\overset{\leftrightarrow}{\in}$ to range over $\{\ni, \in\}$, and when this symbol occurs multiple times in a rule, it is intended that such occurrences be read the same way (i.e., read the occurrences as either all $\ni$ or all $\in$). During type inference, types may be call-by-name reduced to weak head normal form in order to reveal the outermost construct used to form a type. We use the shorthand $\Gamma \vdash t\ \overset{\dashrightarrow^*}{\in}\ T$ (defined formally near the top of Figure 3.7) in some premises to indicate that t synthesizes some type T' that reduces to T.

$$\boxed{\Gamma^- \vdash t^- \in T^+} \quad \boxed{\Gamma^- \vdash T^- \ni t^-}$$

$$\Gamma \vdash t \overset{\dashrightarrow^*}{\in} T \;=\; \exists T'.\, (\Gamma \vdash t \in T') \wedge (T' \dashrightarrow^* T)$$

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x \in T} \qquad \frac{\Gamma \vdash t \in T' \quad T' \cong T}{\Gamma \vdash T \ni t}$$

$$\frac{T \dashrightarrow^* \Pi\, x{:}T_1.\, T_2 \quad \Gamma, x{:}T_1 \vdash T_2 \ni t}{\Gamma \vdash T \ni \lambda\, x.\, t} \qquad \frac{\Gamma \vdash t \overset{\dashrightarrow^*}{\in} \Pi\, x{:}T_1.\, T_2 \quad \Gamma \vdash T_1 \ni t_1}{\Gamma \vdash t\ t_1 \in T_2[t_1/x]^{T_1}}$$

$$\frac{T' \dashrightarrow^* \forall\, X{:}\kappa.\, T \quad \Gamma, X{:}\kappa \vdash T \ni t}{\Gamma \vdash T' \ni \Lambda\, X.\, t} \qquad \frac{\Gamma \vdash t \overset{\dashrightarrow^*}{\in} \forall\, X{:}\kappa.\, T \quad \Gamma \vdash T' : \kappa' \quad \kappa' \cong \kappa}{\Gamma \vdash t \cdot T' \in T[T'/X]}$$

$$\frac{\begin{array}{c} T \dashrightarrow^* \forall\, x{:}T_1.\, T_2 \\ \Gamma, x{:}T_1 \vdash T_2 \ni t \quad x \notin FV(|t|) \end{array}}{\Gamma \vdash T \ni \Lambda\, x.\, t} \qquad \frac{\Gamma \vdash t \overset{\dashrightarrow^*}{\in} \forall\, x{:}T'.\, T \quad \Gamma \vdash T' \ni t'}{\Gamma \vdash t\ \text{-}t' \in T[t'/x]^{T'}}$$

$$\frac{\begin{array}{c} T \dashrightarrow^* \iota\, x{:}T_1.\, T_2 \quad \Gamma \vdash T_1 \ni t_1 \\ \Gamma \vdash T_2[t_1/x]^{T_1} \ni t_2 \quad |t_1| =_{\beta\eta} |t_2| \end{array}}{\Gamma \vdash T \ni [t_1, t_2]} \qquad \frac{\Gamma \vdash t \overset{\dashrightarrow^*}{\in} \iota\, x{:}T_1.\, T_2}{\Gamma \vdash t.1 \in T_1} \quad \frac{\Gamma \vdash t \overset{\dashrightarrow^*}{\in} \iota\, x{:}T_1.\, T_2}{\Gamma \vdash t.2 \in T_2[t.1/x]}$$

$$\frac{\begin{array}{c} T \dashrightarrow^* \{t_1 \simeq t_2\} \\ FV(t') \subseteq DV(\Gamma) \quad |t_1| =_{\beta\eta} |t_2| \end{array}}{\Gamma \vdash T \ni \beta\{t'\}} \qquad \frac{\Gamma \vdash t \in T' \quad T' \cong \{\lambda\, x.\, \lambda\, y.\, x \simeq \lambda\, x.\, \lambda\, y.\, y\}}{\Gamma \vdash T \ni \delta\ \text{-}\ t}$$

$$\frac{\begin{array}{c} \Gamma \vdash t \overset{\dashrightarrow^*}{\in} \{t_1 \simeq t_2'\} \quad FV(t_2) \subseteq DV(\Gamma) \quad |t_2'| =_{\beta\eta} |t_2| \\ \Gamma \vdash T'[t_2/x] : \star \qquad \Gamma \vdash T'[t_2/x] \ni t' \quad T'[t_1/x] \cong T \end{array}}{\Gamma \vdash T \ni \rho\ t\ @x\langle t_2\rangle.T'\ \text{-}\ t'}$$

$$\frac{\begin{array}{c} \Gamma \vdash t' \overset{\leftrightarrow}{\in} T \\ FV(t'') \subseteq DV(\Gamma) \quad \Gamma \vdash \{t' \simeq t''\} \ni t \end{array}}{\Gamma \vdash \varphi\ t\ \text{-}\ t'\ \{t''\} \overset{\leftrightarrow}{\in} T} \qquad \frac{\Gamma \vdash T : \star \quad \Gamma \vdash T \ni t}{\Gamma \vdash \chi\ T\ \text{-}\ t\ \in T}$$

$$\frac{\Gamma \vdash T_1 \ni t_1 \quad \Gamma \vdash t_2[t_1/x]^{T_1} \overset{\leftrightarrow}{\in} T_2}{\Gamma \vdash [x : T_1 = t_1]\ \text{-}\ t_2 \overset{\leftrightarrow}{\in} T_2}$$

Figure 3.7: Bidirectional type inference rules for terms

$$
\begin{array}{lcllcl}
|x| & = & x & |\lambda\, x.\, t| & = & \lambda\, x.\, |t| \\
|t\ t'| & = & |t|\ |t'| & |t \cdot T| & = & |t| \\
|\Lambda\, x.\, t| & = & |t| & |t\ \text{-}t'| & = & |t| \\
|[t, t']| & = & |t| & |t.1| & = & |t| \\
|t.2| & = & |t| & |\beta\{t\}| & = & |t| \\
|\rho\ t\ @x.T'\ \text{-}\ t'| & = & |t'| & |\varphi\ t\ \text{-}\ t'\ \{t''\}| & = & |t''| \\
|\chi\ T\ \text{-}\ t| & = & |t| & |\delta\ \text{-}\ t| & = & \lambda\, x.\, x \\
|[x : T_1 = t_1]\ \text{-}\ t_2| & = & |t_2[t_1/x]| & & &
\end{array}
$$

Figure 3.8: Erasure for annotated terms

In describing both the new and old type constructs of CDLE, we make reference to the erasures of the corresponding annotations for terms. The full definition of the erasure function $|-|$, which extracts an untyped lambda calculus term from a term with type annotations, is given in Figure 3.8.

3.1.4.1 Subsystem CC

We briefly overview the subsystem of CDLE inherited from CC.

- $\Pi\, x : T_1.\, T_2$ is called a *dependent product type*, *dependent function type*, or simply Π-*type*. The introduction form is lambda abstractions $\lambda\, x.\, t$ (erasing to $\lambda\, x.\, |t|$), where the body t is checked to have type T_2 under the assumption that bound variable x has type T_1, and the elimination form is applications $t\ t_1$ (erasing to $|t|\ |t_1|$), where the argument t_1 is checked against type T_1 and the entire expression has type $T_2[t_1/x]^{T_1}$.
- $\forall\, X : \kappa.\, T$ is for *polymorphic terms*. The introduction form is type constructor abstraction $\Lambda\, X.\, t$ (erasing to $|t|$), where the body t is checked to have type T under the assumption that type variable X has kind κ, and the elimination form is type constructor instantiation $t \cdot T$ (erasing to $|t|$), where the type constructor

argument is checked to have a kind convertible with κ and the entire expression has type $T[T'/X]$.

- Local definitions ("let-expressions") of terms have the syntax $[x : T_1 = t_1]$ - t_2, which we read as *"let x be t_1 in t_2"*. Let-expressions may either have their types checked or synthesized; either way, when performing type inference on the body we substitute all free occurrences of x with t_1 (with an annotation). Let-expressions erase to the erasure of t_2 with all remaining free occurrences x replaced by the erasure of t_1).
- Type annotations χ T - t allow us to ascribe a type to a term whose type we might only be able to check, with the type of the whole expression being synthesized. Note that this is not arbitrary type coercion, which would be unsound: the entire expression synthesizes type T *only when* t can be checked against type T.
- Finally, we have a so-called judgemental rule (top right of the figure) that allows us to move from type checking to synthesis (reading bottom-up). It says that to check term t against type T, it suffices that t synthesizes a convertible type T'. There is an implicit side condition that t is of a form whose type we are otherwise only able to synthesize; for the CC subsystem, that's when t is a variable, application, or type constructor instantiation.

Remark 3.12 (Omitted type constructor arguments). In Cedille code listings, type arguments are sometimes omitted when the Cedille tool can infer these from the types of term arguments.

3.1.4.2 The Implicit Product Type $\forall\, x{:}T_1.\, T_2$

We now turn to the first of the type formers with which CDLE extends CC. The implicit dependent product type of Miquel [65] is similar to dependent products, except that they are restricted such that they may only use their arguments in a computationally irrelevant fashion. Note that the notion of computational irrelevance here is not that of a different sort of classifier for types (e.g. *Prop* in Coq, see [87]) that separates terms in the language into those which can be used for computation and those which cannot. Instead, it is similar to *quantitative type theory* [7]: relevance and irrelevance are properties of binders, indicating how a function may *use* an argument. For CDLE specifically, this means that a function with an irrelevant parameter may only use that parameter where the erasure function guarantees it will be removed.

Implicit products are introduced with $\Lambda\, x.\, t$, and the type inference rule is the same as for ordinary function abstractions except for the additional condition that x does not occur free in the erasure of the body t ($x \notin FV(|t|)$). Thus, the argument can play no computational role in the function and exists solely for the purposes of typing. The erasure of the introduction form is $|t|$. For application, if t has type $\forall\, x{:}T_1.\, T_2$ and t' has type T_1, then t -t' has type $T_2[t'/x]^{T_1}$ and erases to $|t|$.

Remark 3.13. As an alternative to checking occurrences of free variables in the erasure of of annotated terms, one could instead enforce the restrictions on bound variables by tracking their usage information in the context. This is the approach taken by McBride [62] and Atkey [7] for quantitative type theory, as there they are interested in tracking usage information more general than just "erased" or "unerased."

3.1.4.3 The Dependent Intersection Type $\iota\, x{:}T_1.\, T_2$

The dependent intersection type of Kopylov [52] is the type for terms which can be assigned two different types, where the second type assignment may refer to the first. It is a dependent generalization of intersection types (see Barendregt et al. [8]) in Curry-style theories, which allow one to express the fact that an untyped lambda calculus term can be assigned two different types. In the annotated language, the introduction form for dependent intersections is $[t_1, t_2]$, and it can be checked against type $\iota\, x{:}T_1.\, T_2$ if t_1 can be checked against type T_1, t_2 can be checked against $T_2[t_1/x]^{T_1}$, and t_1 and t_2 are $\beta\eta$-equivalent modulo erasure. For the elimination forms, if t synthesizes type $\iota\, x{:}T_1.\, T_2$ then $t.1$ (which erases to $|t|$) synthesizes type T_1, and $t.2$ (erasing to the same) synthesizes type $T_2[t.1/x]$.

Dependent intersections can be thought of as a dependent pair type where the two components are equal. Thus, we may "forget" the second component: $[t_1, t_2]$ erases to $|t_1|$. Put another way, dependent intersections are a restricted form of computationally transparent subset types where the proof that some term t inhabits the subset must be definitionally equal to t. A consequence of this restriction is that the proof may be recovered, in the form of t itself, for use in computation.

3.1.4.4 The Equality Type $\{t_1 \simeq t_2\}$

$\{t_1 \simeq t_2\}$ is the type of proofs that t_1 is provably equal to t_2. The introduction form $\beta\{t'\}$ proves reflexive equations between $\beta\eta$-equivalence classes of terms: it can be checked against the type $\{t_1 \simeq t_2\}$ if $|t_1| =_{\beta\eta} |t_2|$ and if the subexpression t' has no undeclared free variables. We discuss the significance of the fact that t' is unrelated

to the terms being equated, dubbed the *Kleene trick*, below. In code listings, if t' is omitted from the introduction form, it defaults to $\lambda\, x.\, x$.

Remark 3.14 (Reflexive equations). In the code listings involving equational proofs in Chapter 2, we used the term *refl* rather than β to avoid confusion. We can now reveal that *refl* can be defined using β.

```
refl : ∀ X: ⋆. Π x: X. { x ≃ x }
= Λ X. λ x. β.
```

The elimination form $\rho\ t\ @x\langle t_2\rangle.T'$ - t' for the equality type $\{t_1 \simeq t_2'\}$ lets us perform substitution: it replaces specified occurrences of t_1 in the checked type with t_2 before checking t'.

- The first subexpression t must synthesize (possibly after some normalization) an equality type of the form $\{t_1 \simeq t_2'\}$. This is the equality to rewrite by.
- The sequence of tokens $@x\langle t_2\rangle T'$ is called the *guide* of the substitution, and its purpose is to allow the user to specify (using the variable x) the occurrences of t_1 to replace and to ensure that the resulting substitution is well-typed.

  As there is no guarantee that the right-hand side of the synthesized equation, t_2', is well-typed, the user gives a term t_2 which must be definitionally equal to t_2' and when substituted for occurrences of x in T' produces a well-formed type.
- The term t' is then checked against the type $T'[t_2/x]$.
- Finally, $T'[t_1/x]$ (which need not be well-kinded) must be convertible with the expected type T.

The entire expression erases to $|t'|$. In particular, the equality proof t is erased, meaning that its computational content is irrelevant to the substitution. This aspect of equality elimination helps to make sense of the *Kleene trick*.

In Cedille, the guide is optional, and a heuristic is used to produce a resulting type that is well-kinded. The implementation of Cedille as of July 28, 2023 does not support specifying t_2 in the guide $@x\langle t_2\rangle.T$.

Equality Axioms

Equality types in CDLE come with two additional axioms: a strong form of the direct computation rule of NuPRL [6] given by φ, and proof by contradiction given by δ. The inference rule for an expression of the form $\varphi\ t$ - $t'\ \{t''\}$ says that the entire expression can be checked against (or synthesize) type T if t' can, if there are no undeclared free variables in t'' (so, t'' is a well-scoped but otherwise untyped term), and if t proves that t' and t'' are equal. The crucial feature of φ is its erasure: the expression erases to $|t''|$, effectively enabling us to cast t'' to the type of t'. Again, the equality witness is erased.

An expression of the form δ - t may be checked against any type if t synthesizes a type convertible with a particular false equation, $\{\lambda\, x.\, \lambda\, y.\, x \simeq \lambda\, x.\, \lambda\, y.\, y\}$. To broaden the class of false equations to which a user may apply δ, the Cedille tool implements the *Böhm-out* semi-decision procedure [16] for discriminating between $\beta\eta$-inequivalent terms. The expression δ - t erases to $\lambda\, x.\, x$, so once again the equality witness is erased.

Remark 3.15. There is a subtlety in the formulation of the δ axiom. We cannot combine the two premises to the single premise $\Gamma \vdash t\ \overset{\dashrightarrow^*}{\in}\ \{\lambda\, x.\, \lambda\, y.\, x \simeq \lambda\, x.\, \lambda\, y.\, y\}$, as the reduction relation $\dashrightarrow^*$ for types does not reduce term subexpressions and thus the rule would fail to apply when $t\ \overset{\dashrightarrow^*}{\in}\ \{t_1 \simeq t_2\}$ with $|t_1| =_{\beta\eta} \lambda\, x.\, \lambda\, y.\, x$ and $|t_2| =_{\beta\eta} \lambda\, x.\, \lambda\, y.\, y$. While this could be repaired by instead having the premise be $\Gamma \vdash \{\lambda\, x.\, \lambda\, y.\, x \simeq \lambda\, x.\, \lambda\, y.\, y\} \ni t$, this would not generalize to the above-mentioned generalization taken by the Cedille tool where we first synthesize the type of t, then check whether the terms of the synthesized equality type are Böhm-seperable.

Finally, the Cedille tool provides a symmetry axiom ς for equality types, with $|t|$ the erasure of $\varsigma\ t$. This axiom is purely a convenience; without ς, symmetry for equality types can be proven with ρ.

Remark 3.16 (Substitution). In the code listings involving equational proofs in Chapter 2, we used the term *subst* rather than ρ to avoid confusion. We can now reveal that *subst* can be defined using ρ.

```
subst
: ∀ X: ⋆. ∀ P: X → ⋆. Π x1: X. Π x2: X. { x1 ≃ x2 } → P x1 → P x2
= Λ X. Λ P. λ x1. λ x2. λ eq. λ p. ρ (ς eq) @x<x2>.(P x) - p .
```

The Kleene trick

As mentioned earlier, the introduction form $\beta\{t'\}$ for the equality type contains a subexpression t' that is unrelated to the equated terms. By allowing t' to be any well-scoped term, we are able to define a type of all untyped lambda calculus terms.

Definition 3.17 (Top)**.** Let *Top* be the type $\{\lambda\, x.\, x \simeq \lambda\, x.\, x\}$.

We dub this the *Kleene trick*, as one may find the idea in Kleene's later definitions of numeric realizability in which any number is allowed as a realizer for a true atomic formula [50].

Combined with dependent intersections, the Kleene trick also allows us to derive computationally transparent equational subset types. For example, let *Nat* be the type of natural numbers, *Bool* the type of booleans with *true* the truth value, and *isEven* : *Nat* $\to$ *Bool* a function returning *true* if and only if its argument is even. Then, the type even naturals can be defined as $\iota\, x : Nat.\, \{isEven\ x \simeq true\}$.

3.2 Meta-Theory

Now that we have discussed in depth the theory of CDLE, we turn our attention to its *meta-theory*. Understanding the meta-theoretic properties of the kernel

language of an ITP is of vital importance, as they tell us what users of the ITP can expect of the programs and proofs carried out within it. This covers everything from basic sanity checks, such as whether types synthesized from terms are always well-formed, to the nature of type inference itself, such as whether it is decidable, to the language's usefulness as a logic, which would be ruined if it is inconsistent.

This section presents only a summary of the important meta-theoretic properties of CDLE. For full details, including worked-out proofs of these properties and details of the realizability semantics used to establish logical consistency, see Stump and Jenkins [83].

3.2.1 Logical Consistency

It may concern the reader that, with the Kleene trick, it is possible to type nonterminating terms, leading to a failure of normalization in general in CDLE. For example, the looping term Ω, which in the untyped lambda calculus is $(\lambda\, x.\, x\; x)\; \lambda\, x.\, x\; x$, can be checked against type *Top* by adding annotations to obtain the term $\beta\{(\lambda\, x.\, x\; x)\; \lambda\, x.\, x\; x\}$. More subtly, the φ axiom in combination with impredicativity allows nontermination in inconsistent contexts (see Abel and Coquand [2]). Assume there is a typing context Γ and term t such that $\Gamma \vdash t \in \forall\, X : \star.\, X$, and let ω be the following term.

$$\varphi\ (t \cdot \{\lambda\, x.\, x \simeq \lambda\, x.\, x\ x\})\ \text{-}\ (\Lambda\, X.\, \lambda\, x.\, x)\ \{\lambda\, x.\, x\ x\}$$

Under Γ, ω can be checked against the type $\forall\, X : \star.\, X \to X$, and by the erasure rules ω erases to $\lambda\, x.\, x\ x$. We can then type the looping term Ω:

$$\Gamma \vdash \omega \cdot (\forall\, X : \star.\, X \to X)\ \omega \in \forall\, X : \star.\, X \to X$$

The existence of non-normalizing terms does not threaten the logical consistency of CDLE. For example, extensional Martin-Löf type theory is consistent but, due to a similar difficulty with inconsistent contexts, is non-normalizing [33]. Concerning the two counter-examples to termination just discussed, we can make the following informal observations as to why they do not affect logical consistency. For the example using the Kleene trick, all non-normalizing terms constructed this way have as their type an equality, not an arbitrary type, and any such type that has a closed inhabitant (that is, a trivially true equation) cannot be used to construct values of an arbitrary type, because the type system already affirms these definitional equalities. The example using φ relies on an inconsistent context, and logical consistency concerns the typing of closed terms. Since that example already assumes the existences of a term t and context Γ such that $\Gamma \vdash t \in \forall\, X : \star.\, X$, we can guarantee nothing *viz-à-vis* logical consistency in that context.

Below, Proposition 3.18 states formally the logical consistency of CDLE.

Proposition 3.18 (Logical consistency [83])**.**
There is no term t such that $\emptyset \vdash t \in \forall\, X : \star.\, X$.

3.2.2 Termination Guarantee

The previous counter-examples to termination we saw using the Kleene trick and φ do not preclude the possibility of a qualified guarantee of termination in CDLE. However, they do serve to illustrate the restrictions we must place on terms in order to guarantee their termination. The Kleene trick tells us that we cannot expect termination for terms of an *arbitrary* type, as it allows us to embed the untyped lambda calculus into any provable equality type. The example of typing Ω with φ

tells us we can make no guarantees about terms in an inconsistent context.

Given these constraints, the meta-theory of CDLE provides the following guarantee: closed terms of a function type are call-by-name normalizing.

Proposition 3.19 (Call-by-name normalization for functions [83])**.** *Suppose that* $\emptyset \vdash t \in \Pi\, x : T_1.\, T_2$*. Then* $|t|$ *is call-by-name normalizing, e.g., there exists an untyped lambda calculus term* t' *such that* $|t| \dashrightarrow^* \lambda\, x.\, t'$*.*

Thus, in CDLE the termination guarantee one has for a given closed term depends upon the type we are able to assign it.

CDLE's language of type constructors and kinds also features reduction, and in particular the type inference rules for terms relies on this reduction to put types in weak head normal form. Neither type constructors nor kinds can be computed from terms in CDLE, so reduction to weak head normal form for these cannot fail solely due to a nonterminating term. We therefore expect — and have — that when we remove reduction of terms from the reduction relation for type constructors and kinds, the language of type constructors and kinds is terminating.

Proposition 3.20 (Termination for type constructors and kinds [83])**.** *Assume* $\vdash \Gamma$*.*

1. *If* $\Gamma \vdash \kappa$ *then there exists* κ' *such that* $\kappa \dashrightarrow^*_\beta \kappa' \not\dashrightarrow_\beta$
2. *If* $\Gamma \vdash T : \kappa$ *then there exists* T' *such that* $T \dashrightarrow^*_\beta T' \not\dashrightarrow_\beta$

In the termination guarantee for terms, the restriction to terms of a function type may at first appear to be too strict. After all, in CDLE there are many more type constructs than just Π-types. Additionally, we of course wish to ensure that the CDLE types to which we elaborate surface language datatypes ensure call-by-name normalization for closed terms. This motivates the following generalization of Proposition 3.19: for any type T whose terms can be *retyped* to a function type, we

have that its closed terms are also call-by-name normalizing.

Corollary 3.21. *Suppose that* $\emptyset \vdash t \in T$ *and that there exists* t' *such that* $|t'| = \lambda\, x.\, x$ *and* $\emptyset \vdash t' \in T \to \Pi\, x : T_1.\, T_2$ *for some* T_1 *and* T_2*. Then* $|t|$ *is call-by-name normalizing.*

Proof. We have that $\emptyset \vdash t\ t' \in \Pi\, x : T_1.\, T_2$, so apply Proposition 3.19 to obtain that $|t'\ t|$ is call-by-name normalizing. By the definition of erasure and assumption, $|t'\ t| = (\lambda\, x.\, x)\ |t|$. The next step of call-by-name operational semantics reduces the term to $|t|$. Therefore, $|t|$ is also call-by-name normalizing. □

When we formally present Cedille's datatype system, we will ensure that all datatypes are elaborated to types that satisfy the premise of Corollary 3.21. We will revisit the notion of "retyping" in Section 3.3.3, where it is formally presented as the derived type construct *Cast*.

Remark 3.22 (Apologia). Lack of normalization in general does mean that type inference in CDLE (and thus Cedille) is formally undecidable, as there are several inference rules in which full $\beta\eta$-equivalence of terms is checked, meaning that we must reduce open terms. The experience of the Cedille team shows that this is not a significant impediment for the practical use of the theory, as it is rare for such nonterminating terms to occur unintentionally in the properties we are interested in proving. Furthermore, even in implementations of strongly normalizing dependent type theories, it is possible for type inference to check convertibility of terms that contain astronomically slow functions, effectively causing the implementation to hang. Since this is the case, and since the features that lead to failure of strong normalization in CDLE also enable some of its more unique expressive capabilities (see Section 3.3), it is my assessment that where it comes to CDLE's intended purpose, the benefits outweigh the drawbacks.

3.2.3 Type Preservation

As we discussed in Section 3.1, the operational semantics of CDLE is that of the untyped lambda calculus. In particular, there is *no* operational semantics for the annotated language itself, as these annotations only exist to facilitate a machine implementation of a type assignment system. We therefore cannot state the usual version of the type preservation theorem, an important sanity check for intrinsically

typed theories that says the operational semantics preserves typing, for the extrinsically typed theory CDLE. However, we can state (and prove) a semantic version of type preservation for CDLE. The realizability semantics for CDLE that establishes the theory's logical consistency interprets types as sets of $\beta\eta$-equivalence classes of untyped lambda calculus terms. By construction, therefore, if an untyped term is in the meaning of a type, then so is any term it reduces to.

Below, $[\![T]\!]_{\sigma.\rho}$ denotes the interpretation of type T where σ gives a valuation for free term variables and ρ gives a valuation for free type variables, and $t \dashrightarrow_{\beta\eta} t'$ denotes a computation step of untyped term t to t' under full $\beta\eta$-reduction.

Proposition 3.23 (Type Preservation (Semantic) [83])**.** *If $t \in [\![T]\!]_{\sigma,\rho}$ and $t \dashrightarrow_{\beta\eta} t'$ then $t' \in [\![T]\!]_{\sigma,\rho}$.*

Proposition 3.23 can be seen as the semantic justification for the φ axiom of equality, reflecting the choice of the preferred model of CDLE back into the theory itself. We can illustrate this with a simple proof below of a syntactic version of type preservation.

Proposition 3.24 (Type Preservation (Syntactic))**.** *If $\Gamma \vdash t \in T$ and $|t| \dashrightarrow t'$ for some untyped term t' then there exists t'' such that $|t''| = t'$ and $\Gamma \vdash t'' \in T$.*

Proof. Exhibit $\varphi\ \beta$ - $t\ \{t'\}$ for witness t''. It satisfies required conditions by assumption and the erasure and typing rules for φ. □

3.2.4 Other Meta-Theoretic Properties

Aside from the above-discussed semantics properties, there are other meta-theoretic results of a more formalistic nature that impact machine implementations of both the kernel theory and surface language. For example, when we check positivity of a type constructor to confirm its suitability as a signature for a datatype, we may

require reducing it (and its type-level subexpressions) to weak head normal form, after which we desire to know that the resulting type-expression is well-kinded and of a kind that is convertible with the kind of the type-expression we started with.

Proposition 3.25 (Subject reduction for type constructors and kinds [83])**.**
1. *If* $\Gamma \vdash \kappa$ *and* $\kappa \dashrightarrow_\beta \kappa'$ *then* $\Gamma \vdash \kappa'$
2. *If* $\Gamma \vdash T : \kappa$ *and* $T \dashrightarrow_\beta T'$ *then* $\Gamma \vdash T' : \kappa'$ *for some* κ' *such that* $\kappa \cong \kappa'$.

Judgement validity (or *judgement agreement*) means that if our inference rules associate a classifier to a term or type, that classifier is well-formed according to *its own* classification rules.

Proposition 3.26 (Judgement validity [83])**.** *If* $\vdash \Gamma$ *then:*
1. *if* $\Gamma \vdash T : \kappa$ *then* $\Gamma \vdash \kappa$
2. *if* $\Gamma \vdash t \in T$ *then* $\Gamma \vdash T : \star$

As an example of why this property is important, consider the following example. Assume we have $id : \forall\, X : \star.\, X \to X$ (it is easy enough to give a definition for this identifier) and a term t and context Γ such that $\Gamma \vdash t \in T$, but for which it is *not* the case that $\Gamma \vdash T : \star$. Then we run into the truly undesirable situation that we cannot apply id to t, because we cannot instantiate the type parameter X of id with T.

3.3 Derived Constructs of CDLE

In this section, we present two CDLE type constructs that will aide us in elaborating Cedille's datatype system, *View* and *Cast*. Each of these constructs are derivable within CDLE itself. However, rather than taking the reader through a full derivation of each, for the sake of comprehensibility we present each axiomatically — that is to say, giving for each construct rules for type formation, introduction,

elimination, and erasure. For full, worked-out derivations of each of these constructs, see Jenkins and Stump [48]. Understand that each construct presented this way may appear partially applied, even though in the inference rules they appear fully applied to arguments. For example, the type constructor *View*, when it appears by itself, has kind $\star \to \mathit{Top} \to \star$.

3.3.1 Motivation

Before we dive into the details of these derived type constructs, it is important to understand what they are intended to accomplish generally and what role they will play in the elaboration of inductive types specifically. The first construct, *View*, fully reflects CDLE's realizability semantics into the theory itself: as a proposition, $\mathit{View} \cdot T\ t$ expresses the existence of an assignment of the type T to untyped (in particular, *Top*-typed) term t. The second, *Cast*, is used to internalize the notion of a *retyping function* (see Corollary 3.21), inducing a notion of subtyping.

Recall the combinator for course-of-values recursion discussed in Section 2.2.6, *mcovr*. The computation rule tells us that the parameter that serves as the "abstract out" (a function of type $R \to F \cdot R$, where R is the parametric type variable) is always instantiated with $\mathit{out} : \mu F \to F \cdot \mu F$. When we generalize to dependent type theory, in order to reason about the behavior of programs constructed with *mcovr* it is important to provide evidence of this fact. This is exactly what *View* enables us to accomplish. Similarly, the parameter that serves to "reveal" that all terms of type R can be converted to a term of type μF is always instantiated with $\lambda\, x.\, x$, and to reason about program behavior it is important to give evience that the "reveal"

function is just a retyping function that plays no significant computational role — this is what *Cast* enables us to do. After introducing *View* in Section 3.3.2 and *Cast* in Section 3.3.3, we show how they can be put together to give a CDLE-idiomatic course-of-values induction principle in Section 3.4. *Cast* also plays a crucial role in checking positivity of datatype signatures, as it induces a preorder on types. This preorder is suitable for defining a notion, *in CDLE itself*, of datatype signature positivity (*Mono*, Section 3.3.3.2) as the property of a type constructor being *monotonic*.

3.3.2 Views

Figure 3.9 summarizes the derivation of the *View* type family in Cedille. There are two ways we can understand the type $\mathit{View} \cdot T\ t$. Arguably the most intuitive reading is that is a type whose inhabitants consist of the subset of the terms of type T that are provably equal to the untyped (more precisely *Top*-typed, see Definition 3.17) term t. The other reading, which we have outlined already, is that it is the type of proofs of the existence of an assignment of the type T to untyped term t. It is defined using dependent intersection: $\mathit{View} \cdot T\ t = \iota\, x : T.\, \{x \simeq t\}$.

The introduction form *intrView* takes an untyped term t_1 and two computationally irrelevant arguments: a term t_2 of type T and a proof t that t_2 is equal to t_1. The expression $\mathit{intrView}\ t_1\ \text{-}t_2\ \text{-}t$ erases to $(\lambda\, x.\, x)\ |t_1|$. The elimination form *elimView* takes an untyped term t and an erased argument t' proving that t may be viewed as having type T, and produces a term of type T. The crucial property of *elimView* is its erasure: $\mathit{elimView}\ t\ \text{-}t'$ also erases to $(\lambda\, x.\, x)\ |t|$, and so the expression is definitionally equal to t itself. Finally, *eqView* provides an extensionality law for

$$\frac{\Gamma \vdash T : \star \quad \Gamma \vdash \mathit{Top} \ni t}{\Gamma \vdash \mathit{View} \cdot T\ t : \star} \qquad \frac{\Gamma \vdash T : \star \quad \Gamma \vdash \mathit{Top} \ni t \quad \Gamma \vdash \mathit{View} \cdot T\ t \ni t'}{\Gamma \vdash \mathit{elimView} \cdot T\ t\ \text{-}t' \in T}$$

$$\frac{\Gamma \vdash T : \star \quad \Gamma \vdash \mathit{Top} \ni t_1 \quad \Gamma \vdash T \ni t_2 \quad \Gamma \vdash \{t_2 \simeq t_1\} \ni t}{\Gamma \vdash \mathit{intrView} \cdot T\ t_1\ \text{-}t_2\ \text{-}t \in \mathit{View} \cdot T\ t_1}$$

$$\frac{\Gamma \vdash T : \star \quad \Gamma \vdash \mathit{Top} \ni t \quad \Gamma \vdash \mathit{View} \cdot T\ t \ni t'}{\Gamma \vdash \mathit{eqView} \cdot T\ \text{-}t\ \text{-}t' \in \{t \simeq t'\}}$$

$$\begin{array}{rcl} |\mathit{elimView}| & = & \lambda\, x.\, x \\ |\mathit{intrView}| & = & \lambda\, x.\, x \\ |\mathit{eqView}| & = & \lambda\, x.\, x \end{array}$$

Figure 3.9: Internalized typing, axiomatically

views. It states that every proof t' of the proposition that t may be viewed as having type T is itself equal to t.

3.3.3 Casts

Understand $\mathit{Cast} \cdot S \cdot T$ as the type of proofs that there exists a *retyping function* from type S to type T. Formally, such a retyping function t has type $S \to T$ and erases to $\lambda\, x.\, x$. Under CDLE's realizability semantics, inhabitation of the type $\mathit{Cast} \cdot S \cdot T$ means that the interpretation of S (a set of $\beta\eta$-equivalence classes of untyped terms) is a subset of the interpretation of T. We therefore speak of $\mathit{Cast} \cdot S \cdot T$ as the type of *type inclusions* of S into T. We summarize casts in CDLE in Figure 3.10.

The introduction form, *intrCast*, takes two erased arguments: a function $t : S \to T$ and a proof that t is *extensionally* the identity function for all terms of type S. In intrinsic type theories, this would have limited utility as identity functions cannot map from S to T unless S and T are convertible types. But in an extrinsic type theory like CDLE, there are many nontrivial casts, especially in the presence of

$$\frac{\Gamma \vdash S : \star \quad \Gamma \vdash T : \star}{\Gamma \vdash \mathit{Cast} \cdot S \cdot T : \star}$$

$$\frac{\Gamma \vdash S : \star \quad \Gamma \vdash T : \star \quad \Gamma \vdash S \to T \ni t \quad \Gamma \vdash \Pi\, x{:}S.\, \{t\ x \simeq x\} \ni t'}{\Gamma \vdash \mathit{intrCast} \cdot S \cdot T\ \text{-}t\ \text{-}t' \in \mathit{Cast} \cdot S \cdot T}$$

$$\frac{\Gamma \vdash S : \star \quad \Gamma \vdash T : \star \quad \Gamma \vdash \mathit{Cast} \cdot S \cdot T \ni t}{\Gamma \vdash \mathit{elimCast} \cdot S \cdot T\ \text{-}t\ \in S \to T}$$

$$\frac{\Gamma \vdash S : \star \quad \Gamma \vdash T : \star \quad \Gamma \vdash \mathit{Cast} \cdot S \cdot T \ni t}{\Gamma \vdash \mathit{eqCast} \cdot S \cdot T\ \text{-}t \in \{\lambda\, x.\, x \simeq t\}}$$

$$\begin{array}{rcl} |\mathit{intrCast}| & = & \lambda\, x.\, x \\ |\mathit{elimCast}| & = & \lambda\, x.\, x \\ |\mathit{eqCast}| & = & \lambda\, x.\, x \end{array}$$

Figure 3.10: Casts, axiomatically

```
castRefl : ∀ S: ⋆. Cast ·S ·S
= Λ S. intrCast -(λ x. x) -(λ _. β) .

castTrans : ∀ S: ⋆. ∀ T: ⋆. ∀ U: ⋆. Cast ·S ·T ⇒ Cast ·T ·U ⇒ Cast ·S ·U
= Λ S. Λ T. Λ U. Λ c1. Λ c2.
intrCast -(λ x. elimCast -c2 (elimCast -c1 x)) -(λ x. β) .

castUnique
 : ∀ S: ⋆. ∀ T: ⋆. ∀ c1: Cast ·S ·T. ∀ c2: Cast ·S ·T. { c1 ≃ c2 }
= Λ S. Λ T. Λ c1. Λ c2. ρ ς (eqCast -c1) @x.{ x ≃ c2 } - eqCast -c2 .
```

Figure 3.11: Casts induce a preorder

the φ axiom. The elimination form, *elimCast*, takes as an erased argument a proof t of the inclusion of type S into type T. Its crucial property is its erasure: *elimCast* erases to $\lambda\, x.\, x$. The last construct listed in Figure 3.10 is *eqCast*, the extensionality principle. It states that every witness of a type inclusion is provably equal to $\lambda\, x.\, x$.

```
Mono : (⋆ → ⋆) → ⋆
= λ F: ⋆ → ⋆. ∀ X: ⋆. ∀ Y: ⋆. Cast ·X ·Y ⇒ Cast ·(F ·X) ·(F ·Y) .
```

Figure 3.12: Monotonicity

3.3.3.1 Casts Induce a Preorder on Types

Recall that a preorder $(S, \sqsubseteq)$ consists of a set S and a reflexive and transitive binary relation $\sqsubseteq$ over S. In the proof-relevant setting of type theory, establishing that a relation on types induces a preorder requires that we also show, for all types T_1 and T_2, that proofs of $T_1 \sqsubseteq T_2$ are unique — otherwise, we might only be working in a category. We now show that *Cast* satisfies all three of these properties.

Theorem 3.27. *Cast induces a preorder (or thin category) on Cedille types.*

Proof. Figure 3.11 gives the proofs in Cedille of reflexivity (*castRefl*), transitivity (*castTrans*), and uniqueness (*castUnique*) for *Cast*. □

3.3.3.2 Monotonicity

Monotonicity of a type scheme $F : \star \to \star$ in this preorder is defined as a lifting, for all types S and T, of any cast from S to T to a cast from $F \cdot S$ to $F \cdot T$. This is *Mono* in Figure 3.12. When we discuss positivity checking in Chapter 4, we will see how monotonicity proofs can be automatically derived for a family of type schemes.

3.4 Specification of Course-Of-Values Induction in CDLE

We now bring together what we have developed in this section to present the typing and computation laws for the CDLE-idiomatic Mendler-style course-of-values induction principle, shown in Figure 3.13. To understand this combinator, it may be useful to compare it with the course-of-values recursion combinator from Section 2.2.6.

$$\begin{array}{rcl} Alg \cdot P & =_{\mathbf{df}} & \forall\, R{:}\star.\, \forall\, c{:}\mathit{Cast} \cdot R \cdot \mu F.\, \mathit{View} \cdot (R \to F \cdot R)\ \beta\{\mathit{out}\} \Rightarrow \\ & & (\Pi\, x{:}R.\, P\ (\mathit{elimCast}\ \text{-}c\ x)) \to \\ & & \Pi\, xs{:}F \cdot R.\, P\ (\mathit{in}\ (\mathit{elimCast}\ \text{-}(\mathit{mono}\ c)\ xs)) \end{array}$$

$$\frac{\Gamma \vdash P : \mu F \to \star \quad \Gamma \vdash Alg \cdot P \ni t_1 \quad \Gamma \vdash \mu F \ni t_2}{\Gamma \vdash \mathit{mcovi} \cdot P\ t_1\ t_2 \in P\ t_2}$$

$$\begin{array}{rcl} |\mathit{mcovi} \cdot P\ t_1\ (\mathit{in}\ t_2')| & \dashrightarrow^* & |t_1 \cdot \mu F\ \text{-}(\mathit{castRefl} \cdot \mu F)\ \text{-}(\mathit{intrView}\ \beta\{\mathit{out}\}\ \text{-}\mathit{out}\ \text{-}\beta) \\ & & \quad (\lambda\, x.\, \mathit{mcovi} \cdot P\ t_1\ x)\ t_2'| \\ & = & |t_1|\ (\lambda\, x.\, |\mathit{mcovi}|\ |t_1|\ x)\ |t_2'| \end{array}$$

Figure 3.13: Course-of-values induction

3.4.1 Typing

When we generalize course-of-values recursion to dependent types, the typing of our recursive combinator understandably gets more complicated. We consider each of the parts.

- Instead of a type $T : \star$, we now have a predicate $P : \mu F \to \star$. This is the predicate we want to prove (using induction) holds for all inhabitants of the datatype μF.
- Term t_1 describes one step of the inductive proof. As its type is now much busier, we abbreviate it to $Alg \cdot P$, with the definition of *Alg* shown above the inference rule. Like before, t_1 has a type parameter $R : \star$ which we interpret as the subtype of μF for which it is safe to invoke the induction hypothesis.
- Where before we had a function of type $R \to \mu F$ (the "reveal" operation) as an argument to t_1, we now have evidence $c : \mathit{Cast} \cdot R \cdot \mu F$ of a *type inclusion* from R to the datatype μF. Since the eliminator *elimCast* requires only an erased

witness of the type inclusion, we make c an erased parameter to t_1.

- Similarly, where we had a function of type $R \to F \cdot R$ (the "abstract out" operation), we now have a proof that *out* can be assigned the type $R \to F \cdot R$. This means that when we are writing a particular t_1 for some proof, we know that the parameter that acts like an R-preserving destructor *really is* the datatype destructor. Again, since the eliminator *elimView* requires only an erased witness of the type assignment, we make this *View* argument an erased parameter to t_1.
- Our handle for recursion has type $\Pi\, x : R.\, P$ (*elimCast -c x*), which says that the induction hypothesis holds for all predecessors of type R. Observe that we use the type inclusion witness c to state the inductive hypothesis, as the domain of P is μF, not R.
- Finally, in the type of t_1 we name the F-collection of R-predecessors *xs* so that we may speak of the predicate P holding for the successor value constructed from it. Now, since the domain of *in* is $F \cdot \mu F$, we need to be able to cast the collection of predecessors to this type. This is only possible if the signature of the datatype, $F : \star \to \star$, is *monotonic* with respect to the preorder of type inclusions. Thus, when we elaborate a datatype declaration, we will need to produce evidence $mono : Mono \cdot F$ that F is monotonic.

3.4.2 Computation

We have not yet given an implementation for combinator *mcovi* (for this, see Section 5.6), so for now we only describe reduction behavior desired of it. While

computation in CDLE is only defined for the erasures of terms, much of what is interesting about the combinator lies in how its erased parameters are instantiated. We therefore show these implicit arguments in the computation rule for *mcovi*. It is important to remember, however, that these arguments *are erased*, and play no computational role. In particular, the presence of the annotations obscures the fact that the computational behavior of *mcovi* is in fact the same as *miter* (Figure 2.14), **not** *mcovr*.

With the annotations present, we see how the implicit parameters of t_1 can be instantiated. With type parameter R instantiated as μF, the obligation for a type inclusion becomes $Cast \cdot \mu F \cdot \mu F$, which holds by reflexivity of type inclusions *castRefl*. Similarly, we next need to give a proof that *out* can be assigned the type $\mu F \to F \cdot \mu F$ — this is trivial, since it is assumed that type inference already confirms that *out* has this type. The remainder of the arguments to t_1 are what we expect from the typing rules for combinators we discussed in Chapter 2: the handle for the induction hypothesis recursively uses the combinator *mcovi* and t_1, and the parameter for the predecessors is instantiated to t_2', the predecessors guarded by the constructor *in* on the left-hand side of the comptuation rule.

Type Inference and Conversion

Before concluding this section, we make a final remark about the annotations involved in the computation law. By the typing of *mcovi*, the annotated expression on the left-hand side of the computation has type

$$P\ (in\ t_2')$$

and by the typing of t_1, the annotated expression directly to the right of the reduction has type

$$P\ (in\ (elimCast\ \text{-}(mono\ (castRefl \cdot \mu F))\ t_2'))$$

and these types are not exactly the same. Instead, they are *convertible*. Recalling Figure 3.6 for type constructor convertibility, we see that corresponding terms appearing in type expressions are checked for $\beta\eta$-equivalence modulo erasure. By the erasure rules of *elimCast* and implicit products, type inference will be left with checking (and easily confirming) the following convertibility condition.

$$|in|\ |t_2'| =_{\beta\eta} |in|\ ((\lambda\, x.\, x)\ |t_2'|)$$

3.5 Chapter Conclusion

This chapter gave a deep dive into the theory and meta-theory of CDLE. Since elaboration gives meaning to a surface language via translation to a target language, a robust understanding of the target language is the *sine qua non* of this dissertation. CDLE's system of erased typing annotations is a synthesis of its philosophical commitment to an extrinsic ontology of types and a preferred semantic model with the pragmatic need for syntax-directed type inference, and understanding this illuminates the principles underlying the laws for the type constructs with which it extends CC. Meta-theoretically, CDLE guarantees logical consistency and a qualified guarantee of termination for terms. Though the presence of the qualification makes CDLE's termination guarantee weaker than many other ITPs' kernel languages, its chief causes, the φ axiom and the Kleene trick, are also the source of some of CDLE's most potent facilities — the ability to internalize the notions of type inclusion (*Cast*) and type

assignment (*View*). Both of these play central roles in the elaboration of inductive types, as we will see in subsequent chapters.

History of CDLE

The *Calculus of Dependent Lambda Eliminations*, first proposed by Stump [80, 79], aims to be a compact kernel theory for ITPs. It achieves this compactness by being a pure typed lambda calculus. In particular, it has no primitive constructs for inductive definitions, opting instead to represent these using lambda encodings. The history of lambda representations for datatypes is rich, starting with the origin of the lambda calculus itself with what is now called the *Church encoding* of datatypes [20]. Many other lambda encodings followed in its wake — the *Scott encoding* [73], *Parigot encoding* [69], the *Mendler encoding* [63], and the *Stump-Fu encoding* [82] — as did the project of grounding inductive definitions in such encodings, examples of which include the work of Böhm and Berarducci [14] on the translation of a family of ADTs to Church encodings in System F, the logical frameworks of Parigot [69, 53], and Coquand and Huet's *Calculus of Constructions* [25].

Unfortunately for this project, Geuvers [37] showed that induction was not derivable in second-order dependent type theory, a subsystem of CC. Meanwhile, the success of the proposal by Werner [95] to extend CC with primitives for inductive types, called the *Calculus of Inductive Constructions*, and the successes of its descendants, such as Luo's *Extended Calculus of Constructions* [56] and the *Calculus of (Co)Inductive Constructions* (the basis of Coq's kernel theory [87]), and languishment of lambda encodings of datatypes, resulted in the dominant view that lambda

encodings were ultimately unsuitable for representing inductive types.

Stump [80] challenged this status quo, deriving induction in second-order dependent type theory extended with three new type constructs and renewing interest in lambda encodings. Subsequent work by Firsov and Stump [36] and Firsov et al. [34] paved the way for this dissertation, deriving induction for a broad familiy of inductive types *generically*, that is, parametric in the signature of the datatype. The key idea behind Stump's original derivation, attributed to Leivant [55], is that "Church-encoded numbers realize their own typings." CDLE can be seen as the end result of a search for a kernel theory with the ability to reason about its own type assignments.

NuPRL and ICC

Of the influences on the design of CDLE, two type theories in particular deserve special mention: Miquel's *Implicit Calculus of Constructions* [65] (ICC) and the NuPRL proof assistant of Constable et al. [23], the latter of which in particular shares CDLE's commitment to an extrinsic ontology of types. The obvious influences are, of course, the direct use of certain type constructs and axioms (implicit products from ICC, dependent intersections and φ from NuPRL). CDLE's system of erased type annotations can be traced back to the work of Barras and Bernardo [9] which reformulates Miquel's original type assignment presentation of ICC to this setup.

The notion of subtyping as type inclusion is also readily apparent in both works, though only meta-linguistically: Miquel defined subtyping in terms of the typing judgement,

$$\Gamma, x : S \vdash x : T$$

and in NuPRL, it can be defined as

$$\forall\, x : S.\, x \in T$$

where $x \in T$ is, again, a judgement. The NuPRL team recognized the major deficiency of this notion of subtyping (see Crary [26]): as a judgement, it is not a first-class concept of the language and cannot be an antecedent of a proof carried out within the theory. Concretely, this makes it unsiutable for the uses to which we will (and have already) put *Cast*, namely, in the formulation of the course-of-values induction principle and in defining a notion of positivity for type constructors.

Course-of-values Induction

The original formulation of course-of-values induction in CDLE, due to Firsov et al. [35], differs from the version presented in this chapter (see Figure 3.13). In particular, in ibid. terms playing the role of t_1 have the type listed below.

$$\begin{aligned}&\forall\, R : \star.\, \forall\, c : \mathit{Id} \cdot R \cdot \mu F.\, \Pi\, o : R \to F \cdot R.\, \{o \simeq \mathit{out}\} \Rightarrow \\ &\quad (\Pi\, x : R.\, P\ (\mathit{elimId}\ \text{-}c\ x)) \to \Pi\, xs : F \cdot R.\, P\ (\mathit{elimId}\ \text{-}(\mathit{imap}\ \text{-}c)\ xs)\end{aligned}$$

The first difference concerns the nature of the type inclusion of bound variable R to the datatype μF, with *Id*, *elimId*, and *imap* replaced by *Cast*, *elimCast*, and *mono* in the present formulation. This is more than just a renaming: Firsov et al. use a different (though equivalent in most respects) notion of type inclusion formulated by Diehl et al. [31] referred to as "identity function[s] in a Curry-style theory".

More significantly, the formulation of course-of-values induction in Firsov et al. [35] has what in this dissertation is encapsulated as erased evidence of $\mathit{View} \cdot (R \to F \cdot R)\ \beta\{\mathit{out}\}$ decomposed into two arguments: a relevant argument of type $R \to F \cdot R$

and an irrelevant equality proof that this argument is equal to *out*. The impact of this difference on the elaboration of inductive types is significant, though it requires a grasp of the low-level details to appreciate. For now, we highlight two key points which we will return to in subsequent chapters.

- Combining the relevant argument $o : R \to F \cdot R$ and an irrelevant equality proof into a single *irrelevant View* argument changes the computational behavior of the combinator — recall that in Section 3.4.2 we noted that the computational behavior of *mcovi* is the same as that for Mendler-style *iteration*, **not** Mendler-style course-of-values iteration or recursion. This introduces a tension between the standard translation of the categorical semantics of course-of-values iteration into type theory and the CDLE-idiomatic form used in this dissertation, a tension we will resolve in Section 5.6 when we show how to reduce the former to the latter.
- On the more pragmatic side, the separation of the "computational" and "logical" components of course-of-values induction present in the Firsov et al. presentation affects the structure of proofs written about programs which use the combinator. The difference is this: bound variable $o : R \to F \cdot R$ is only *propositionally* equal to *out*, not definitionally, meaning that proofs may sometimes require performing a substitution using the assumed proof of equality $\{o \simeq out\}$. While this is not much trouble when working in CDLE directly, it poses a non-trivial difficulty in the user experience of the surface language, to wit, how should these different uses of the destructor be distinguished (CDLE

is Curry-style, so it cannot be by the types alone). The answer taken by this dissertation is that *they should not be*, which requires an adjustment to the underlying recursive combinator and its implementation.

CDLE is, of course, not the only setting in which course-of-values induction has been explored. Goguen et al. [41], primarily interested in showing how *dependent pattern matching* [24] can be elaborated to the dependent eliminators of datatypes, also showed how to derive course-of-values induction in their setting. Goguen et al. implement it by providing as the inductive hypothesis $Below_{\mu F}\ P\ x$, an F-branching tree containing proofs that P holds for all subdata of x. Functions analysing a static number of cases may easily make use of this, but accessing a proof for dynamically computed subdata (e.g. the result of $minusCoV$ in div, Figure 2.11) requires an inductive proof of a lemma such as $Below_{Nat}\ P\ (succ\ n) \to P\ (minus\ n\ m)$ (for any m), which is not required in Cedille. This is yet another example of the relative advantages of the Mendler style of coding recursion compared to the Squiggol style, as outlined by Uustalu and Vene [91]: the need for an additional lemma arises directly from the introduction of the intermediate structure $Below_{\mu F}\ P\ x$, which is unnecessary in the Mendler style.

CHAPTER 4
ELABORATION OF DATATYPE SIGNATURES AND THEIR POSITIVITY

> Most infuriatingly, we have some elegant models for inductive families but we seem stuck with some clumsy syntactic presentation. . . Being stuck with a syntactic artifact, the ghost of the de Bruijn criterion haunts our type theories: inductive definitions elude the type checker and must be enforced by a not-so-small positivity checker.
>
> Dagand and McBride
> *Elaborating Inductive Definitions (Technical Report)*

Semantically, inductive datatypes can be decomposed into two components: the datatype's *signature*, which gives the datatype its unique structure and proof-by-cases principle, and a *least fixedpoint operation*, which (when defined for the signature) gives the datatype its inductive nature, including a proof-by-induction principle. This chapter concerns the first of these, specifically the measures taken to ensure least fixedpoints of datatype signatures exist and the elaboration of these signatures from datatype declarations. In particular, recall that in Section 2.1.2.3 we saw that to support the standard case distinction scheme for all datatypes, we must ensure that all datatypes are *positive* to avoid introducing logical inconsistency and new sources of nontermination to the theory.

In ITPs based on theories where inductive types are axiomatized, like the *Calculus of Inductive Constructions* (CIC), the kernel theory must absorb the complexity of positivity checking. As expressed in the chapter epigraph, when the positivity criterion is in particular expressed *syntactically*, it introduces "not-so-small" complexity to the trusted computing base of the kernel theory. This runs afoul of the *de Bruijn*

criterion [38], which is the idea that ITPs should create proof objects that can be checked by independent programs that skeptical users could feasibly write themselves. When positivity checking is part of the TCB of an ITP, any implementation of the kernel that we might use as a basis for trusting proofs carried out in the ITP must also implement positivity checking.

In CDLE, inductive types are *derived* rather than axiomatized. Cedille and any other ITP that is uses CDLE as a kernel theory may, in this respect, exorcise the "ghost of the de Bruijn criterion," as all that is required is that a suitable notion of positivity, such as *Mono* (Section 3.3.3.2), be defined in CDLE, and that the ITP is able to generate this evidence for the signature of an admitted datatype declaration.

In this chapter, we demonstrate how Cedille achieves this. In particular, we describe the elaboration of non-parameterized, non-indexed datatypes. While the gap between this and datatypes with parameters and indices is conceptually not so great, technically it is significantly more complex. The full generalization of positivity checking and datatype signature elaboration, as well as pattern matching and recursion, is left as future work.

4.0.1 About This Chapter

Prospectus. We present the inference rules for the evidence-producing positivity checker for datatype signature in Section 4.1. Section 4.2 presents the rules for elaborating signatures from datatype declarations, and it is this section that comprises the greatest share of the present chapter, both in bulk and in technical complexity.

Before we present signature elaboration, we formally describe in Section 4.2.1

datatype declarations and the notions of *telescope* and *sequence* used to define them. We also take some space (Sections 4.2.2 and 4.2.4) to demonstrate the principles behind the elaboration of datatype signatures using the much simpler case of binary coproducts before showing the inference rules in their full generality (Section 4.2.3 and 4.2.5).

For both positivity checking and signature elaboration, we prove, up to some assumptions, that the set of inference rules defining them are algorithmic (Proposition 4.5 and Theorem 4.6 for positivity checking, and Proposition 4.53 and Theorem 4.54 for signature elaboration), and sound with respect to the target language (Theorem 4.4 for positivity checking and Theorem 4.52 for signature elaboration). We also show that the elaborated signatures and their constructors admit the expected dependent eliminator (Theorem 4.56), which we will need later to elaborate pattern matching in σ- and μ-expressions in the next chapter.

Reading Guide. For all readers, the following is **strongly recommended:**

- *at least skim* Section 4.1 to understand the big picture behind positivity checking and subtyping;
- *read closely* Theorems 4.3 and 4.4, which express soundness of the positivity checker;
- *read closely* Section 4.2.1 to understand the definitions of telescopes, sequences, and datatype declarations, as these are ubiquitous in the formalisms presented in this and the next chapter;
- *read closely* Sections 4.2.2 and 4.2.4, which describe the impredicative encoding

$$\boxed{\Gamma^{-} \vdash +F^{-} \rightsquigarrow\ t^{+}}$$

$$\frac{\begin{array}{c} F \dashrightarrow^{*} \lambda\, R{:}\star.\, T \\ \Gamma, R_1{:}\star, R_2{:}\star, z{:}\mathit{Cast} \cdot R_1 \cdot R_2 \vdash T[R_1/R] \leqslant_{\mathit{elimCast}\ \text{-}z} T[R_2/R] \rightsquigarrow t \end{array}}{\Gamma \vdash +F \rightsquigarrow \Lambda\, R_1.\, \Lambda\, R_2.\, \Lambda\, z.\, \mathit{intrCast}\ \text{-}t\ \text{-}(\lambda\, x.\, \beta)}$$

Figure 4.1: Positivity checking

of coproducts, as this captures the essence of datatype signature elaboration; and

- *read closely* Theorems 4.52 and 4.56, which express soundness of signature elaboration and soundness of the signature's dependent eliminator.

Original Contributions. The evidence producing positivity checker and elaboration of datatype signatures are original work. Earlier formulations of these first appeared in an unpublished manuscript by Jenkins et al. [46].

4.1 Evidence Producing Positivity Checker

Earlier, in Section 3.3.3, we presented an internal notion of type inclusion, *Cast*, and showed that it induces a preorder on types. This allowed us to define *monotonicity* of datatype signatures with respect to this preorder. Then, in Section 3.4 we formulated a CDLE-idiomatic combinator for course-of-values induction that requires evidence that the datatype signature is monotonic. Now, in this section we show how such monotonicity witnesses can be systematically produced from a positive datatype signature.

The single inference rule for the judgment $\Gamma \vdash +F \rightsquigarrow t$ for positivity checking

is given in Figure 4.1, which we may read "under context Γ, type constructor F is monotonic as witnessed by t" (for guidance on how to read inference rules for judgments, see Section 3.1.2). This rule, which only applies when F can be reduced to weak head normal form $\lambda\, R : \star.\, T$, appeals to the auxiliary judgment for checking subtyping given evidence of a type inclusion, listed in Figure 4.2.

To aid the reader in understanding this inference rule for positivity, we restate below the definition of *Mono*.

```
Mono : (⋆ → ⋆) = ⋆
= λ F: ⋆ → ⋆. ∀ X: ⋆. ∀ Y: ⋆. Cast ·X ·Y ⇒ Cast ·(F ·X) ·(F ·Y) .
```

To produce evidence of $\mathit{Mono} \cdot F$, in the premise of the inference rule we add to the typing context fresh type variables R_1 and R_2 such that the first is included into the second — $z : \mathit{Cast} \cdot R_1 \cdot R_2$ — and must prove $\lambda\, R : \star.\, T$ respects that type inclusion. The retyping function (see Section 3.3.3) we give to the subtyping judgment is *elimCast* $-z : R_1 \to R_2$ and what we expect to get back is a retyping function: $t : T[R_1/R] \to T[R_2/R]$ with $|t| =_{\beta\eta} \lambda\, x.\, x$.

4.1.1 Subtyping

Read the judgments $\Gamma \vdash S \leqslant_s T \rightsquigarrow t$ and $\Gamma \vdash S \leqslant_s^{\mathbf{n}} T \rightsquigarrow t$ as "under context Γ and base subtyping assumption s, S is a subtype of T as witnessed by t" (the second judgment is for types in weak head normal form). Here, input s is assumed to be a retyping function between type variables and output t is ensured to be retyping function from S to T derived from s. The invariant we maintain is that $\Gamma \vdash S \to T \ni t$ and if $|s| =_{\beta\eta} \lambda\, x.\, x$, then so too does $|t|$. To check that S is a subtype of T, we first reduce both types to weak head normal form (shown in the first rule of

$$\boxed{\Gamma^- \vdash S^- \leqslant_{s^-} T^- \rightsquigarrow t^+}$$

$$\frac{S \dashrightarrow^* S' \not\dashrightarrow \quad T \dashrightarrow^* T' \not\dashrightarrow \quad \Gamma \vdash S' \leqslant_s^{\mathbf{n}} T' \rightsquigarrow t}{\Gamma \vdash S \leqslant_s T \rightsquigarrow t}$$

$$\boxed{\Gamma^- \vdash S^- \leqslant_{s^-}^{\mathbf{n}} T^- \rightsquigarrow t^+}$$

$$\frac{}{\Gamma \vdash X \leqslant_s^{\mathbf{n}} X \rightsquigarrow \lambda\, x.\, x} \qquad \frac{R_1 \neq R_2 \quad \Gamma \vdash s \in R_1 \to R_2}{\Gamma \vdash R_1 \leqslant_s^{\mathbf{n}} R_2 \rightsquigarrow s}$$

$$\frac{S\ t \cong T\ t'}{\Gamma \vdash S\ t \leqslant_s^{\mathbf{n}} T\ t' \rightsquigarrow \lambda\, x.\, x} \qquad \frac{S_1 \cdot S_1 \cong T_1 \cdot T_2}{\Gamma \vdash S_1 \cdot S_2 \leqslant_s^{\mathbf{n}} T_1 \cdot T_2 \rightsquigarrow \lambda\, x.\, x}$$

$$\frac{\{s_1 \simeq s_2\} \cong \{t_1 \simeq t_2\}}{\Gamma \vdash \{s_1 \simeq s_2\} \leqslant_s^{\mathbf{n}} \{t_1 \simeq t_2\} \rightsquigarrow \lambda\, x.\, x}$$

$$\frac{\kappa_1 \cong \kappa_2 \quad \Gamma, X : \kappa_1 \vdash T_1 \leqslant_s T_2 \rightsquigarrow s_2}{\Gamma \vdash \forall\, X : \kappa_1.\, T_1 \leqslant_s^{\mathbf{n}} \forall\, X : \kappa_2.\, T_2 \rightsquigarrow \lambda\, f.\, \Lambda\, X.\, s_2'\ (f \cdot X)}$$

$$\frac{\Gamma \vdash S_2 \leqslant_s S_1 \rightsquigarrow s_1 \quad \Gamma, x : S_2 \vdash T_1[s_1'\ x/x] \leqslant_s T_2 \rightsquigarrow s_2}{\Gamma \vdash \Pi\, x : S_1.\, T_1 \leqslant_s^{\mathbf{n}} \Pi\, x : S_2.\, T_2 \rightsquigarrow \lambda\, f.\, \lambda\, x.\, s_2'\ (f\ (s_1'\ x))}$$

$$\frac{\Gamma \vdash S_2 \leqslant_s S_1 \rightsquigarrow s_1 \quad \Gamma, x : S_2 \vdash T_1[s_1'\ x/x] \leqslant_s T_2 \rightsquigarrow s_2}{\Gamma \vdash \forall\, x : S_1.\, T_1 \leqslant_s^{\mathbf{n}} \forall\, x : S_2.\, T_2 \rightsquigarrow \lambda\, f.\, \Lambda\, x.\, s_2'\ (f\ \text{-}(s_1'\ x))}$$

$$\frac{\Gamma \vdash S_1 \leqslant_s S_2 \rightsquigarrow s_1 \quad \Gamma, x : S_1 \vdash T_1 \leqslant_s T_2[s_1'\ x/x] \rightsquigarrow s_2}{\Gamma \vdash \iota\, x : S_1.\, T_1 \leqslant_s^{\mathbf{n}} \iota\, x : S_2.\, T_2 \rightsquigarrow \lambda\, x.\, [s_1'\ x.1, s_2'[x.1/x]\ x.2]}$$

Figure 4.2: Subtyping

Figure 4.2), then proceed depending on the outermost type construct used to form the two resulting types (if the weak head normal forms of S and T are formed from different type constructs, we fail to assert the subtyping judgments).

Remark 4.1. The subtyping judgments produce as outputs terms t that may be formed by λ- and Λ-abstractions. When this is the case, we assume that the bound variables of t are chosen so that they are *fresh* with respect to the given context.

Notation 4.2. In the rules of Figure 4.2, we adopt the following convention. Whenever there is a unique premise $\Gamma \vdash S \leqslant_s T \rightsquigarrow s_i$, we let $s_i' =_{\mathbf{df}} \chi\ (S \to T)$ - s_i in the subsequent premises and the output of the conclusion.

We now describe the inference rules for judgment $\Gamma \vdash S \leqslant_s^{\mathbf{n}} T \rightsquigarrow t$.

- Our base cases are variables, neutral application spines, and equalities. Identical variables, applications, and equalities are related when they are convertible, and the retyping function is $\lambda\, x.\, x$. Distinct type variables, R_1 (contravariantly) and R_2 (covariantly), are in the subtyping relation when the base subtyping assumption s has type $R_1 \to R_2$, and the output retyping function is s (the premise $R_1 \neq R_2$ ensures that the subtyping rules are syntax-directed).
- For polymorphic types, $\forall\, X : \kappa_1.\, T_1$ is a subtype of $\forall\, X : \kappa_2.\, T_2$ when κ_1 and κ_2 are convertible and T_1 is a subtype of T_2 witnessed by some retyping function s_2. The retyping function that results from the rule abstracts over a type constructor X then applies s_2 to the result of polymorphic instantiation with that variable.

  CDLE's bidirectional typing judgments introduce a subtle issue: our invariant allows us to assume that s_2 may be *checked* against the type $T_1 \to T_2$, but where we would wish to use it in the conclusion of the rule for type constructor

polymorphism (and in the subsequent rules) is at the head of an application, which requires that s_2 *synthesizes* this type. We therefore ascribe to s_2 the appropriate type annotation and alias this expression (whose type we can now synthesize) as s_2' (see Notation 4.2).

- The rules for dependent products and implicit products are similar, so we only detail the former. Type $\Pi\, x : S_1.\, T_1$ is a subtype of $\Pi\, x : S_2.\, T_2$ when S_2 is a subtype of S_1 (note the contravariance!), witnessed by s_1, and $T_1[s_1'\ x/x]$ is a subtype of T_2 under a context extended with fresh variable $x : S_2$, witnessed by s_2. For the retyping function generated in the conclusion of the rule, given $f : \Pi\, x : S_1.\, T_1$, we precompose with s_1' (obtained by annotating s_1) to obtain an expression of type $\Pi\, x : S_2.\, T_1[s_1'\ x/x]$, then postcompose with s_2' (obtained by annotating s_2) to obtain an expression of type $\Pi\, x : S_2.\, T_2$, as desired.
- Dependent intersections are covariant in both types, so to show that $\iota\, x : S_1.\, T_1$ is a subtype of $\iota\, x : S_2.\, T_2$, we must have S_1 is a subtype of S_2 with coercion s_1, and T_1 is a subtype of $T_2[s_1'\ x/x]$ (in a context extended by fresh variable $x : S_1$) with coercion s_2. The retyping function that results from this rule rebuilds the dependent intersection after using s_1' and s_2' to retype its projections. Note that in order for this coercion to be well typed, we must have that $|s_1| =_{\beta\eta} \lambda\, x.\, x$ and $|s_2| =_{\beta\eta} \lambda\, x.\, x$ (see the introduction rule for dependent intersections in Figure 3.7 and the erasure rule for χ in Figure 3.8).

4.1.2 Properties

Our first property of interest for positivity checking is that it is *sound*, which is to say that if $\Gamma \vdash +F \rightsquigarrow t$ then t really does witness the monotonicity of F, i.e., $\Gamma \vdash \mathit{Mono} \cdot F \ni t$. To confirm this, we first prove soundness for the subtyping judgment. As we will see throughout the remainder of this dissertation, soundness consists of two parts, *static* and *dynamic*. For our subtyping judgment, static soundness means that the retyping functions it produces have the expected type, and dynamic soundness means the produced functions are convertible modulo erasure with $\lambda\, x.\, x$.

The formal statement of soundness is given below in Theorem 4.3, and the proof is given in Appendix A.1.

Theorem 4.3 (Static and dynamic soundness of subtyping)**.**
For all Γ, S, T, *and* s *with* $|s| =_{\beta\eta} \lambda\, x.\, x$, *if*

- $\vdash \Gamma$ *and* $\Gamma \vdash s \in R_1 \to R_2$, *and*
- $\Gamma \vdash S : \star$ *and* $\Gamma \vdash T : \star$,

then we have the following:

1. *if* $\Gamma \vdash S \leqslant_s T \rightsquigarrow t$ *then* $\Gamma \vdash S \to T \ni t$ *with* $|t| =_{\beta\eta} \lambda\, x.\, x$; *and*
2. *if* $\Gamma \vdash S \leqslant_s^{\boldsymbol{n}} T \rightsquigarrow t$ *then* $\Gamma \vdash S \to T \ni t$ *with* $|t| =_{\beta\eta} \lambda\, x.\, x$.

Soundness of the positivity checker follows as a corollary to Theorem 4.3.

Corollary 4.4. *For all* Γ *and* F *with* $\vdash \Gamma$ *and* $\Gamma \vdash F : \star \to \star$, *if* $\Gamma \vdash +F \rightsquigarrow t$ *then* $\Gamma \vdash \mathit{Mono} \cdot F \ni t$ *with* $|t| =_{\beta\eta} \lambda\, x.\, x$.

The inference rules for positivity checking and subtyping are meant to specify a *semi-algorithm* for an implementation of an ITP, so in addition to soundness we also wish to establish that these rules are *syntax-directed* and *mode-correct*, and for the judgment these rules comprise we desire that, under certain assumptions, it is *decidable* (for the proof of this last property, see Appendix A.2).

Proposition 4.5. *The inference rules comprising judgments* $\Gamma \vdash +F \rightsquigarrow t$, $\Gamma \vdash S \leqslant_s T \rightsquigarrow t$, *and* $\Gamma \vdash S \leqslant_s^{\boldsymbol{n}} T \rightsquigarrow t$ *are mode-correct and syntax-directed.*

The qualification we must make in stating decidability of the positivity and subtyping judgment is that the terms present in the subjects of these judgments are normalizing. This is because some of the inference rules for subtyping require checking type convertibility, which in turn may require checking convertibility of non-normalizing terms.

Theorem 4.6 (Decidability)**.**

- *If* $\vdash \Gamma$ *and* $\Gamma \vdash F : \star \to \star$ *and* F *is terminating under full* β*-reduction for types and* $\beta\eta$*-reduction for terms, then it is decidable whether* $\Gamma \vdash +F \rightsquigarrow t$ *for some* t*.*
- *If* $\vdash \Gamma$ *and* $\Gamma \vdash S : \star$ *and* $\Gamma \vdash T : \star$ *and* S, T *are terminating under full* β*-reduction for types and* $\beta\eta$*-reduction for terms and* $\Gamma \vdash s \in R_1 \to R_2$ *with* $|s| = \lambda\, x.\, x$*, then:*
  1. *it is decidable whether* $\Gamma \vdash S \leqslant_s T \rightsquigarrow t$ *for some* t*; and*
  2. *it is decidable whether* $\Gamma \vdash S \leqslant_s^{\boldsymbol{n}} T \rightsquigarrow t$ *for some* t*.*

4.2 Datatype Signatures

In the previous section, we saw an algorithm for producing evidence that a given type constructor $F : \star \to \star$ is positive. However, at this stage it is not yet clear where this F comes from and what relation it has with the Cedille syntax for datatype declarations. This is the subject we address in the present section. First, we give a formal definition of datatype declarations in Section 4.2.1. This then becomes, in Sections 4.2.3 and 4.2.5 the subject of judgments whose inference rules elaborate the datatype signature and its constructors to CDLE expressions.

Elaboration of datatype signatures is separated into these two subsections to highlight two separate concerns, mirroring the original recipe by Stump [80] for

deriving inductive encodings in CDLE. In Section 4.2.3, we are concerned with the elaboration of the "computational" component of the datatype signature, which is to say we are only elaborating the parts needed for the non-dependent case distinction scheme of the signature. In Section 4.2.5, we elaborate the "inductive" component of the datatype signature, used to justify a fully dependent proof-by-cases scheme, and combine the two using dependent intersections.

As always, we are concerned that this elaboration is sound with respect to the static and dynamic semantics of the target language CDLE. Concerning static soundness, we will give theorems for each set of elaboration rules we discuss. Since we are not yet introducing σ- and μ-expressions (see Chapter 5), we cannot yet make a direct statement of soundness of the dynamic semantics of the surface language with respect to the target language. However, we can and do formulate the dependent eliminators of elaborated signatures and show that these satisfy the expected typing and computation laws (Theorem 4.56), which will in turn assist us in proving dynamic soundness for σ- and μ-expressions.

4.2.1 Datatype Declarations

Notation 4.7. In the remainder of this dissertation, we extend our notational conventions as follows.

- We use meta-variable D and its primed and subscripted variants to range over type variables. D is intended specifically to refer to the name of a user-declared datatype, though we will see it used also in type constructor quantifications and abstractions in elaborated expressions.
- We use meta-variable c and its primed and subscripted variants for term identifiers. Such meta-variables are intended to range over the identifiers for constructors in a user-declared datatype, though we will see it used also in telescopes and term quantifications and abstractions in elaborated expressions.
- We use capital greek letters Δ, Ξ, Θ for telescopes.
- We use lowercase greek letters $\tau, \gamma, \xi, \theta$ for sequences.

4.2.1.1 Telescopes and Sequences

Telescopes and sequences are essential for reasoning generically about datatypes in dependent type theories, so we dedicate a significant portion of our discussion to their definitions, operations, and inference rules for well-formedness.

Definition 4.8 (Telescopes)**.** A *telescope* is a generalization of a typing context. Formally, a telescope Δ is generated by the grammar.

$$\Delta \ ::= \ () \mid (x\!:\!T)\Delta \mid (\text{-}\ x\!:\!T)\Delta \mid (X\!:\!\kappa)\Delta$$

That is to say, telescopes are either empty or formed by prepending to a telescope an unerased or erased term variable or a type constructor variable. When a given telescope is nonempty, we omit the trailing (), i.e., we write $(z\!:\!Nat)(s\!:\!Nat \to Nat)$ for $(z\!:\!Nat)(s\!:\!Nat \to Nat)()$. We write $\#\Delta$ to denote the number of entries in Δ, $\Delta(i)$ for the ith entry of Δ (when $1 \leq i \leq \#\Delta$) and write $\Delta_1\Delta_2$ for the concatenation of telescopes Δ_1 and Δ_2.

For the remainder of this dissertation, it will be convenient for us to define a telescope in terms of a family of identifiers and types.

Notation 4.9 (Parallel definition of a telescope)**.** Let $(x_i, T_i)_{i\in\{1\ldots n\}}$ be a family of pairs of unerased term identifiers x_i and types T_i, where n is a natural number. We introduce the following definition form that gives each entry of the telescope simultaneously in terms of this family.

$$\underset{i\leq n}{\Delta(i)} =_{\mathbf{df}} (x_i\!:\!T_i)$$

This definition form is to be understood as being the same as the following.

$$\Delta =_{\mathbf{df}} (x_1\!:\!T_1)\ldots(x_n\!:\!T_n)$$

Well-formedness for telescopes

Figure 4.3 gives the CDLE inference rules for judgment $\Gamma \vdash \Delta$, which we read "under Γ, Δ is a well-formed telescope." These rules are similar to the rules for context formation given in Figure 3.3. The empty telescope is always well-formed, and a non-empty telescope is well-formed when the classifier associated to the first

$$\boxed{\Gamma^- \vdash \Delta^-}$$

$$\frac{}{\Gamma \vdash ()} \qquad \frac{\Gamma \vdash T : \star \quad \Gamma, x{:}T \vdash \Delta}{\Gamma \vdash (x{:}T)\Delta}$$

$$\frac{\Gamma \vdash T : \star \quad \Gamma, x{:}T \vdash \Delta}{\Gamma \vdash (\text{-}\ x{:}T)\Delta} \qquad \frac{\Gamma \vdash \kappa \quad \Gamma, X{:}\kappa \vdash \Delta}{\Gamma \vdash (X{:}\kappa)\Delta}$$

Figure 4.3: Well-formedness for telescopes

entry of the telescope is well-formed and the remainder of the telescope is well-formed under a context extended by the first entry.

When compared to typing contexts, telescopes have an additional construction for tracking whether a term variable is meant to be erased or unerased. This is to facilitate the following definitions.

Definition 4.10 (Telescope quantification and abstraction)**.**

- For types, we define $\Pi\,\Delta.\,T$ as follows.

$$\begin{array}{rcl} \Pi\,().\,T & = & T \\ \Pi\,(x{:}T')\Delta.\,T & = & \Pi\,x{:}T'.\,\Pi\,\Delta.\,T \\ \Pi\,(\text{-}\ x{:}T')\Delta.\,T & = & \forall\,x{:}T'.\,\Pi\,\Delta.\,T \\ \Pi\,(X{:}\kappa)\Delta.\,T & = & \forall\,X{:}\kappa.\,\Pi\,\Delta.\,T \end{array}$$

- For kinds, we define $\Pi\,\Delta.\,\kappa$ as follows.

$$\begin{array}{rcl} \Pi\,().\,\kappa & = & \kappa \\ \Pi\,(x{:}T')\Delta.\,\kappa & = & \Pi\,x{:}T.\,\Pi\,\Delta.\,\kappa \\ \Pi\,(\text{-}\ x{:}T')\Delta.\,\kappa & = & \Pi\,x{:}T.\,\Pi\,\Delta.\,\kappa \\ \Pi\,(X{:}\kappa')\Delta.\,\kappa & = & \Pi\,X{:}\kappa'.\,\Pi\,\Delta.\,\kappa \end{array}$$

- For terms, we define $\lambda\,\Delta.\,t$ as follows.

$$\begin{array}{rcl} \lambda\,().\,t & = & t \\ \lambda\,(x{:}T)\Delta.\,t & = & \lambda\,x.\,\lambda\,\Delta.\,t \\ \lambda\,(\text{-}\ x{:}T)\Delta.\,t & = & \Lambda\,x.\,\lambda\,\Delta.\,t \\ \lambda\,(X{:}\kappa)\Delta.\,t & = & \Lambda\,X.\,\lambda\,\Delta.\,t \end{array}$$

- For type constructors, we define $\lambda\,\Delta.\,T$ as follows.

$$\begin{array}{rcl} \lambda\,().\,T & = & T \\ \lambda\,(x{:}T')\Delta.\,T & = & \lambda\,x{:}T'.\,\lambda\,\Delta.\,T \\ \lambda\,(\text{-}\ x{:}T')\Delta.\,T & = & \lambda\,x{:}T'.\,\lambda\,\Delta.\,T \\ \lambda\,(X{:}\kappa)\Delta.\,T & = & \lambda\,X{:}\kappa.\,\lambda\,\Delta.\,T \end{array}$$

$$\boxed{\Gamma^- \vdash \tau^- : \Delta^-}$$

$$\frac{}{\Gamma \vdash () : ()} \qquad \frac{\Gamma \vdash T \ni t \quad \Gamma \vdash \tau : \Delta[t/x]^T}{\Gamma \vdash (t)\tau : (x{:}T)\Delta}$$

$$\frac{\Gamma \vdash T \ni t \quad \Gamma \vdash \tau : \Delta[t/x]^T}{\Gamma \vdash (\text{-}\ t)\tau : (\text{-}\ x{:}T)\Delta} \qquad \frac{\Gamma \vdash T : \kappa \quad \Gamma \vdash \tau : \Delta[T/X]}{\Gamma \vdash (\cdot\ T)\tau : (X{:}\kappa)\Delta}$$

Figure 4.4: Classification of sequences

Among other uses, telescopes classify *sequences*, which are ordered lists of terms and types. Like with telescopes, in sequences we track whether the term is intended to be erased or unerased.

Definition 4.11 (Sequences)**.** A *sequence* τ is an ordered list of terms and types. It is formally generated by the following grammar.

$$\tau ::= () \mid (t)\tau \mid (\text{-}\ t)\tau \mid (\cdot\ T)\tau$$

That is to say, sequences are either empty or formed by prepending to a sequence an unerased or erased term or a type constructor. Like with telescopes, we omit the trailing () in nonempty sequences, write $\#\tau$ for the number of entries in τ, $\tau(i)$ for the ith entry of τ (when $1 \leq i \leq \#\tau$), and write $\tau_1\tau_2$ for concatenation of sequences.

As with telescopes, it is useful to have a definition form for sequences that defines each entry simultaneously with respect to some family of terms.

Notation 4.12 (Parallel definition of a sequence)**.** Let $(t_i)_{i\in\{1\ldots n\}}$ be a family of terms, where n is a natural number. We introduce the following parallel definition form for sequences, indicating that the sequence has n entries and the ith entry (for $1 \leq i \leq n$) is t_i.

$$\underset{i\leq n}{\tau(i)} =_{\mathbf{df}} (t_i)$$

Classification of sequences

The classification rules for sequences are shown in Figure 4.4, which gives the definition of the judgment $\Gamma \vdash \tau : \Delta$, which we read as "under Γ, τ is an instantiation

of Δ." These rules use substitution on telescopes, which is defined as follows.

Definition 4.13 (Substitutions on telescopes)**.** For a given telescope Δ, variable x, term t, and type T, define $\Delta[t/x]^T$ as follows (substitution of type variables is defined similarly).

$$\begin{array}{rcll} ()[t/x]^T & = & () & \\ ((x:S)\Delta[t/x])^T & = & (x:S[t/x]^T)\Delta & \\ ((y:S)\Delta)[t/x]^T & = & (y:S[t/x]^T)\Delta[t/x]^T & \text{when } y \neq x \\ ((\text{-}\ x:S)\Delta[t/x])^T & = & (\text{-}\ x:S[t/x]^T)\Delta & \\ ((\text{-}\ y:S)\Delta)[t/x]^T & = & (\text{-}\ y:S[t/x]^T)\Delta[t/x]^T & \text{when } y \neq x \\ ((X:\kappa)\Delta)[t/x]^T & = & (X:\kappa[t/x]^T)\Delta[t/x]^T & \end{array}$$

Note that substitution does not affect the declared variables of the telescope.

We return our discussion of the classification rules for sequences. We have that:

- the empty sequence is an instantiation of the empty telescope;
- sequence $(t)\tau$ is an instantiation of $(x:T)\Delta$ when t can be checked against type T and τ is an instantiation of Δ with all free occurrences of x replaced with $\chi\ T$ - t in the classifiers of the entries of Δ;
- sequence $(\text{-}\ t)\tau$ is treated similarly to the above, but the first entry of the telescope it instantiates must also be marked erased; and
- sequence $(\cdot T)\tau$ is an instantiation of $(X:\kappa)\Delta$ when $\Gamma \vdash T : \kappa$ and τ is an instantiation of $\Delta[T/X]$.

Corresponding to the quantifications and abstractions over telescopes, for sequences we have operations for applying to them terms and types.

Definition 4.14 (Applications to sequences)**.**

- Applications of terms to sequences are written $t\ \tau$, and defined as follows.

$$\begin{array}{rcl} t\ () & = & t \\ t\ (t_1)\tau & = & (t\ t_1)\ \tau \\ t\ (\text{-}\ t_1)\tau & = & (t\ \text{-}t_1)\ \tau \\ t\ (T)\tau & = & (t \cdot T)\ \tau \end{array}$$

- Applications of types to sequences are written $T\ \tau$, and defined as follows.

$$\begin{array}{rcl} T\ ()\ & = & T \\ T\ (t)\tau & = & (T\ t)\ \tau \\ T\ (\text{-}\ t)\tau & = & (T\ t)\ \tau \\ T\ (T_1)\tau & = & (T \cdot T_1)\ \tau \end{array}$$

Finally, a telescope may be "demoted" to a sequence. To show this transformation, we explicitly define an operation *Seq* below.

Definition 4.15 (Demotion of telescopes to sequence)**.** Let Δ be a telescope, and define $Seq(\Delta)$ as follows.

$$\begin{array}{rcl} Seq(()) & = & () \\ Seq((x:T)\Delta) & = & (x)Seq(\Delta) \\ Seq((\text{-}\ x:T)\Delta) & = & (\text{-}\ x)Seq(\Delta) \\ Seq((X:\kappa)\Delta) & = & (\cdot\ X)Seq(\Delta) \end{array}$$

We omit the explicit use of *Seq* when it is clear from the context that a telescope is being treated as a sequence, such as in the expression $t\ \Delta$ (sequences, not telescopes, may be used as arguments to terms and type constructors).

We record the following fact about telescopes which are equal as sequences. This will be needed when we consider the signature elaboration rules to come.

Fact 4.16. *If* $Seq(\Delta_1) = Seq(\Delta_2)$ *then for all* t, $\lambda\,\Delta_1.\,t = \lambda\,\Delta_2.\,t$.

Subtyping

The final bit of kit for telescopes and sequences we present — telescope subtyping and pointwise sequence applications — will be needed only in Chapter 5, specifically in the typing rules for μ- and σ-expressions. Recall that the Mendler style of coding recursion (Section 2.2) introduces a type variable to stand in for recursive occurrences of the datatype in the types of constructor arguments. This poses a problem for expressing induction principles, as the predicate must be shown to hold for all constructor forms of the datatype, and the domain of these constructors is a telescope referring to the *actual* datatype, not the type variable stand-in. This mismatch

$$\boxed{\Gamma^- \vdash \Delta_1^- \leqslant_{s^-} \Delta_2^- \rightsquigarrow \xi^+}$$

$$\frac{}{\Gamma \vdash () \leqslant_s () \rightsquigarrow ()}$$

$$\frac{\Gamma \vdash S \leqslant_s T \rightsquigarrow s_1 \quad \Gamma, x\!:\!S \vdash \Delta_1 \leqslant_s \Delta_2[s'\ x/x] \rightsquigarrow \xi}{\Gamma \vdash (x\!:\!S)\Delta_1 \leqslant_s (x\!:\!T)\Delta_2 \rightsquigarrow (s_1)\xi}$$

$$\frac{\Gamma \vdash S \leqslant_s T \rightsquigarrow s_1 \quad \Gamma, x\!:\!S \vdash \Delta_1 \leqslant_s \Delta_2[s_1'\ x/x] \rightsquigarrow \xi}{\Gamma \vdash (-\ x\!:\!S)\Delta_1 \leqslant_s (-\ x\!:\!T)\Delta_2 \rightsquigarrow (-\ s_1)\xi}$$

$$\frac{\kappa_1 \cong \kappa_2 \quad \Gamma, X\!:\!\kappa_1 \vdash \Delta_1 \leqslant_s \Delta_2 \rightsquigarrow \xi}{\Gamma \vdash (X\!:\!\kappa_1)\Delta_1 \leqslant_s (X\!:\!\kappa_2)\Delta_2 \searrow (\lambda\, X\!:\!\kappa_1.\, X)\xi}$$

Figure 4.5: Telescope subtyping

is resolved by introducing subtyping rules for telescopes which produce *sequences of coercions* as evidence, shown in Figure 4.5, and applying each of these pointwise to the sequence of constructor arguments.

Definition 4.17 (Sequence pointwise application)**.** The pointwise application of sequence ξ to sequence τ, written $\xi \odot \tau$, is defined inductively as follows.

$$\begin{array}{rcl} () \odot () & = & () \\ (s)\xi \odot (t)\tau & = & (s\ t)(\xi \odot \tau) \\ (-\ s)\xi \odot (-\ t)\tau & = & (-\ (s\ t))(\xi \odot \tau) \\ (\cdot\ S)\xi \odot (\cdot T)\tau & = & (\cdot\ (S \cdot T))(\xi \odot \tau) \end{array}$$

As with the subtyping rules from Section 4.1, we desire that the elaborating telescope subtyping rules are *sound*, *algorithmic*, and *decidable* up to some assumptions.

Theorem 4.18 (Soundness of telescope subtyping)**.** *For all* $\vdash \Gamma$, $\Gamma \vdash \Delta_1$, *and* $\Gamma \vdash \Delta_2$, *and* $\Gamma \vdash s \in R_1 \to R_2$ *with* $|s| =_{\beta\eta} \lambda\, x.\, x$, *if* $\Gamma \vdash \Delta_1 \leqslant_s \Delta_2 \searrow \xi$ *for some* ξ *then*

- $\#\xi = \#\Delta_1 = \#\Delta_2$ *and for all* $i \in \{1 \ldots \#\Delta_1\}$, *if* $\Delta(i) = (x\!:\!T)$ *or* $\Delta(i) = (-\ x\!:\!T)$ *(for some* x, T*) then* $|\xi(i)| =_{\beta\eta} \lambda\, x.\, x$*; and*
- *if* $\Gamma \vdash \tau : \Delta_1$ *then* $\Gamma \vdash \xi \odot \tau : \Delta_2$

Proposition 4.19 (Mode-correctness and syntax-directedness of telescope subtyping)**.** *The judgment of Figure 4.5 for producing subtyping evidence between telescopes is mode-correct and syntax-directed.*

Proof sketch. By inspection of the inference rules. □

Theorem 4.20 (Decidability of telescope subtyping)**.** *For all* $\vdash \Gamma$, $\Gamma \vdash \Delta_1$, *and* $\Gamma \vdash \Delta_2$, *if the classifiers of the bound variables of* Δ_1 *and* Δ_2 *are terminating under full* β*-reduction, then it is decidable whether* $\Gamma \vdash \Delta_1 \leqslant_s \Delta_2 \rightsquigarrow \xi$ *for some* ξ.

Proof sketch. By a straightforward inductive argument on the structure of Δ_1 and Δ_2, appealing to Theorem 4.6 and decidability of kind conversion checking under the assumption of termination. □

4.2.1.2 Datatype Declaration Syntax

With these preliminaries, we now define a general notion of a datatype declaration.

Definition 4.21 (Datatype declaration syntax)**.** The *syntax of a datatype declaration* is given by tuple $Decl[D, \Delta]$, where D is a type variable and Δ is a telescope. The intended reading is as follows.

- D is the name of the datatype we are declaring.
- Δ gives the names and types of the datatype's constructors, which are all unerased term identifiers.

  For telescopes intended for this use, as well as the telescopes we elaborate from Δ, our convention will be to write $(c_i : \Pi\, \Delta_i.\, T) \in \Delta$ (with $1 \leq i \leq \#\Delta$) to express that $(c_i : \Pi\, \Delta_i : T.\,)$ is the ith entry of Δ.

  The Cedille syntax of a datatype declaration, which has the form

$$
\begin{array}{l}
\texttt{data}\ D\ :\ \star \\
\texttt{=}\ c_1\ :\ \Pi\, \Delta_1.\, D \\
\vdots \\
\texttt{|}\ c_n\ :\ \Pi\, \Delta_n.\, D \\
\texttt{.}
\end{array}
$$

is represented in our notation as $Decl[D, \underset{i \leq n}{\Delta(i) =_{\mathbf{df}} (c_i : \Pi\, \Delta_i.\, D)}]$ (we are here using the notation for defining telescopes introduced in Notation 4.9).

We write $\Gamma \vdash Decl[D, \Delta]$ *vs*, read "under Γ, $Decl[D, \Delta]$ is *v*alid *s*yntax for a datatype declaration," when

- the declared variables of Δ are distinct from each other,
- $(\{D\} \cup DV(\Delta)) \cap DV(\Gamma) = \emptyset$ (this is for the programmer to ensure), and
- for all $i \in \{1 \ldots \#\Delta\}$, the ith entry of Δ is of the form $(c_i : \Pi\, \Delta_i.\, D)$ where
  - $DV(\Delta_i) \cap (\{D\} \cup DV(\Delta)) = \emptyset$ (the programmer must ensure this), and
  - $DV(\Delta_i) \cap DV(\Gamma) = \emptyset$ (we may freely rename the declared variables of Δ_i to ensure this).

Example 4.22 (Declaration of natural numbers)**.** *Recall the datatype declaration for Nat, the type of natural numbers.*

```
data Sum (T1: ⋆) (T2: ⋆) : ⋆
= inl : T1 → Sum
| inr : T2 → Sum .

elimSum : ∀ T1: ⋆. ∀ T2: ⋆. ∀ X: ⋆. (T1 → X) → (T2 → X)
          → Sum ·T1 ·T2 → X
elimSum f1 f2 (inl t1) = f1 t1
elimSum f1 f2 (inr t2) = f2 t2
```

Figure 4.6: *Sum* datatype

```
data Nat : ⋆
= zero : Nat
| succ : Nat → Nat .
```

This would be represented by the tuple

$$Decl[Nat, (zero : \Pi\,().\ Nat)(succ : \Pi\,(x : Nat).\ Nat)]$$

or, equivalently,

$$Decl[Nat, (zero : Nat)(succ : Nat \to Nat)]$$

4.2.2 Impredicative Encoding of Coproducts (Part 1)

The elaboration of datatype signatures, which we begin to discuss in Section 4.2.3, features some of the most technically complex inference rules in this dissertation. This technical complexity obscures relative conceptual simplicity in the first portion of the process: all we are doing is translating the "sum-of-products" recipe of datatype signatures to impredicative encodings. Furthermore, the seemingly intimidating encodings to which we translate them, and the lambda encodings to which we translate their constructors, arise as a straightforward solution to a set of constraints on both. Therefore, before we discuss the elaboration rules in their full generality, we detour to discuss the simpler method of encoding binary coproducts.

First, let us recall the axiomatic presentation of *Sum*, shown in Figure 4.6 using

a parameterized datatype and Agda-style pseudocode for the eliminator *elimSum*. View this a set of constraints on *Sum*: fix some T_1 and T_2, let us write $\leq$ as an alias for $\to$ (merely for emphasis), and from the types we have:

$$\begin{array}{rcl} T_1 & \leq & \mathit{Sum} \cdot T_1 \cdot T_2 \\ T_2 & \leq & \mathit{Sum} \cdot T_1 \cdot T_2 \\ \mathit{Sum} \cdot T_1 \cdot T_2 & \leq & \forall\, X : \star.\, (T_1 \to X) \to (T_2 \to X) \to X \end{array}$$

where the first two inequalities come from the constructors and the last comes from *elimSum* and the type isomorphism

$$\begin{array}{rl} & \forall\, X : \star.\, (T_1 \to X) \to (T_2 \to X) \to \mathit{Sum} \cdot T_1 \cdot T_2 \to X \\ \cong & \mathit{Sum} \cdot T_1 \cdot T_2 \to \forall\, X : \star.\, (T_1 \to X) \to (T_2 \to X) \to X \end{array}$$

Of the three, the third inequality offers us a good candidate as the *solution* for $\mathit{Sum} \cdot T_1 \cdot T_2$ (it cannot be either T_1 or T_2, since no relationship between the two is assumed).

$$\mathit{Sum} =_{\mathbf{df}} \lambda\, Y : \star.\, \lambda\, Z : \star.\, \forall\, X : \star.\, (Y \to X) \to (Z \to X) \to X$$

Identifying these two types also suggests to us the following definition for *elimSum*.

$$\mathit{elimSum} =_{\mathbf{df}} \Lambda\, Y.\, \Lambda\, Z.\, \Lambda\, X.\, \lambda\, f_1.\, \lambda\, f_2.\, \lambda\, x.\, x \cdot X\ f_1\ f_2$$

That leaves us with finding a definition for the constructors $\mathit{in}_l : \forall\, Y : \star.\, \forall\, Z : \star.\, Y \to \mathit{Sum} \cdot Y \cdot Z$ and $\mathit{in}_r : \forall\, Y : \star.\, \forall\, Z : \star.\, Z \to \mathit{Sum} \cdot Y \cdot Z$. To assist us in this, we look to the computation rules for *elimSum*.

$$\begin{array}{rcl} \mathit{elimSum}\ f_1\ f_2\ (\mathit{in}_l\ t_1) & = & f_1\ t_1 \\ \mathit{elimSum}\ f_1\ f_2\ (\mathit{in}_r\ t_2) & = & f_2\ t_2 \end{array}$$

Substitute into these equations our definition for *elimSum*, and reduce to obtain

$$\begin{array}{rcl} \mathit{in}_l\ t_1\ f_1\ f_2 & = & f_1\ t_1 \\ \mathit{in}_r\ t_2\ f_1\ f_2 & = & f_2\ t_2 \end{array}$$

and now the following definitions to in_l and in_r become obvious solutions that we may readily assign the desired types.

$$\begin{array}{rcl} in_l & = & \Lambda\, Y.\, \Lambda\, Z.\, \lambda\, z_1.\, \Lambda\, X.\, \lambda\, x_1.\, \lambda\, x_2.\, x_1\ z_1 \\ in_r & = & \Lambda\, Y.\, \Lambda\, Z.\, \lambda\, z_2.\, \Lambda\, X.\, \lambda\, x_1.\, \lambda\, x_2.\, x_2\ z_2 \end{array}$$

This is the essence of the more elaborate encodings of signatures we shall see next.

4.2.3 Signature Elaboration (Computational)

Figures 4.8 and 4.10 list the elaborating inference rules for the computational part of a datatype signature. By "computational part," we mean that it elaborates types and terms sufficient for implementing a signature's non-dependent eliminator. The judgments and inference rules of Figure 4.8 are for the elaboration of the impredicative encoding of datatype signatures, and those of Figure 4.10 are for elaborating the types and lambda encodings of the signature constructors (in turn used for the lambda encodings of datatype constructors). We divide our discussion of the figures along these lines, providing examples along the way.

To orient the reader, we illustrate in Figure 4.7 the high-level information flow of the main judgments we will be discussing.

1. From the datatype declaration $\mathit{Decl}[D, \Delta]$, we elaborate the information $\mathcal{S}$ for the impredicative encoding of the datatype's signature with judgment $\Gamma \vdash_{\mathbf{sig}} \mathit{Decl}[D, \Delta] \searrow \mathcal{S}$. This corresponds to the type constructor for the signature of *Sum* we saw previously, $\lambda\, Y : \star.\, \lambda\, Z : \star.\, \forall\, X : \star.\, (Y \to X) \to (Z \to X) \to X$

2. Next, we use this information to produce the types of the signature's constructors with judgment $\Gamma \vdash_{\mathbf{ctm}} \mathcal{S} \searrow \Delta'$, where telescope Δ' associates the constructor identifiers with these types. For *Sum*, this would correspond to the

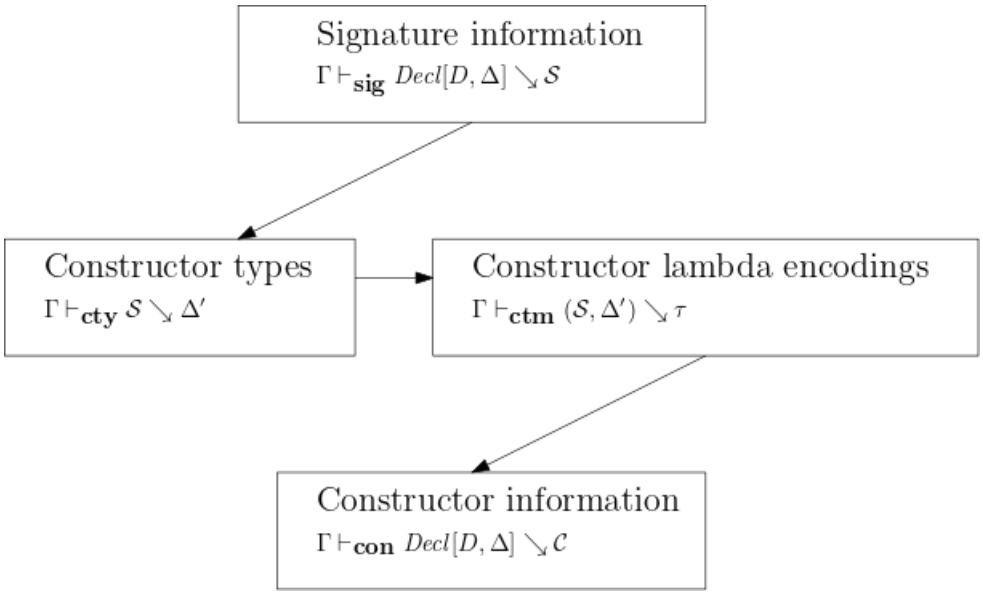


Figure 4.7: Flow of computational signature elaboration

following telescope.

$$(in_l : \forall\, Y : \star.\, \forall\, Z : \star.\, Y \to Sum \cdot Y \cdot Z)(in_r : \forall\, Y : \star.\, \forall\, Z : \star.\, Z \to Sum \cdot Y \cdot Z)$$

3. After this, with judgment $\Gamma \vdash (\mathcal{S}, \Delta') \searrow \tau$ we produce a sequence containing the lambda encodings of the signature constructors. For *Sum*, this corresponds to the following.

$$(\Lambda\, Y.\, \Lambda\, Z.\, \lambda\, z_1.\, \Lambda\, X.\, \lambda\, x_1.\, \lambda\, x_2.\, x_1\ z_1)(\Lambda\, Y.\, \Lambda\, Z.\, \lambda\, z_2.\, \Lambda\, X.\, \lambda\, x_1.\, \lambda\, x_2.\, x_2\ z_2)$$

4. Finally, the last judgment, $\Gamma \vdash_{\mathbf{con}} Decl[D, \Delta] \searrow \mathcal{C}$, merely collects all we have obtained so far into $\mathcal{C}$, the signature constructor information. This process is cumulative, so $\mathcal{C}$ contains also all the information present in $\mathcal{S}$.

We now elaborate on each of these steps.

$$\boxed{\Gamma^- \vdash_{\mathbf{sig}} (D^-, R^-, \Delta^-, i^-) \searrow (c^+, \Delta'^+)} \quad \boxed{\Gamma^- \vdash_{\mathbf{sig}} \mathit{Decl}[D, \Delta]^- \searrow \mathcal{S}^+}$$

$$\frac{(c_i : \Pi\, \Delta_i.\, D) \in \Delta \quad \Gamma, D : \star \vdash \Delta_i \searrow \Delta_i'}{\Gamma \vdash_{\mathbf{sig}} (D, R, \Delta, i) \searrow (c_i, \Delta_i'[R/D])}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{Decl}[D, \Delta]\ \mathit{vs} \quad R \notin \{D\} \cup DV(\Gamma) \cup \bigcup_{(c_i : \Pi\, \Delta_i.\, D) \in \Delta} DV(\Delta_i) \\ (\Gamma \vdash_{\mathbf{sig}} (D, R, \Delta, i) \searrow (c_i, \Delta_i'))_{i \in \{1 \ldots \#\Delta\}} \quad \underset{i \leq \#\Delta}{\Xi(i)} =_{\mathbf{df}} (c_i : \Pi\, \Delta_i'.\, D)\end{array}}{\Gamma \vdash_{\mathbf{sig}} \mathit{Decl}[D, \Delta] \searrow \mathsf{record}\ \mathit{Sig}\ \{D = D; R = R; \Xi = \Xi\}}$$

Figure 4.8: Signature elaboration (computational)

$$\boxed{\Gamma^- \vdash \Delta^- \searrow \Delta'^+}$$

$$\frac{}{\Gamma \vdash () \searrow ()} \qquad \frac{\Gamma \vdash T : \star \searrow T' \quad \Gamma, x : T \vdash \Delta \searrow \Delta'}{\Gamma \vdash (x : T)\Delta \searrow (x : T')\Delta'}$$

$$\frac{\Gamma \vdash T : \star \searrow T' \quad \Gamma, x : T \vdash \Delta \searrow \Delta'}{\Gamma \vdash (\text{-}\ x : T)\Delta \searrow (\text{-}\ x : T')\Delta'} \qquad \frac{\Gamma \vdash \kappa \searrow \kappa' \quad \Gamma, X : \kappa \vdash \Delta \searrow \Delta'}{\Gamma \vdash (\cdot\ X : \kappa)\Delta \searrow (\cdot\ X : \kappa')\Delta'}$$

Figure 4.9: Elaboration of telescopes

4.2.3.1 Impredicative Encoding of Signatures

For datatype signature elaboration, the main judgment is $\Gamma \vdash_{\mathbf{sig}} \mathit{Decl}[D, \Delta] \searrow \mathcal{S}$, which we read "under Γ, $\mathit{Decl}[D, \Delta]$ elaborates signature information $\mathcal{S}$." Rather than elaborate the signature, we elaborate a (meta-linguistic) *record* containing information sufficient to construct the signature. We do this because, as we will see when looking at the later inference rules, it is convenient to refer to components of the signature directly. This main judgment consists of a single inference rule that, after checking that the declaration is valid syntax for a datatype (see Definition 4.21), picks a type variable R that is fresh with respect to D, the declared variables of Γ,

and all bound variables in the types of the datatype constructors, then invokes the auxiliary judgment $\Gamma \vdash_{\mathbf{sig}} (D, R, \Delta, i) \searrow (c, \Delta')$ for each entry of the constructor telescope Δ. The auxiliary judgment, in turn, elaborates the parameter telescope of the ith constructor in Δ, replacing all free occurrences of D with R.

Figure 4.9 shows the inference rules comprising the judgment $\Gamma \vdash \Delta \searrow \Delta'$ for elaborating telescopes. This in turn appeals to judgments we will see in Chapter 5: elaboration of type constructors with $\Gamma \vdash T : \kappa \searrow T'$, and elaboration of kinds with $\Gamma \vdash \kappa \searrow \kappa'$.

Remark 4.23. Note that at this point of our development, we are introducing a collection of judgments that are *mutually* inductively defined: the inference rule for the elaboration of signatures invokes the judgment for elaborating telescopes, whose inferene rules invoke elaboration of type constructors and kinds, whose rules may in turn invoke the elaboration of the signature of another datatype. Rather than presenting these inference rules all at once, we opt to tackle them in smaller, digestible portions.

Of course, this explanation is not complete without saying more concerning the record object that the judgment elaborates.

Definition 4.24 (Computational signature)**.** In the meta-theory, define *Sig* as the set of records with fields D, R, and Ξ, where D and R are type variables and Ξ is a telescope whose only entries are unerased term variables and their classifying types. Meta-variable $\mathcal{S}$ ranges over members of *Sig*, and accessing the fields is written with $\mathcal{S}.D$, $\mathcal{S}.R$, and $\mathcal{S}.\Xi$ (this associates to the left and has highest precendence among all operations).

For a given $\mathcal{S}$, we obtain the computational signature with $sig(\mathcal{S})$, defined as follows.

$$sig(\mathcal{S}) =_{\mathbf{df}} \lambda\, \mathcal{S}.R{:}\star.\, \forall\, \mathcal{S}.D{:}\star.\, \Pi\, \mathcal{S}.\Xi.\, \mathcal{S}.D$$

Note that we understand abstraction and quantification over the type variable fields of $\mathcal{S}$ as capturing free occurrences of those variables in the body of the abstraction or quantification. As bound variables, we are still free to work up to α-equivalence as before.

Example 4.25. *We have the following derivation of the signature for datatype Nat (Example 4.22; we pick a fresh variable x for the parameter telescope of succ).*

$$\dfrac{\dfrac{\Gamma, Nat{:}\star \vdash () \searrow ()}{\Gamma \vdash_{\textbf{\textit{sig}}} (Nat, R, (zero{:}Nat)(succ{:}\Pi\, x{:}Nat.\, Nat), 1) \searrow (zero, ())} \;\; \mathcal{J}}{\begin{array}{l}\Gamma \vdash_{\textbf{\textit{sig}}} Decl[Nat, (zero{:}Nat)(succ{:}\Pi\, x{:}Nat.\, Nat)] \\ \quad \searrow \textsf{\textit{record}}\ Sig\ \{D = Nat; R = R; \Xi = (zero{:}Nat)(succ{:}\Pi\, x{:}R.\, Nat)\}\end{array}}$$

where $\mathcal{J}$ *is the derivation*

$$\dfrac{\Gamma, Nat{:}\star \vdash (x{:}Nat) \searrow (x{:}Nat)}{\Gamma \vdash_{\textbf{\textit{sig}}} (Nat, R, (zero{:}Nat)(succ{:}\Pi\, x{:}Nat.\, Nat), 2) \searrow (succ, (x{:}R))}$$

Putting everything together with sig and simplifying, the elaborated signature for Nat is the expected one.

$$\lambda\, R{:}\star.\, \forall\, Nat{:}\star.\, Nat \to (R \to Nat) \to Nat$$

Having seen the rules in action, we are now interested in the properties they possess. First and foremost among these properties is *soundness*, which is to say the telescope part of the elaborated signature information is well-formed. As we mentioned earlier, we are in the process of a *mutually* inductive definition, and we cannot expect to prove soundness of signature elaboration in isolation of the soundness of the other rules. Therefore, for the time being we will work under the following set of assumptions, which will be satisfied by the induction hypothesis of our final proof of soundness for the entire elaboration system.

Definition 4.26 (Assumptions for signature soundness)**.** For given Γ, Γ', and $Decl[D, \Delta]$, we refer to the conjuction of the following properties as the *assumptions for signature soundness*.

1. $\vdash \Gamma \searrow \Gamma'$
2. for all $(c_i{:}\Pi\, \Delta_i.\, D) \in \Delta$, if $\Gamma, D{:}\star \vdash \Delta_i \searrow \Delta_i'$ (for some Δ_i'), then $\Gamma', D{:}\star \vdash \Delta_i'$.

Note that the definition of the judgment for elaborating typing contexts, $\vdash \Gamma \searrow \Gamma'$, is postponed to Section 5.2.2.1.

Next, we define what we mean by soundness of the elaborated signature information.

Definition 4.27 (Soundness of computational signature for a datatype)**.** Define judgment $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{S})$ *valid*, which we read "under Γ, $\mathit{Decl}[D, \Delta]$ elaborates to valid *computational signature* $\mathcal{S}$ in Γ'," by the following inference rule.

$$\boxed{\Gamma^- \vdash \mathit{Decl}[D, \Delta]^- \searrow (\Gamma'^-, \mathcal{S}^-) \ \mathit{valid}}$$

$$\frac{\Gamma \vdash_{\mathbf{sig}} \mathit{Decl}[D, \Delta] \searrow \mathcal{S} \quad (\Gamma', \mathcal{S}.R : \star, \mathcal{S}.D : \star \vdash \mathcal{S}.\Xi(i))_{i \in \{1 \ldots \#\mathcal{S}.\Xi\}}}{\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{S}) \ \mathit{valid}}$$

From soundness of the computational signature for a datatype, it follows that the telescope part of the signature information is well-formed and thus the entire signature is well kinded at kind $\star \to \star$; see Appendix B for proofs of the properties in this section.

Proposition 4.28. *If* $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{S})$ *valid then*

1. $\Gamma', \mathcal{S}.R : \star, \mathcal{S}.D : \star \vdash \mathcal{S}.\Xi$
2. $\Gamma' \vdash \mathit{sig}(\mathcal{S}) : \star \to \star$.

It also follows (by the elaboration rules themselves) that the elaborated signature telescope denotes the same sequence as that of the original constructor telescope, and furthermore corresponding entries of these telescopes have parameter telescopes that denote the same sequence.

Fact 4.29. *If* $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{S})$ *valid then the following properties hold.*

- $\mathit{Seq}(\Delta) = \mathit{Seq}(\mathcal{S}.\Xi)$
- *if* $(c_i : \Pi\, \Delta_i.\, D) \in \Delta$ *then* $\mathcal{S}.\Xi(i)$ *is of the form* $(c_i : \Pi\, \Delta_i'[\mathcal{S}.R/D].\, D)$ *where* $\Gamma, D : \star \vdash \Delta_i \searrow \Delta_i'$

We now state soundness of the elaboration of computational signature information.

Theorem 4.30 ($\vdash_{\mathbf{sig}}$ soundness)**.** *Let* Γ, Γ', *and* $\mathit{Decl}[D, \Delta]$ *be such that the assumptions for signature soundness hold (Definition 4.26). If* $\Gamma \vdash_{\mathit{sig}} \mathit{Decl}[D, \Delta] \searrow \mathcal{S}$ *then* $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{S})$ *valid*.

Finally, let us record the observation that these judgments are algorithmic (see Section 3.1.2) and decidable under the assumption that the elaboration of the constructor parameter telescopes can be decided.

Proposition 4.31. *The judgments of Figure 4.8 for elaborating computational signature information are mode-correct and syntax-directed.*

Proof sketch. By inspection of the inference rules. We assume that we may always pick a variable that is fresh with respect to a finite set of variables. □

Definition 4.32 (Assumptions for signature elaboration decidability)**.** For give Γ and $\mathit{Decl}[D, \Delta]$, we refer to the following propert as the *assumption for signature decidability.* For all $(c_i : \Pi\, \Delta_i.\, D) \in \Delta$, it is decidable whether $\Gamma, D : \star \vdash \Delta_i \searrow \Delta_i'$ for some Δ_i'.

Proposition 4.33 ($\vdash_{\mathbf{sig}}$ decidability)**.** *For all Γ and $\mathit{Decl}[D, \Delta]$, such that the assumption for signature decidability holds, it is decidable whether $\Gamma \vdash_{\mathit{sig}} \mathit{Decl}[D, \Delta] \searrow \mathcal{S}$ for some $\mathcal{S}$.*

Considering the strength of the assumption used for Proposition 4.33, the reader may question the usefulness of establishing this property. Our purpose for this is to establish that signature elaboration does not introduce *new* sources of undecidability than were already present in CDLE's type inference rules.

4.2.3.2 Lambda Encoding of Signature Constructors

The main judgment elaborating signature constructors is $\Gamma \vdash_{\mathbf{con}} \mathit{Decl}[D, \Delta] \searrow \mathcal{C}$, which we read "under Γ, $\mathit{Decl}[D, \Delta]$ elaborates signature constructor information $\mathcal{C}$." The single inference rule defining this judgment, as well as the inference rules for the auxiliary judgments it uses, is shown in Figure 4.10. The record value $\mathcal{C}$ we elaborate contains all the signature information $\mathcal{S}$ elaborated by the earlier set of inference rules, plus a telescope $\mathcal{C}.\Delta$ and sequence $\mathcal{C}.\tau$ containing the types and definitions, respectively of the constructors for the signature.

$$\boxed{\Gamma^- \vdash_{\mathbf{cty}} (\mathcal{S}^-, i^-) \searrow (c^+, T^+)} \quad \boxed{\Gamma^- \vdash_{\mathbf{cty}} \mathcal{S}^- \searrow \Delta^+}$$

$$\frac{(c_i : \Pi\, \Delta'_i.\ D) \in \mathcal{S}.\Xi}{\Gamma \vdash_{\mathbf{cty}} (\mathcal{S}, i) \searrow (c_i, \forall\, \mathcal{S}.R : \star.\, \Pi\, \Delta'_i.\ sig(\mathcal{S}) \cdot \mathcal{S}.R)}$$

$$\frac{(\Gamma \vdash_{\mathbf{cty}} (\mathcal{S}, i) \searrow (c_i, T_i))_{i \in \{1 \ldots \#\mathcal{S}.\Xi\}} \qquad \underset{i \leq \#\mathcal{S}.\Xi}{\Delta(i)} =_{\mathbf{df}} (c_i : T_i)}{\Gamma \vdash_{\mathbf{cty}} \mathcal{S} \searrow \Delta}$$

$$\boxed{\Gamma^- \vdash_{\mathbf{ctm}} (\mathcal{S}^-, \Delta^-, i^-) \searrow t^+} \quad \boxed{\Gamma^- \vdash_{\mathbf{ctm}} (\mathcal{S}^-, \Delta^-) \searrow \tau}$$

$$\frac{(c_i : \Pi\, \Delta'_i.\ D) \in \mathcal{S}.\Xi \quad (c_i : T_i) \in \Delta}{\Gamma \vdash_{\mathbf{ctm}} (\mathcal{S}, \Delta, i) \searrow \chi\ T_i\ \text{-}\ \Lambda\, \mathcal{S}.R.\, \lambda\, \Delta'_i.\, \Lambda\, \mathcal{S}.D.\, \lambda\, \mathcal{S}.\Xi.\, c_i\ \Delta'_i}$$

$$\frac{(\Gamma \vdash (\mathcal{S}, \Delta, i) \searrow t_i)_{i \in \{1 \ldots \#\mathcal{S}.\Xi\}} \qquad \underset{i \leq \#\mathcal{S}.\Xi}{\tau(i)} =_{\mathbf{df}} (t_i)}{\Gamma \vdash_{\mathbf{ctm}} (\mathcal{S}, \Delta) \searrow \tau}$$

$$\boxed{\Gamma^- \vdash_{\mathbf{con}} \mathit{Decl}[D, \Delta]^- \searrow \mathcal{C}^+}$$

$$\frac{\Gamma \vdash_{\mathbf{sig}} \mathit{Decl}[D, \Delta] \searrow \mathcal{S} \quad \Gamma \vdash_{\mathbf{cty}} \mathcal{S} \searrow \Delta' \quad \Gamma \vdash_{\mathbf{ctm}} (\mathcal{S}, \Delta') \searrow \tau}{\Gamma \vdash_{\mathbf{con}} \mathit{Decl}[D, \Delta] \searrow \mathsf{record}\ \mathit{Con}\ \{\mathcal{S} = \mathcal{S}; \Delta = \Delta'; \tau = \tau\}}$$

Figure 4.10: Signature constructor elaboration (computational)

Definition 4.34 (Computational signature constructors)**.** In the meta-theory, define *Con* as the set of records with the fields $\mathcal{S}$, Δ, and τ, where $\mathcal{S} \in \mathit{Sig}$, Δ is a telescope and τ is a sequence. Meta-variable $\mathcal{C}$ ranges over members of *Sig*.

By abuse of notation, we treat the fields of $\mathcal{C}.\mathcal{S}$ as though they were fields of $\mathcal{C}$ itself, e.g., we write $\mathcal{C}.\Xi$ for $\mathcal{C}.\mathcal{S}.\Xi$. Field names are chosen such that no ambiguity may arise by this convention. Similarly, meta-linguistic functions such as *sig* that act over elements of *Sig* are lifted to act over elements of *Con* via an implicit projection: $\mathit{sig}(\mathcal{C}) =_{\mathbf{df}} \mathit{sig}(\mathcal{C}.\mathcal{S})$.

To elaborate the computational signature constructor information for datatype declaration $\mathit{Decl}[D, \Delta]$, we first elaborate the declaration's signature information $\mathcal{S}$. We use this information to elaborate a telescope Δ' for classifying the signature constructors with the judgment $\Gamma \vdash_{\mathbf{cty}} \mathcal{S} \searrow \Delta'$. This inference rule, in turn, produces each entry of the telescope in parallel using the judgment $\Gamma \vdash_{\mathbf{cty}} (\mathcal{S}, i) \searrow (c, T)$, which uses type quantification to capture the free occurrences of $\mathcal{S}.R$ in the constructor's parameter telescope and replaces the codomain type $\mathcal{S}.D$ with $\mathit{sig}(\mathcal{S}) \cdot \mathcal{S}.R$.

With the telescope for classifying the signature constructors at hand, we next use the judgment $\Gamma \vdash_{\mathbf{ctm}} (\mathcal{S}, \Delta) \searrow \tau$ to elaborate the sequence of signature constructors. The sequence is defined in parallel with respect to the $\#\mathcal{S}.\Xi$ derivations of the auxiliary judgment $\Gamma \vdash (\mathcal{S}, \Delta, i) \searrow t_i$ $(1 \leq i \leq \#\mathcal{S}.\Xi)$, where each t_i abstracts over the type variable $\mathcal{S}.R$, the constructor argument telescope Δ'_i, the type variable $\mathcal{S}.D$ and the signature telescope $\mathcal{S}.\Xi$, and for the body picks the ith entry of $\mathcal{S}.\Xi$ and applies this to the constructor parameters (compare with the coproduct injections of Section 4.2.2). The outermost term construct is a type ascription (χ, see Figure 3.7), as we require that the signature constructors synthesize their types.

Example 4.35. *We have the following derivation of the signature constructors for the declaration of datatype Nat. Below, we show only the derivations for* $\vdash_{\mathbf{cty}}$ *and* $\vdash_{\mathbf{ctm}}$*, and* $\mathcal{S}$ *is the elaborated signature information we saw in Example 4.25.*

For $\vdash_{\textbf{\textit{cty}}}$*, we have:*

$$\dfrac{(zero : Nat) = \mathcal{S}.\Xi(1) \qquad (succ : \Pi\, x : R.\, Nat) = \mathcal{S}.\Xi(2)}{\dfrac{\Gamma \vdash_{\textbf{\textit{cty}}} (\mathcal{S}, 1) \searrow (zero, \forall\, R : \star.\, sig(\mathcal{S}) \cdot R) \qquad \Gamma \vdash_{\textbf{\textit{cty}}} (\mathcal{S}, 2) \searrow (succ, \forall\, R : \star.\, \Pi\, x : R.\, sig(\mathcal{S}) \cdot R)}{\Gamma \vdash_{\textbf{\textit{cty}}} \mathcal{S} \searrow (zero : \forall\, R : \star.\, sig(\mathcal{S}) \cdot R)(succ : \forall\, R : \star.\, \Pi\, x : R.\, sig(\mathcal{S}) \cdot R)}}$$

For $\vdash_{\textbf{\textit{ctm}}}$*, let* $\Delta' =_{\textbf{df}} (zero : \forall\, R : \star.\, sig(\mathcal{S}) \cdot R)(succ : \forall\, R : \star.\, \Pi\, x : R.\, sig(\mathcal{S}) \cdot R)$*,* $T_1 =_{\textbf{df}} \forall\, R : \star.\, sig(\mathcal{S}) \cdot R$*, and* $T_2 =_{\textbf{df}} \forall\, R : \star.\, \Pi\, x : R.\, sig(\mathcal{S}) \cdot R$*, and we have*

$$\dfrac{(zero : Nat) = \mathcal{S}.\Xi(1) \qquad (succ : \Pi\, x : R.\, Nat) = \mathcal{S}.\Xi(2)}{\dfrac{\Gamma \vdash_{\textbf{\textit{ctm}}} (\mathcal{S}, \Delta', 1) \searrow \chi\ T_1\ \text{-}\ \Lambda\, R.\, \Lambda\, Nat.\, \lambda\, zero.\, \lambda\, succ.\, zero \qquad \Gamma \vdash_{\textbf{\textit{ctm}}} (\mathcal{S}, \Delta', 2) \searrow \chi\ T_2\ \text{-}\ \Lambda\, R.\, \lambda\, x.\, \Lambda\, Nat.\, \lambda\, zero.\, \lambda\, succ.\, succ\ n}{\Gamma \vdash_{\textbf{\textit{ctm}}} (\mathcal{S}, \Delta') \searrow (\chi\ T_1\ \text{-}\ \Lambda\, R.\, \Lambda\, Nat.\, \lambda\, zero.\, \lambda\, succ.\, zero)(\chi\ T_2\ \text{-}\ \Lambda\, R.\, \lambda\, n.\, \Lambda\, Nat.\, \lambda\, zero.\, \lambda\, succ.\, succ\ n)}$$

As we did with the datatype's signature information, we start with a definition of what it means for the elaborated signature constructor information to be sound. In this case, not only do we want well-formedness of the telescope for typing these constructors, but also that this telescope classifies the sequence of signature constructors.

Definition 4.36 (Soundness of computational signature constructors for a datatype)**.**

$$\dfrac{\Gamma \vdash_{\textbf{con}} Decl[D, \Delta] \searrow \mathcal{C} \qquad \Gamma \vdash Decl[D, \Delta] \searrow (\Gamma', \mathcal{C}.\mathcal{S})\ valid \qquad (\Gamma' \vdash \mathcal{C}.\Delta(i))_{i \in \{1 \ldots \#\mathcal{C}.\Delta\}} \qquad \#\mathcal{C}.\Delta = \#\mathcal{C}.\tau \qquad (\Gamma' \vdash \mathcal{C}.\tau(i) : \mathcal{C}.\Delta(i))_{i \in \{1 \ldots \#\mathcal{C}.\Delta\}}}{\Gamma \vdash Decl[D, \Delta] \searrow (\Gamma', \mathcal{C})\ valid}$$

It is also worth observing how valid signature constructor information relates to the original datatype, and the shape of the erasures of the signature constructors.

Fact 4.37. *If* $\Gamma \vdash Decl[D, \Delta] \searrow (\Gamma', \mathcal{C})$ *valid then the following properties hold.*

- $Seq(\Delta) = Seq(\mathcal{C}.\Delta)$
- *if* $(c_i : \Delta_i D) \in \Delta$ *then*
  - $\mathcal{C}.\Delta(i)$ *is of the form* $(c_i : \forall\, \mathcal{C}.R : \star.\, \Pi\, \Delta_i'[\mathcal{C}.R/D].\, sig(\mathcal{C}) \cdot \mathcal{C}.R)$ *where* $\Gamma \vdash \Delta_i \searrow \Delta_i'$
  - $|\mathcal{C}.\tau(i)\ \Delta_i| =_{\beta\eta} \lambda\, \Delta.\, |c_i\ \Delta_i|$

We now state soundness of elaboration of the computational signature constructors for datatype declarations.

Theorem 4.38 ($\vdash_{\mathbf{con}}$ soundness)**.** *Let* Γ, Γ', *and* $Decl[D, \Delta]$ *be such that the assumptions for signature soundness hold. If* $\Gamma \vdash_{\boldsymbol{con}} Decl[D, \Delta] \searrow \mathcal{C}$ *then* $\Gamma \vdash Decl[D, \Delta] \searrow (\Gamma', \mathcal{C})$ *valid.*

Lastly, we confirm syntax-directedness, mode-correctness, and decidability up to assumption for the rules.

Proposition 4.39 ($\vdash_{\mathbf{con}}$ syntax-directedness and mode-correctness)**.** *The inference rules listed in Figure 4.10 are syntax-directed and mode-correct.*

Proof sketch. By inspection of the rules. □

Proposition 4.40 ($\vdash_{\mathbf{con}}$ decidability)**.** *For all* Γ *and* $Decl[D, \Delta]$ *such that the assumptions for signature decidability hold, it is decidable whether* $\Gamma \vdash_{\boldsymbol{con}} Decl[D, \Delta] \searrow \mathcal{C}$ *for some* $\mathcal{C}$*.*

Proof sketch. We invoke Proposition 4.33 for the first premise, $\Gamma \vdash_{\mathbf{sig}} Decl[D, \Delta] \searrow \mathcal{S}$. Since the judgment can only be affirmed when this premise is, if the decision we obtain is a refutation then we refute the entire judgment. Othwerise, the premises are constructable and we affirm the judgment. □

4.2.4 Impredicative Encoding of Coproducts (Part 2)

As we did in Section 4.2.2, before digging through the elaboration rules that equip the datatype signature with a dependent eliminator (i.e., allow us to write proofs by cases), we pause to show the simpler case for doing the same with coproducts.

Using Agda-style pseudocode, we can define the induction principle for the datatype *Sum* as follows.

```
indSum : ∀ T1: ⋆. ∀ T2: ⋆. ∀ X: Sum ·T1 ·T2 → ⋆.
         (Π x: T1. X (inl x)) → (Π y: T2. X (inr y))
         → Π s: Sum ·T1 ·T2. X s
indSum f1 f2 (inl x) = f1 x
indSum f1 f2 (inr y) = f2 y
```

As before, we seek to define (for fixed T_1 and T_2) the type $Sum \cdot T_1 \cdot T_2$ *in terms of* the type of the above eliminator. However, in this case the solution is far

from obvious, as the constraint we generate from this type signature is both *recursive* and *dependent*.

$$(s : Sum \cdot T_1 \cdot T_2) \leq \forall\, X : Sum \cdot T_1 \cdot T_2 \to \star.\, (\Pi\, x : T_1.\, X\ (inl\ x)) \to (\Pi\, y : T_2.\, X\ (inr\ y)) \to X\ s$$

We start by expressing the right-hand side of this constraint as a predicate over the impredicative encoding we developed in Section 4.2.3. To avoid confusion, we will rename some of the definitions given there.

$$\begin{array}{rcl} SumC & =_{\mathbf{df}} & \lambda\, Y : \star.\, \lambda\, Z : \star.\, \forall\, X : \star.\, (Y \to X) \to (Z \to X) \to X \\ inC_l & =_{\mathbf{df}} & \Lambda\, Y.\, \Lambda\, Z.\, \lambda\, y.\, \Lambda\, X.\, \lambda\, x_1.\, \lambda\, x_2.\, x_1\ y \\ inC_r & =_{\mathbf{df}} & \Lambda\, Y.\, \Lambda\, Z.\, \lambda\, z.\, \Lambda\, X.\, \lambda\, x_1.\, \lambda\, x_2.\, x_2\ z \end{array}$$

Now we can define our predicate, $SumI$.

$$\begin{array}{rcl} SumI & =_{\mathbf{df}} & \lambda\, Y : \star.\, \lambda\, Z : \star.\, \lambda\, x : SumC \cdot Y \cdot Z.\, \forall\, X : SumC \cdot Y \cdot Z \to \star. \\ & & \quad (\Pi\, y : Y.\, X\ (inC_l\ y)) \to (\Pi\, z : Z.\, X\ (inC_r\ z)) \to X\ x \end{array}$$

Following the recipe of Stump [80] (which is based on an observation by Leivant [55]), we observe that we can prove that the constructors inC_l and inC_r satisfy this predicate, and furthermore that those proofs are themselves *equal* (modulo erasure) to the constructors to which they refer. We show these below, with type ascriptions added for readability.

$$\begin{array}{rcl} inI_l & =_{\mathbf{df}} & \chi\ (\forall\, Y : \star.\, \forall\, Z : \star.\, \Pi\, y : Y.\, SumI\ (inC_l\ y)) \\ & & \quad \text{-}\ \Lambda\, Y.\, \Lambda\, Z.\, \lambda\, y.\, \Lambda\, X.\, \lambda\, x_1.\, \lambda\, x_2.\, x_1\ y \\ inI_r & =_{\mathbf{df}} & \chi\ (\forall\, Y : \star.\, \forall\, Z : \star.\, \Pi\, z : Z.\, SumI\ (inC_r\ z)) \\ & & \quad \text{-}\ \Lambda\, Y.\, \Lambda\, Z.\, \lambda\, z.\, \Lambda\, X.\, \lambda\, x_1.\, \lambda\, x_2.\, x_2\ z \end{array}$$

We therefore use dependent intersection (see Figure 3.7) to form our candidate type constructor Sum whose inhabitants *inherently* satisfy the $SumI$ predicate, and define its constructors.

$$\begin{array}{rcl} Sum & =_{\mathbf{df}} & \lambda\, Y : \star.\, \lambda\, Z : \star.\, \iota\, x : SumC \cdot Y \cdot Z.\, SumI \cdot Y \cdot Z\ x \\ in_l & =_{\mathbf{df}} & \Lambda\, Y.\, \Lambda\, Z.\, \lambda\, y.\, [inC_l\ y, inI_l\ y] \\ in_r & =_{\mathbf{df}} & \Lambda\, Y.\, \Lambda\, Z.\, \lambda\, z.\, [inC_r\ z, inI_r\ z] \end{array}$$

Finally, we turn to the dependent eliminator *indSum*. Observe that there is a mismatch between the desired type for *indSum*, which should allow us to prove properties over inhabitants of type $Sum \cdot T_1 \cdot T_2$ (for arbitrary T_1,T_2), and the proof principle baked into the definition of *Sum* itself, which allows us to prove properties about inhabitants of the weaker type $SumC \cdot T_1 \cdot T_2$. There are a few ways to bridge this gap, but arguably the simplest and most elegant method is to observe that every predicate $T : Sum \cdot T_1 \cdot T_2 \to \star$ can be "lifted" to a related predicate over inhabitants t of $SumC \cdot T_1 \cdot T_2$ which states that T holds of t if t can be *viewed as having the required type* $Sum \cdot T_1 \cdot T_2$.

$$\begin{array}{rcl} Lift & =_{\mathbf{df}} & \lambda\, Y\!:\!\star.\, \lambda\, Z\!:\!\star.\, \lambda\, X\!:\!Sum \cdot Y \cdot Z.\, \lambda\, x\!:\!SumC \cdot T_1 \cdot T_2. \\ & & \quad \forall\, v\!:\!View \cdot (Sum \cdot T_1 \cdot T_2)\ \beta\{x\}.\, X\ (elimView\ \beta\{x\}\ \text{-}v) \end{array}$$

(*View* is presented axiomatically in Figure 3.9).

With this final piece, we can define the induction principle *indSum* for *Sum*.

$$\begin{array}{rcl} indSum & =_{\mathbf{df}} & \Lambda\, Y.\, \Lambda\, Z.\, \Lambda\, X.\, \lambda\, f_1.\, \lambda\, f_2.\, \lambda\, x. \\ & & \quad x.2 \cdot (Lift \cdot Y \cdot Z \cdot X)\ (\lambda\, y.\, \Lambda\, v.\, f_1\ y)\ (\lambda\, z.\, \Lambda\, v.\, f_2\ z) \\ & & \quad\ \text{-}(intrView \cdot (Sum \cdot Y \cdot Z)\ \beta\{x\}\ \text{-}x\ \text{-}\beta) \end{array}$$

Observe that $|indSum| =_{\beta\eta} |elimSum|$ (see Section 4.2.2). Let us walk through this final definition carefully, as there are some subtleties.

- In the body of *indSum*, we use the second projection of x, $x.2 : SumI \cdot Y \cdot Z\ x.1$, and instantiate its type constructor argument with $Lift \cdot Y \cdot Z \cdot X$. This creates three obligations for proving $X\ x$, given as the three remaining arguments to $x.2$.
- The first two of these obligations are similar, so we discuss only the first. The goal is to inhabit the type $\Pi\, y\!:\!Y.\, Lift \cdot Y \cdot Z \cdot X\ (inC_l\ y)$. Unrolling the definition

of the type and abstracting over $y : Y$ and $v : \mathit{View} \cdot (\mathit{Sum} \cdot X \cdot Y)\ \beta\{\mathit{inC}_l\ y\}$, the goal becomes $X\ (\mathit{elimView}\ \beta\{\mathit{inC}_l\ y\}\ \text{-}v)$. The expression we give is $f_1\ y$, whose type $X\ (\mathit{in}_l\ y)$ is convertible with the desired one.

- The last obligation is a proof that x may be viewed as having type $\mathit{Sum} \cdot Y \cdot Z$. This is trivial, as x is already assumed to have this type!
- Stepping back a bit, the entire application spine headed by $x.2$ has type

$$X\ (\mathit{elimView}\ \beta\{x\}\ \text{-}(\mathit{intrView} \cdot (\mathit{Sum} \cdot Y \cdot Z)\ \beta\{x\}\ \text{-}x\ \text{-}\beta))$$

  This is, again, convertible with the desired type $X\ x$.

This is the essence of the more elaborate encodings we shall see next.

4.2.5 Signature Elaboration (Inductive)

Figures 4.12 and 4.13 give the final set of elaboration rules for datatype signatures and their signature constructors. As before, we discuss each of these in turn, walking through the inference rules and proving soundness up to the assumptions of Definition 4.26.

We again orient the reader to the main judgments of this process with a high-level illustration of the flow of elaboration, shown in Figure 4.11.

1. Picking up where we left of with the elaboration of constructor information for the "computational" signature, we elaborate the "inductive" signature information with $\Gamma \vdash_{\mathbf{isig}} \mathit{Decl}[D, \Delta] \searrow \mathcal{I}$. This corresponds to SumI from Section 4.2.4.
2. We next elaborate the types, collected together in telescope Θ, of the inductive signature constructors with judgment $\Gamma \vdash_{\mathbf{icty}} \mathcal{I} \searrow \Theta$. This corresponds to the

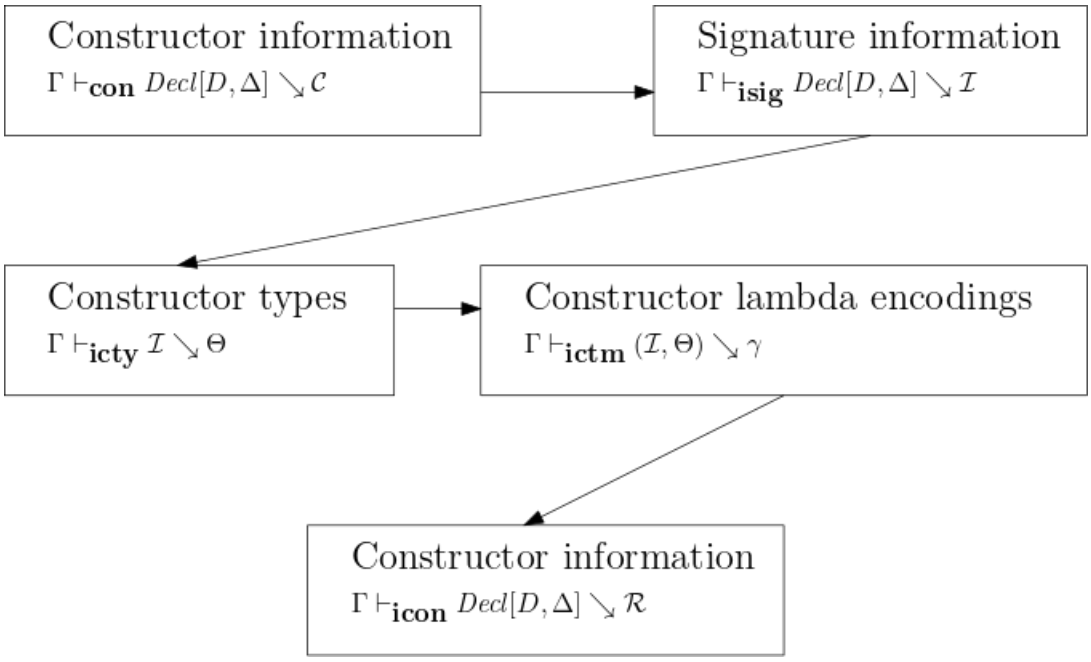


Figure 4.11: Flow of inductive signature elaboration

types of inI_l and inI_r.

3. We obtain the elaborated lambda encodings, collected together in sequence γ, of the inductive signature constructors with judgment $\Gamma \vdash_{\mathbf{ictm}} (\mathcal{I}, \Theta) \searrow \gamma$. This corresponds to the definitions of inI_l and inI_r.

4. Finally, the last judgment, $\Gamma \vdash_{\mathbf{icon}} Decl[D, \Delta] \searrow \mathcal{R}$, collects all we have obtained so far into $\mathcal{R}$, the inductive signature constructor information. This process is cumulative, so $\mathcal{C}$ contains also all the information present in $\mathcal{C}$ and $\mathcal{I}$.

4.2.5.1 Impredicative Encoding of Signatures

The only judgment for signature elaboration is $\Gamma \vdash_{\mathbf{isig}} Decl[D, \Delta] \searrow \mathcal{I}$, which we read "under Γ, $Decl[D, \Delta]$ elaborates inductive signature information $\mathcal{I}$." Before discussing its single inference rule, which is somewhat intricate, we define the record

$$\boxed{\Gamma^- \vdash_{\mathbf{isig}} \mathit{Decl}[D, \Delta]^- \searrow \mathcal{I}^+}$$

$$\frac{\Gamma \vdash_{\mathbf{con}} \mathit{Decl}[D, \Delta] \searrow \mathcal{C} \quad \{z, v\} \cap (DV(\Gamma) \cup DV(\Delta)) = \emptyset \wedge z \neq v \quad \underset{i \leq \#\mathcal{C}.\Xi}{\Sigma(i)} =_{\mathbf{df}} (c_i : \Pi\, \Delta_i'.\, D\ (\mathcal{C}.\tau(i) \cdot \mathcal{C}.R\ \Delta_i')) \text{ where } (c_i : \Pi\, \Delta_i'.\, D \in \mathcal{C}.\Xi)}{\Gamma \vdash_{\mathbf{isig}} \mathit{Decl}[D, \Delta] \searrow \mathsf{record}\ \mathit{ISig}\ \{\mathcal{C} = \mathcal{C}; z = z; v = v; \Sigma = \Sigma\}}$$

Figure 4.12: Signature elaboration (inductive)

object that is its output and the two operations it must support.

Definition 4.41. In the meta-theory, define *ISig* as the set of records with fields $\mathcal{C}$ (the computational signature constructor information), term variables z and v, and telescope Σ. Meta-variable $\mathcal{I}$ ranges over members of *ISig*. As before, we treat the fields of $\mathcal{C}$ (and now $\mathcal{C}.\mathcal{S}$ as well) as if they were fields of $\mathcal{I}$ as well, e.g., we write $\mathcal{I}.\tau$ and $\mathcal{I}.\Xi$ for $\mathcal{I}.\mathcal{C}.\tau$ and $\mathcal{I}.\mathcal{C}.\mathcal{S}.\Xi$, respectively, and automatically lift the operation *sig* to act on members of *ISig* via implicit projections.

For a given $\mathcal{I}$, we define the following two operations: $\mathit{isig}(\mathcal{I})$, which gives the elaborated datatype signature, and *lift*, which gives the predicate transformer used for dependent elimination of the signature.

$$\begin{array}{rcl} \mathit{isig}(\mathcal{I}) & =_{\mathbf{df}} & \lambda\, \mathcal{I}.R : \star.\, \iota\, \mathcal{I}.z : \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R.\, \forall\, \mathcal{I}.D : \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star.\, \Pi\, \mathcal{I}.\Sigma.\, \mathcal{I}.D\ \mathcal{I}.z \\ \mathit{lift}(\mathcal{I}) & =_{\mathbf{df}} & \lambda\, \mathcal{I}.R : \star.\, \lambda\, \mathcal{I}.D : \mathit{isig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star.\, \lambda\, \mathcal{I}.z : \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R. \\ & & \quad \forall\, \mathcal{I}.v : \mathit{View} \cdot (\mathit{isig}(\mathcal{I}) \cdot \mathcal{I}.R)\ \beta\{\mathcal{I}.z\}.\, \mathcal{I}.D\ (\mathit{elimView}\ \beta\{\mathcal{I}.z\}\ \text{-}\mathcal{I}.v) \end{array}$$

With Definition 4.41, we can now see that the premise of the rule in Figure 4.12 concerning variables z and v exists to prevent capture of bound variables when generating the datatype signature with *isig* or predicate transformer *lift*. For the remainder of the rule, after elaborating the computational signature constructor information $\mathcal{C}$ for the datatype declaration, we form the telescope Σ with the types of each entry being proofs that an arbitrary predicate $D : \mathit{sig}(\mathcal{C}) \cdot \mathcal{C}.R \to \star$ holds for each computational signature constructor $\mathcal{C}.\tau(i)$ (recall the definition of *SumI* in Section 4.2.4).

Example 4.42. *Below, let* $\mathcal{C}$ *be such that* $\Gamma \vdash \mathit{Decl}[\mathit{Nat}, (\mathit{zero} : \mathit{Nat})(\mathit{succ} : \Pi\, x : \mathit{Nat}.\, \mathit{Nat})] \searrow \mathcal{C}$*, and let* $\mathit{zero}' =_{\mathbf{df}} \mathcal{C}.\tau(i)$ *and* $\mathit{succ}' =_{\mathbf{df}} \mathcal{C}.\tau(2)$ *(see Example 4.35 for the full definitions). The telescope* Σ *elaborated by the inference rule of Figure 4.12 is*

$$(\mathit{zero} : \mathit{Nat}\ (\mathit{zero}' \cdot \mathcal{C}.R))(\mathit{succ} : \Pi\, x : \mathcal{C}.R.\, \mathit{Nat}\ (\mathit{succ}' \cdot \mathcal{C}.R\ x))$$

and the completed signature elaborated for datatype Nat is

$$\begin{array}{l}\lambda\, \mathcal{C}.R : \star.\, \iota\, z : \mathit{sig}(\mathcal{C}) \cdot \mathcal{C}.R.\, \forall\, \mathit{Nat} : \mathit{sig}(\mathcal{C}) \cdot \mathcal{C}.R \to \star. \\ \quad \Pi\, \mathit{zero} : \mathit{Nat}\ (\mathit{zero}' \cdot \mathcal{C}.R).\, \Pi\, \mathit{succ} : (\Pi\, x : \mathcal{C}.R.\, \mathit{Nat}\ (\mathit{succ}' \cdot \mathcal{C}.R\ x)).\, \mathit{Nat}\ z\end{array}$$

We again use a judgment to define what we mean by soundness of signature elaboration.

Definition 4.43 (Soundness of inductive signature for a datatype)**.** Define judgment $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{I})$, which we read "under Γ, $\mathit{Decl}[D, \Delta]$ elaborates to valid *inductive signature information* $\mathcal{I}$ in Γ'," by the following inference rule.

$$\boxed{\Gamma^- \vdash \mathit{Decl}[D, \Delta]^- \searrow (\Gamma'^-, \mathcal{I}^-)\ \mathit{valid}}$$

$$\frac{\Gamma \vdash_{\mathbf{isig}} \mathit{Decl}[D, \Delta] \searrow \mathcal{I} \quad \Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{I}.\mathcal{C})\ \mathit{valid} \quad (\Gamma', \mathcal{I}.R : \star, \mathcal{I}.D : \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star \vdash \mathcal{I}.\Sigma(i))_{i \in \{1 \ldots \#\mathcal{I}.\Sigma\}}}{\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{I})\ \mathit{valid}}$$

From soundness of the datatype signature information, it follows that the telescope part is well-formed and thus the entire signature is well-kinded at kind $\star \to \star$, and furthermore that the predicate transformer obtained by *lift* is well-kinded.

Proposition 4.44. *If* $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{I})$ *valid then*

1. $\Gamma', \mathcal{I}.R : \star, \mathcal{I}.D : \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star \vdash \mathcal{I}.\Sigma$
2. $\Gamma' \vdash \mathit{isig}(\mathcal{I}) : \star \to \star$
3. $\Gamma' \vdash \mathit{lift}(\mathcal{I}) : \Pi\, \mathcal{I}.R : \star.\, (\mathit{isig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star) \to \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star$

We also record the following facts regarding inductive signature information judged to be sound (these will be useful to refer to when we look next to the signature constructors).

Fact 4.45. *If* $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow \mathcal{I}$ *then*

- $\mathit{Seq}(\Delta) = \mathit{Seq}(\mathcal{I}.\Xi) = \mathit{Seq}(\mathcal{I}.\Sigma)$

- *if* $(c_i : \Delta_i D) \in \Delta$*, then* $\mathcal{I}.\Sigma(i)$ *is of the form* $(c_i : \Pi\, \Delta_i'[\mathcal{I}.R/D].\, D\ (\mathcal{I}.\tau(i) \cdot \mathcal{I}.R\ \Delta_i'))$*, where* $\Gamma \vdash \Delta_i \searrow \Delta_i'$

Finally, we state soundness of elaboration of the signature information, and for the inference rules state syntax-directedness, mode-correctness, and decidability up to assumption.

Theorem 4.46 ($\vdash_{\mathbf{isig}}$ soundness)**.** *Let* Γ*,* Γ'*, and* $Decl[D, \Delta]$ *be such that the assumptions for signature soundness hold (Definition 4.26). If* $\Gamma \vdash_{\boldsymbol{isig}} \searrow \mathcal{I}$ *then* $\Gamma \vdash Decl[D, \Delta] \searrow (\Gamma', \mathcal{I})$ *valid.*

Proposition 4.47 ($\vdash_{\mathbf{isig}}$ syntax-directedness and mode-correctness)**.** *The inference rules listed in Figure 4.12 are syntax-directed and mode-correct.*

Proof sketch. By inspection of the rules. □

Proposition 4.48 ($\vdash_{\mathbf{isig}}$ decidability)**.** *For all* Γ *and* $Decl[D, \Delta]$ *such that the assumptions for signature decidability hold (Definition 4.32), it is decidable whether* $\Gamma \vdash_{\boldsymbol{isig}} Decl[D, \Delta] \searrow \mathcal{I}$ *for some* $\mathcal{I}$*.*

Proof sketch. We invoke Proposition 4.40 to obtain a decision for the first premise of the rule, $\Gamma \vdash_{\mathbf{con}} Decl[D, \Delta] \searrow \mathcal{C}$. The judgment can be affirmed if and only if the decision so obtained is an affirmation (all other premises are constructable). □

4.2.5.2 Lambda Encoding of Signature Constructors

The main judgment for the elaboration of signature constructors capable of supporting dependent elimination is $\Gamma \vdash_{\mathbf{icon}} Decl[D, \Delta] \searrow \mathcal{R}$, which we read "under Γ, $Decl[D, \Delta]$ elaborates inductive signature constructor information $\mathcal{R}$." The meta-linguistic record objects elaborated by this judgment are defined below.

Definition 4.49 (Inductive signature constructor information)**.** In the meta-theory, define *ICon* as the set of records whose fields are inductive signature information $\mathcal{I}$, signature constructor telescope Θ, and signature constructor sequence τ. We let meta-variable $\mathcal{R}$ range over members of *ICon*, and continue our usual convention of hereditarily treating the fields of member records of $\mathcal{R}$ as fields of $\mathcal{R}$ directly and hereditarily lifting operations on member fields to $\mathcal{R}$.

Returning to our description of the main judgment, the first step is elaborating the inductive signature information $\mathcal{I}$, using this to elaborate the telescope Θ for

$$\boxed{\Gamma^- \vdash_{\mathbf{icty}} (\mathcal{I}^-, i^-) \searrow (c^+, T^+)} \quad \boxed{\Gamma^- \vdash_{\mathbf{icty}} \mathcal{I} \searrow \Theta}$$

$$\frac{(c_i : \Pi\, \Delta_i'.\ D) \in \mathcal{I}.\Xi}{\Gamma \vdash_{\mathbf{icty}} (\mathcal{I}, i) \searrow (c_i, \forall \mathcal{I}.R : \star.\, \Pi\, \Delta_i'.\ isig(\mathcal{I}) \cdot \mathcal{I}.R)}$$

$$\frac{(\Gamma \vdash_{\mathbf{icty}} (\mathcal{I}, i) \searrow (c_i, T_i))_{i \in \{1 \ldots \#\mathcal{I}.\Xi\}} \quad \underset{i \leq \#\mathcal{I}.\Sigma}{\Theta(i)} =_{\mathbf{df}} (c_i : T_i)}{\Gamma \vdash_{\mathbf{icty}} \mathcal{I} \searrow \Theta}$$

$$\boxed{\Gamma^- \vdash_{\mathbf{ictm}} (\mathcal{I}^-, \Theta^-, i^-) \searrow t^+} \quad \boxed{\Gamma^- \vdash_{\mathbf{ictm}} (\mathcal{I}^-, \Theta^-) \searrow \gamma^+}$$

$$\frac{(c_i : T_i) \in \Theta \quad (c_i : \Pi\, \Delta_i'.\ D) \in \mathcal{I}.\Xi}{\Gamma \vdash_{\mathbf{ictm}} (\mathcal{I}, \Theta, i) \searrow \chi\ T_i \text{ - } \Lambda\, \mathcal{I}.R.\, \lambda\, \Delta_i'.\, [\mathcal{I}.\tau(i) \cdot \mathcal{I}.R\ \Delta_i', \Lambda\, \mathcal{I}.D.\, \lambda\, \mathcal{I}.\Sigma.\, c_i\ \Delta_i']}$$

$$\frac{(\Gamma \vdash_{\mathbf{ictm}} (\mathcal{I}, \Theta, i) \searrow t_i)_{i \in \{1 \ldots \#\Theta\}} \quad \underset{i \leq \#\mathcal{I}.\Theta}{\gamma(i)} =_{\mathbf{df}} t_i}{\Gamma \vdash_{\mathbf{ictm}} (\mathcal{I}, \Theta) \searrow \gamma}$$

$$\boxed{\Gamma^- \vdash_{\mathbf{icon}} Decl[D, \Delta]^- \searrow \mathcal{R}}$$

$$\frac{\Gamma \vdash_{\mathbf{isig}} Decl[D, \Delta] \searrow \mathcal{I} \quad \Gamma \vdash_{\mathbf{icty}} \mathcal{I} \searrow \Theta \quad \Gamma \vdash_{\mathbf{ictm}} (\mathcal{I}, \Theta) \searrow \gamma}{\Gamma \vdash_{\mathbf{icon}} Decl[D, \Delta] \searrow \mathsf{record}\ ICon\ \{\mathcal{I} = \mathcal{I}, \Theta = \Theta, \gamma = \gamma\}}$$

Figure 4.13: Signature constructor elaboration (inductive)

classifying the signature constructors with judgment $\Gamma \vdash_{\mathbf{icty}} \mathcal{I} \searrow \Theta$, then using both $\mathcal{I}$ and Θ to elaborate the sequence γ of inductive signature constructors with judgment $\Gamma \vdash_{\mathbf{ictm}} (\mathcal{I}, \Theta) \searrow \gamma$. For the telescope, each classifier quantifies over type variable $\mathcal{I}.R$ and the constructor parameter telescope and has as its codomain the type $isig(\mathcal{I}) \cdot \mathcal{I}.R$. Elaboration of the inductive signature constructors is a bit more involved. For the constructor corresponding to the ith constructor of a datatype, we:

- begin with a type ascription T_i, where T_i is the classifier corresponding to this constructor (as we need the constructors to synthesize their types);
- abstract over the type variable $\mathcal{I}.R$ and the constructor parameter telescope Δ_i'; and
- introduce a dependent intersection where
  - the first component is the ith computational signature constructor $\mathcal{I}.\tau(i)$ applied to the abstracted parameters, and
  - the second component is a proof that the first component satisfies the inductivity predicate, formed by abstracting over type constructor variable $\mathcal{I}.D : sig(\mathcal{I}) \cdot \mathcal{I}.R \to \star$ and the inductive signature telescope $\mathcal{I}.\Sigma$ and giving as a body the variable c_i (the identifier of the ith entry of $\mathcal{I}.\Xi$, which for sound signature information the same as the identifier of the ith entry $\mathcal{I}.\Sigma$ by Fact 4.45) applied to the constructor parameter telescope.

For soundness of the elaborated information for inductive signature constructors $\mathcal{R}$, we require that the inductive signature information $\mathcal{R}.\mathcal{I}$ is sound, that each of the entries of the signature constructor telescope is well-formed and is the type of

the corresponding entry for the signature constructor sequence.

Definition 4.50 (Soundness of inductive signature constructors for a datatype)**.**

$$\frac{\Gamma \vdash_{\mathbf{icon}} \mathit{Decl}[D, \Delta] \searrow \mathcal{R} \quad \Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{R}.\mathcal{I})\ \mathit{valid} \quad (\Gamma' \vdash \mathcal{R}.\Theta(i))_{i \in \{1 \ldots \#\mathcal{R}.\Theta\}} \quad \#\mathcal{R}.\Theta = \#\mathcal{R}.\gamma \quad (\Gamma' \vdash \mathcal{R}.\gamma(i) : \mathcal{R}.\Theta(i))_{i \in \{1 \ldots \#\mathcal{R}.\Theta\}}}{\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{R})\ \mathit{valid}}$$

Fact 4.51. *If* $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{R})$ *valid then*

- *if* $(c_i : \Pi\, \Delta_i'.\ D) \in \mathcal{R}.\Xi$*, then* $\Gamma' \vdash \mathcal{R}.\gamma(i) \in \forall\, \mathcal{R}.R : \star.\, \Pi\, \Delta_i'.\ \mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R$
- *if* $(c_i : \Pi\, \Delta_i'.\ D) \in \mathcal{R}.\Xi$*, then* $|\mathcal{R}.\gamma(i) \cdot \mathcal{R}.R\ \Delta_i'| =_{\beta\eta} |\mathcal{R}.\tau(i) \cdot \mathcal{R}.R\ \Delta_i'|$

We can then prove elaboration of the inductive signature constructor information is sound under the assumptions of Definition 4.26. We also record that the inferences rules that are syntax-directed, mode-correct, and that the judgment is decidable up to assumptions.

Theorem 4.52 ($\vdash_{\mathbf{icon}}$ soundness)**.** *Let* Γ*,* Γ'*, and* $\mathit{Decl}[D, \Delta]$ *be such that the assumptions for signature soundness hold. If* $\Gamma \vdash_{\mathit{isig}} \mathit{Decl}[D, \Delta] \searrow \mathcal{R}$ *then* $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{R})$ *valid*

Proposition 4.53 ($\vdash_{\mathbf{icon}}$ syntax-directedness and mode-correctness)**.** *The inference rules listed in Figure 4.13 are syntax-directed and mode-correct.*

Proof. By inspection of the rules. □

Proposition 4.54 ($\vdash_{\mathbf{icon}}$ decidability)**.** *For all* Γ *and* $\mathit{Decl}[D, \Delta]$ *such that the assumption for signature decidability hold, it is decidable whether* $\Gamma \vdash_{\mathit{isig}} \mathit{Decl}[D, \Delta] \searrow \mathcal{R}$ *for some* $\mathcal{R}$*.*

Proof sketch. We invoke Proposition 4.48 to obtain a decision for the first premise of the rule, $\Gamma \vdash_{\mathbf{isig}} \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$. The judgment is affirmed iff this decision is an affirmation, as (when it is an affirmation) the remaining premises are constructable. □

4.2.6 Signature Dependent Eliminator

The final piece for elaborating datatype signatures is producing their dependent eliminators. For a given signature, the dependent eliminator justifies pattern matching and proof by cases, which is essential to σ- and μ-expressions in the surface

$$\frac{\Gamma' \vdash S : \star \quad \Gamma' \vdash T : \mathit{isig}(\mathcal{R}) \cdot S \to \star \quad \#\tau = \#\mathcal{R}.\Theta \quad (\Gamma' \vdash \Pi\, \Delta_i'[S/\mathcal{R}.R].\, T\ (\mathcal{R}.\gamma(i) \cdot S\ \Delta_i') \ni \tau(i) \text{ where } (c_i : \Pi\, \Delta_i'.\, \mathcal{R}.D) \in \mathcal{R}.\Xi)_{i \in \{1 \ldots \#\mathcal{R}.\Theta\}}}{\Gamma' \vdash \mathit{case}(\mathcal{R}) \cdot S \cdot T\ \tau \in \Pi\, x : \mathit{isig}(\mathcal{R}) \cdot S.\, T\ x}$$

$$\left(\begin{array}{c} |\mathit{case}(\mathcal{R}) \cdot S \cdot T\ \tau\ (\mathcal{R}.\gamma(i) \cdot S\ \tau')| =_{\beta\eta} |\tau(i)\ \tau'| \\ \text{where } (c_i : \Pi\, \Delta_i'.\, D) \in \mathcal{R}.\Xi \text{ and } \Gamma' \vdash \tau' : \Delta_i' \end{array} \right)_{i \in \{1 \ldots \#\mathcal{R}.\Theta\}}$$

Figure 4.14: Dependent eliminator for datatype signatures: typing and computation laws

language. We start by defining the dependent elimination scheme in Figure 4.14 in its full generality, then we will show how the scheme is implemented. Note that the typing and computation laws of the figure are only sensible under the assumption that $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{R})$ *valid* for some $\mathit{Decl}[D, \Delta]$.

For the typing law, given (inductive) signature information $\mathcal{R}$, the combinator for dependent elimination requires:

- a type S for the signature parameter;
- a predicate $T : \mathit{isig}(\mathcal{R}) \cdot S \to \star$; and
- a sequence of terms τ, each of which are proofs that the predicate holds for the corresponding signature constructors of $\mathcal{R}.\gamma$

The result is a proof that the predicate T holds for its entire domain.

Concerning the computation law, it is conceptually straightforward though a little busy when spelled out formally. The action of $\mathit{case}(\mathcal{R})$ on a value built using the ith signature constructor $\mathcal{R}.\gamma(i)$ is to use the proof $\tau(i)$ corresponding to that constructor, giving to that proof the same arguments given to the signature

constructor.

We now show the implementation of the dependent eliminator for elaborated datatype signatures.

Definition 4.55 (Lambda encoding of $case(\mathcal{R})$)**.**

$$
\begin{array}{rcl}
\underset{i \leq \#\mathcal{R}.\Theta}{Case(\mathcal{R})(i)} & =_{\mathbf{df}} & (c_i : \Pi\, \Delta_i'.\, \mathcal{R}.D\ (\mathcal{R}.\gamma(i) \cdot \mathcal{R}.R\ \Delta_i')) \\
 & & \text{where } (c_i : \Pi\, \Delta_i'.\, D \in \mathcal{R}.\Xi) \\
\underset{i \leq \#\mathcal{R}.\Theta}{branches(\mathcal{R})(i)} & =_{\mathbf{df}} & \lambda\, \Delta_i'.\, \Lambda\, \mathcal{R}.v.\, c_i\ \Delta_i' \\
 & & \text{where } (c_i : \Pi\, \Delta_i'.\, D) \in \mathcal{R}.\Xi \\
case(\mathcal{R}) & =_{\mathbf{df}} & \chi\ (\forall\, \mathcal{R}.R : \star.\, \forall\, \mathcal{R}.D : isig(\mathcal{R}) \cdot \mathcal{R}.R \to \star. \\
 & & \quad \Pi\, Case(\mathcal{R}).\, \Pi\, \mathcal{R}.z : isig(\mathcal{R}) \cdot \mathcal{R}.R.\, \mathcal{R}.D\ z)\ \text{-} \\
 & & \Lambda\, \mathcal{R}.R.\, \Lambda\, \mathcal{R}.D.\, \lambda\, \mathcal{R}.\Sigma.\, \lambda\, \mathcal{R}.z. \\
 & & z.2 \cdot (lift(\mathcal{R}) \cdot \mathcal{R}.R \cdot \mathcal{R}.D)\ branches(\mathcal{R}) \\
 & & \quad \text{-}(intrView \cdot (isig(\mathcal{R}) \cdot \mathcal{R}.R)\ \beta\{z.1\}\ \text{-}z\ \text{-}\beta)
\end{array}
$$

- The telescope $Case(\mathcal{R})$ gives the types of the sequence of proofs τ, where type constructor variable $\mathcal{R}.D : isig(\mathcal{R}) \cdot \mathcal{R}.R \to \star$ stands in for the predicate T.
- Sequence $branches(\mathcal{R})$ is obtained by:
  - referring to each proof $\tau(i)$ via the identifier c_i;
  - η-expanding the proof to get $\lambda\, \Delta_i.\, c_i\ \Delta_i$; and
  - abstracting over $\mathcal{R}.v$, the erased identifier for the *View* evidence introduced by $lift(\mathcal{R})$.
- Finally, the eliminator $case(\mathcal{R})$ itself is defined starting with a type ascription (as we need to have its type synthesize).
  - We abstract over type variable $\mathcal{R}.R$ and type constructor variable $\mathcal{R}.D$ (standing in for S and T in the figure), then abstract over the sequence of proofs $Case(\mathcal{R})$ (standing in for τ in the figure) and scrutinee $z : isig(\mathcal{R}) \cdot \mathcal{R}.R$.
  - With $z.2$, we project out the inductive component of z and instantiate with predicate $lift(\mathcal{R}) \cdot \mathcal{R}.R \cdot \mathcal{R}.D : sig(\mathcal{R}) \cdot \mathcal{R}.R \to \star$.
  - We are next obligated to give a sequence of proofs that the lifted predicate holds for each of the computational constructors $\mathcal{R}.\tau$. We satisfy this obligation with $branches(\mathcal{R})$.
  - Finally, we need a proof that $z.1$ may be viewed as having type $isig(\mathcal{R}) \cdot \mathcal{R}.R$. This is immediate, as z has this type and is definitionally equal to $z.1$ (modulo erasure).

While the statement of static soundness is straightforward — either the term $case(\mathcal{R})$ satisfies the desired typing rule or it doesn't — for dynamic soundness we

must say a little more about the relation between the left- and right-hand sides of the equality. If $case(R)$ and the constructors were primitives, then the equality would in fact realized by a single step of computation. For our derived combinator, we desire that it be satisfied by a number of call-by-name reduction steps that is a function of the datatype declaration alone.

Theorem 4.56 (Static and dynamic soundness of dependent eliminator for datatype signatures)**.** *If* $\Gamma \vdash Decl[D, \Delta] \searrow (\Gamma', \mathcal{R})$ *valid, then* $case(\mathcal{R})$ *satisfies the typing law with context* Γ' *and satisfies the computational law in* $O(\#\Delta + \bigvee_{(c_i:\Pi\,\Delta_i.\,D)\in\Delta} \#|\Delta_i|)$ *call-by-name steps, where* $\#|\Delta_i|$ *is the number of unerased term variables declared in* Δ_i*.*

4.3 Chapter Conclusions

In this chapter, we have seen how Cedille elaborates datatype signatures and their constructors from the syntax of datatype declarations, and how such signatures are confirmed to be positive using evidence-producing inference rules for subtyping. Both sets of rules are shown to be sound (making them fit for purpose) and algorithmic (making them suitable for use in implementing an ITP such as Cedille). These two sets of judgments comprise the bulk of the technical complexity in elaborating inductive definitions in Cedille, and in the next chapter we will see them used together in the elaboration of the datatype proper as well as the elaboration of the μ- and σ-expressions used for course-of-values induction.

The remainder of this section discusses future and related work.

4.3.1 Future Work

The primary deficiency of the work presented in this chapter is the lack of support of datatypes with parameters and indices. Though the Cedille implementation

supports such datatypes, the decision to omit these features was made to simplify the presentation of the material. Formal treatment of datatype indices requires an arity-generic treatment of positivity (and also the type constructor least fixedpoint operation of the next chapter), and would clutter the inference rules for signature elaboration with additional uses of telescopes and sequences in the elaborated types and terms.

Compared with indices, datatype parameters are much less problematic for the presentation, though they do introduce their own wrinkle to positivity checking — one that is currently unresolved in the Cedille implementation. Consider the declaration of rose trees below.

```
data Rose (A: ⋆) : ⋆
= branch : A → List ·Rose → Rose .
```

Identifier *Rose* occurs positively in the parameter telescope of *branch*, since *List* is positive in its type parameter. However, since positivity checking is used in the surface language (see the next chapter), *List* must be treated as an opaque type constructor identifier. Solutions to this issue include dropping the positivity requirement altogether (requiring a suitable restriction to the case distinction scheme for datatypes), and allowing users to annotate parameters of datatype constructors like *List* with polarities, in which case the positivity checker could be used to confirm such annotations are sound and make use of the generated evidence when checking future declarations.

4.3.2 Related Work

Matthes's Monotone Inductive Types

In Section 4.1, we used the semantic notion of *monotonicity* rather than a syntactic criterion for positivity checking. This is in line with the work of Matthes [60] investigating the metatheoretic properties of type theories with inductive types by translating (*elaborating*) them to System F. Indeed, one can directly compare our inference rules for positivity and subtyping and the definition of the family of monotonicity witness in Section 5.1.1 of Matthes [60].

Like Matthes, our elaborated monotonicity witnesses are used to justify the existence of a least fixedpoint operation for certain type constructors (we do this in the next chapter). The main difference is that where Matthes generalizes the notion of monotonicity in a partial order to one suitable for the proof-relevant setting of type theory, our notion of monotonicity is the original (preorder-theoretic) one expressed in terms of types and type inclusions (as realized by retyping functions). *This is no minor distinction!* In the presence of dependent types, the use of retyping functions broadens significantly the family of inductive types which may be readily admitted in Cedille.

Consider the (admittedly contrived) declaration of a syntactically positive datatype (where $Not \cdot T =_{\mathbf{df}} T \to \forall\, X : \star.\, X$)

```
data Foo : ⋆
= p1 : Foo
| p2 : Foo
| pair : Π x1: Foo. Π x2: Foo. ∀ x3: Not ·{ x1 ≃ x2 }. Foo .
```

If we start with a more functorial notion of positivity (as done by Matthes), then

we reach a point where, given type variables R_1 and R_2 and term variables $x_1 : R_1$, $x_2 : R_2$, and $x_3 : \mathit{Not} \cdot \{x_1 \simeq x_2\}$, we must inhabit the type $\mathit{Not} \cdot \{f\ x_1 \simeq f\ x_2\}$ where $f : R_1 \to R_2$ is *arbitrary* (in this case, we clearly need that f is injective). However, since our the evidence-producing positivity checker is expressed in terms of retyping functions, datatype *Foo* is entirely unproblematic for us.

Dagand's Cosmology of Datatypes

In his dissertation, Dagand [27] shows how a class of inductive families may be elaborated to a closed universe of strictly positive inductive types. Though our kernel theories differ, we share the aim of reducing the relative complexity of the syntactic artifacts of datatype declarations through translation. Dagand elaborates strictly positive inductive families in their full generality, treating datatype declarations with parameters and indices — something which we omit in this dissertation. On the other hand, our positivity checker is not limited to strict positivity: we generate witnesses of monotonicity in the style of Matthes [60].

Impredicative Encodings of Datatypes

Böhm and Berarducci [14] undertook one of the earliest systematic translations of inductive types to impredicative encodings. They express strictly positive signatures in the language of universal algebra, and translate the datatypes obtained by these into Church encodings in System F. In a more modern presentation, Geuvers [39] shows how declarations of strictly positive datatypes can be expressed as Church, Scott, and Parigot ("Church-Scott") encodings in $\lambda 2\mu$ (System F extended with positive recursive types). Both correspond to the elaboration of the "compu-

tational" part of datatype signatures discussed in Section 4.2.3, though we support non-strictly positive types (and work in a dependent theory).

Our signature elaboration rules are based on a modified and simplified version of the recipe laid out by Stump [80] for deriving induction for natural numbers in CDLE. This dissertation generalizes the recipe to broad class of datatypes, making it suitable for use in the implementation of an ITP with support for inductive datatype declarations.

CHAPTER 5
ELABORATION OF COURSE-OF-VALUES INDUCTION

> One can envision an alternative history in which rather than change the underlying computational language of CC by adding primitive inductive types, we instead extend the language of types as proposed in this paper, and remain within a pure typed λ-calculus. So a system like Coq could have been founded instead on a PTS along the lines proposed here, rather than the Calculus of Inductive Constructions[.]
>
> Stump
> *From Realizability to Induction via Dependent Intersection*

The previous chapter concerned itself with the elaboration of datatype signatures, the component that gives a particular inductive type its unique character and its (dependent) case distinction scheme. In this chapter, we finish the task of elaborating datatypes by combining these signatures with a (derived) generic least fixedpoint operator, and elaborate the features of the Cedille surface language that enable course-of-values induction to CDLE types and terms defined with respect to this fixedpoint operator.

The goal for the design of the surface language is to make opaque (at least some of) the low-level details of the underlying encodings — for users of the surface language, these encodings should be merely implementation details that they may ignore in their day-to-day use. In particular, Cedille should expose to users neither to the impredicative encodings of the signatures of declared datatypes, nor the lambda encodings of constructors. This is in tension with the CDLE-idiomatic formulation of course-of-values induction we have seen previously (Section 3.4), where inductive proofs have access to evidence that the generic destructor *out* has the alternative

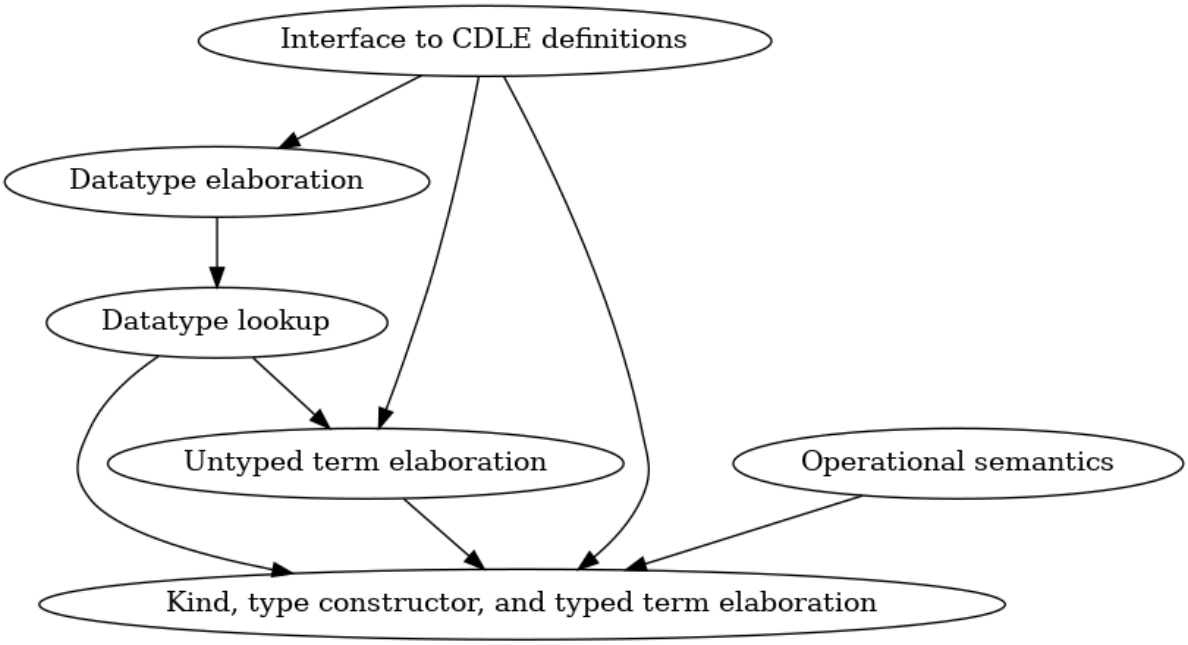


Figure 5.1: Chapter overview

typing $R \to F \cdot R$ (where R is a local, universally quantified type variable), since a direct analogue to this in the surface language would mean giving some representation of the signature F.

Cedille addresses this tension with *course-of-values pattern matching*, a feature that neatly abstracts away from the details of the elaborated signature by shifting the focus from the typing of the destructor to the typing of the case distinction scheme, where the advantage of the latter is that in the surface language the only necessary appearance of the signature implicit: the case trees of μ- and σ-expressions.

5.0.1 About This Chapter

Like the last chapter, in the present chapter we will be presenting a number of interconnected judgments. To assist in orienting the reader, Figure 5.1 shows an

illustration overviewing the components we will be discussion.

Prospectus. We begin with an axiomatic interface to some CDLE definitions in Section 5.1 that serves as a bridge between the source and target languages. We motivate this interface using an extended example of Cedille that uses course-of-values pattern matching, recursion, and induction. The semantics of Cedille is separated into four sections.

- Section 5.2 concerns the elaboration of datatypes, their constructors and exported constants, and contexts. This section covers the entirety of datatype elaboration, all datatype lookup performed in typed contexts, and the elaboration of a handful of type constructors and terms.
- Section 5.3 covers the elaborating type inference rules for μ- and σ-expressions for induction and course-of-values pattern matching, respectively. This section makes heavy use of meta-language records and introduces two auxiliary judgments for elaborating motive information and case trees.
- Section 5.4 covers the elaboration of untyped terms, defines the operational semantics of Cedille, and establishes dynamic soundness (for the elaboration of untyped terms).
- Finally, Section 5.5 covers the remaining elaborating inference rules for Cedille (kinds, type constructors, and terms), and establishes static and dynamic soundness. The section concludes by showing how static and dynamic soundness are used to import logical consistency and a qualified termination guarantee from

CDLE's meta-theory to CDLE.

The last gap in the story of elaborating Cedille is filled in Section 5.6, where we show how the axiomatic interface described in Section 5.1 is implemented in CDLE.

Reading Guide.

- It is **strongly recommended** to read Section 5.1.1 closely, to understand programming and proving with course-of-values pattern matching in Cedille in practice.
- It is **strongly recommended** to read Section 5.1.3 closely, and in particular to study Figure 5.6, as this gives an overview of the CDLE terms that justify induction and course-of-values pattern matching in Cedille.
- The reader **may skim or skip** Section 5.2, as it concerns linking signature elaboration with the axiomatic interface of the previous section and judgments for looking up the datatype declarations that introduce constructors and other datatype constants.
- It is **strongly recommended** to read Section 5.3, which details the elaboration of μ- and σ-expressions — the heart of course-of-values induction in Cedille.
- It is **strongly recommended** to read Section 5.4 on the dynamic semantics of Cedille, specifically the statements of Theorem 5.38 (reduction of a term preserves our ability to elaborate it), Theorem 5.39 (elaboration preserves convertibility), and Theorem 5.42 (elaboration preserves call-by-name reduction).
- It is **strongly recommended** to read Section 5.5.1 closely, in particular the

statement of Theorem 5.48. The reader **may skim or skip** the rest of Section 5.5, as it concerns the elaboration of the language of kinds, type constructors, and terms already present in CDLE.

- The reader **may skim or skip** Section 5.6, as it details a modification of the derivation by Firsov et al. [35] for deriving course-of-values induction in CDLE.

Original Contributions. The design of the surface language, as well as the proofs of static and dynamic soundness for the elaboration rules, is original work. An earlier form of these results appears in Jenkins et al. [43]. The derivation of course-of-values induction described in Section 5.6 is my variation of work originally by Firsov (see Firsov et al. [35]) to align the derivation with the Cedille surface language.

5.1 Interface for Inductive Datatype Encodings

In this section, we specify a CDLE interface that intermediates between the surface language constructs of Cedille for course-of-values induction and the impredicative and lambda encodings in CDLE to which they are ultimately translated. While here we present this interface axiomatically with inference rules, we will see in Section 5.6 how it can be implemented in terms of the generic framework of Firsov et al. [34] for efficient Mendler encodings.

5.1.1 Motivating Example for Course-Of-Values Recursion

To motivate this interface, we start by considering an extended example in Cedille of course-of-values recursion: an implementation of division using Cedille's type-based termination checking. This is shown in Figure 5.2, with two auxiliary definitions (datatype *Bool* and function *if*) in Figure 5.4 shown for completeness.

```
data Nat : ⋆
= zero : Nat
| succ : Nat → Nat .

predCV : ∀ N: ⋆. Is/Nat ·N ⇒ N → N
= Λ N. Λ is. λ n.
  σ<is> n {
  | zero → n
  | succ n' → n'
  }.

pred : Nat → Nat
= predCV -is/Nat.

minusCV : ∀ N: ⋆. Is/Nat ·N ⇒ N → Nat → N
= Λ N. Λ is. λ m. λ n.
  μ m-minus. n {
  | zero → m
  | succ n' → predCV -is (m-minus n')
  }.

minus : Nat → Nat → Nat
= minusCV -is/Nat.

lt : Nat → Nat → Bool
= λ m. λ n.
  σ<is/Nat> (minus (succ m) n) {
  | zero → true
  | succ l → false
  }.

div : Nat → Nat → Nat
= λ m. λ n. μ div-n. m {
  | zero → zero
  | succ l →
    [l' : Nat = to/Nat -isType/div-n l]
  - [dif : Type/div-n = minusCV -isType/div-n l (pred n)]
  - if (lt (succ l') n)
      zero
      (succ (div-n dif))
  }.
```

Figure 5.2: Division (course-of-values recursion)

For a discussion of the expressive power of course-of-values recursion, see Section 2.2. We walk through this code listing in detail, pointing out the surface language features needed to realize it. As we do, it may be helpful to refer back to the typing law for CDLE-idiomatic course-of-values induction discussed in Section 3.4, which we give again here (Figure 5.3) for convenience (recall that this too is an interface in need of an implementation).

$$\begin{aligned} \mathit{Alg} \cdot P \quad =_{\mathbf{df}} \quad & \forall\, R{:}\star.\, \forall\, c{:}\, \mathit{Cast} \cdot R \cdot \mu F.\, \mathit{View} \cdot (R \to F \cdot R)\ \beta\{\mathit{out}\} \Rightarrow \\ & (\Pi\, x{:}\, R.\, P\ (\mathit{elimCast}\ \text{-}c\ x)) \to \\ & \Pi\, \mathit{xs}{:}\, F \cdot R.\, P\ (\mathit{in}\ (\mathit{elimCast}\ \text{-}(\mathit{mono}\ c)\ \mathit{xs})) \end{aligned}$$

$$\frac{\Gamma \vdash P : \mu F \to \star \quad \Gamma \vdash \mathit{Alg} \cdot P \ni t_1 \quad \Gamma \vdash \mu F \ni t_2}{\Gamma \vdash \mathit{mcovi} \cdot P\ t_1\ t_2 \in P\ t_2}$$

Figure 5.3: CDLE course-of-values induction (typing)

```
data Bool : ⋆
= true : Bool
| false : Bool .

if : ∀ X: ⋆. Bool → X → X → X
= Λ X. λ b. λ t. λ f. σ<is/Bool> b {
  | true  → t
  | false → f
  } .
```

Figure 5.4: *Bool* and *if*

Datatype declarations

The first definition of Figure 5.2 is the declaration of datatype *Nat*. In addition to introducing the datatype name and its constructors, in Cedille a declaration of datatype D brings into scope two additional identifiers:

- a predicate $\texttt{Is/}D : \star \to \star$ on types that expresses the property that inhabitants of given type can be treated like datatype D for the purposes of pattern matching and can be *retyped* to type D ($\texttt{Is/}D$ serves to package together in the surface language the *Cast* and *View* evidence introduced when using CDLE-idiomatic course-of-values induction; see *Alg* in Figure 5.3);
- a witness $\texttt{is/}D : \texttt{Is/}D \cdot D$, which is a proof that the datatype D itself satisfies the predicate $\texttt{Is/}D$; and
- a polymorphic function $\texttt{to/}D : \forall\, X : \star.\, \texttt{Is/}D \cdot X \Rightarrow X \to D$, definitionally equal to $\lambda\, x.\, x$ in the source language, that for any type satisfying the predicate $\texttt{Is/}D$ retypes its inhabitants to type D.

Remark 5.1 (Special treatment of datatype identifier families). In the Cedille source language, identifiers containing a forward slash, such as $\texttt{Is/}Nat$, $\texttt{is/}Nat$, and $\texttt{to/}Nat$, are treated specially. They can *only* be used for names automatically introduced as part of the datatype system's type-based termination checking. For identifiers of the form $\texttt{Is/}D$ and $\texttt{is/}D$, the identifier D after the slash is used during type-checking to look up the associated datatype declaration.

Course-of-values pattern matching

The next definition of Figure 5.2, *predCV*, shows how predicate $\texttt{Is/}Nat$ is used to implement course-of-values pattern matching for *Nat*. Function *predCV*, which implements floored predecessor for natural numbers, abstracts over all types which satisfy the predicate $\texttt{Is/}Nat$, takes an inhabitant of such a type, and the result

it returns *has the same type.* In the body of the definition of *predCV*, we use a σ-expression to perform case analysis on the given number $n : N$, providing the assumed witness *is* : `Is/`$Nat \cdot N$ which says that we may pattern match on terms of type N just as we would for inhabitants of *Nat*. In the case for *succ*, the pattern variable n' has type N as well, and this is what we return.

So, course-of-values pattern matching *preserves the type of the scrutinee* in the recursive subdata in datatype constructor arguments, even when this type is not the datatype itself. This feature corresponds to the *View* evidence in Figure 5.3 that the destructor *out* can be viewed as having type $R \to F \cdot R$, where R is universally quantified.

Remark 5.2 (Size-preservation via parametric polymorphism). We may think of the type of *predCV* as saying it returns a natural number no larger than its unerased argument. Why? First, satisfaction of the predicate `Is/`*Nat* means that (as a first approximation) we may think of the type parameter N as an *opaque* stand-in for the datatype *Nat* itself. Because of this opacity (that is, if we reason parametrically — see Wadler's "Theorems for Free" [92]), the only type-specific operations we can perform on inhabitants of N are those permitted using the witness of type `Is/`$Nat \cdot N$: we can retype them to *Nat* with `to/`*Nat*, which would not help us to return something of type N; and we can perform case analysis, potentially revealing a predecessor having type N.

So, since parametricity tells us the only type-correct results are returning the original argument or some predecessor of it, we can conclude that *predCV* produces a result no larger than its argument.

Of course, users will wish to invoke the predecessor function on values of type *Nat* as well. Fortunately, we have witness `is/`*Nat* : `Is/`$Nat \cdot Nat$, which we use in *pred* to implement predecessor for "ordinary" natural numbers in terms of *predCV*. In this way, course-of-values pattern matching generalizes usual pattern matching.

Iteration

Next in Figure 5.2 is the definition of *minusCV*, which combines course-of-values pattern matching and Mendler-style iteration to implement a generalized floored subtraction function that preserves the type of its first number argument (that is, the *subtrahend*). In the body of *minusCV*, we use a μ-expression to define a recursive function *m-minus*, which we use to compute the result of subtracting the scrutinee n from m.

With μ-expressions, we introduce type-based termination checking in Cedille. Within the case branches of this μ-expression, the following identifiers are introduced locally:

- `Type/`*m-minus*, the type of recursive subdata in datatype constructor patterns, corresponding to the quantified type variable R in course-of-values induction (in particular, pattern variable n' has type `Type/`*m-minus*);
- `isType/`*m-minus*, a witness of type `Is/`*Nat* $\cdot$ `Type/`*m-minus*, corresponding to the *Cast* and *View* evidence of course-of-values induction; and
- *m-minus* : `Type/`*m-minus* $\to N$, the handle for making recursive calls.

In *minusCV*, we only use the last of these local identifiers, making the definition of subtraction *iterative* (see Section 2.2.3). We will see the full power of course-of-values recursion shortly with *div*. Also note that, as was the case for predecessor, subtraction for ordinary natural numbers can be readily implemented by specializing *minusCV* to the type *Nat*.

Course-of-values recursion

We pass over the definition of *lt*, as it introduces nothing new (*lt* returns true iff its first *Nat* argument is strictly less than its second argument), and turn to the last definition of the figure, *div*, which implements a form of division for natural numbers (when the divisor is *zero*, *div* returns *zero*). After abstracting over its two parameters m and n, in the body of *div* we use a μ-expression to recursively compute, via repeated subtraction, the result of diving m by n.

Like with *minusCV*, in the body of the μ-expression we introduce identifiers `Type/`*div-n* : $\star$, `isType/`*div-n* : `Is/`*Nat*$\cdot$`Type/`*div-n*, and *div-n* : `Type/`*div-n* $\rightarrow$ *Nat*. In the successor case, to make the code more readable we introduce two local definitions with the syntax $[x : T = t_1]$ - t_2.

- Term l' : *Nat* is the result of retyping pattern variable l : `Type/`*div-n* to *Nat* using `to/`*Nat* and the local evidence `isType/`*div-n*. Note that, by the erasure of `to/`*Nat*, l' is definitionally equal to l.
- Term *dif* : `Type/`*div-n* is the result of subtracting from l the predecessor of the divisor, *pred* n, using *minusCV*. We call *minusCV* with the local evidence `isType/`*div-n*.

In what remains of the definition of *div*, we want to test whether the number we were given at this step of recursion is strictly less than the divisor. However, the expression *lt* (*succ* l) n would not be type correct, as the domain of *succ* is *Nat*. Instead, we compare *succ* l' to n. When the comparison tells us that *succ* l' is not strictly less than the divisor, we use *div-n* to make a recursive call on *dif* and return

the successor of the result.

Note what has happened here: we have made a recursive call on the result of another function, *minusCV*, which is certainly not a pattern variable as would be required for syntactic termination checking. To make the termination checker accept the definition of *div*, the obligations on the user amount to coding against a polymorphic interface and inserting a few type annotations. What the user gains from this effort is *reusability*, such as the use of *predCV* and *minusCV* to define *div*.

5.1.2 Motivating Example for Course-Of-Values Induction

Though the example of division is enough of a starting point to explain the surface language syntax, it does not convey the full power of course-of-values *induction*. In particular, since recursive subdata in the patterns of μ-expressions have an abstract type (such as `Type/`*div-n*), we might anticipate that, in proofs, there could be a difficulty expressing the appropriate goal type for each branch: that the predicate holds for each of the datatype's constructor forms. To explain this point further, recall the use of the retyping function `to/`*Nat* in Figure 5.2 so that the term expressing the successor of pattern variable l was well-typed. In the successor case of an inductive proof of some predicate $T : Nat \to \star$, we similarly need a way to express that T holds for the successor of l, and to do that we need to retype l to type *Nat*.

We will now consider an example that shows how this issue is addressed in Cedille. Figure 5.5 partially lists a proof, using course-of-values induction, of the property that the result of division is always less than or equal to the dividend. In the figure, *Lte* expresses the less-than-or-equal relation, and the terms *lteMinus* and

```
lte : Nat → Nat → Bool
= λ m. λ n. lt m (succ n) .

Lte : Nat → Nat → ⋆
= λ m: Nat. λ n: Nat. { lte m n ≃ true }.

lteMinus : Π m: Nat. Π n: Nat. Lte (minus m n) m
= ...

lteTrans : Π l: Nat. Π m: Nat. Π n: Nat. Lte l m → Lte m n → Lte l n
= ...

lteDiv : Π m: Nat. Π n: Nat. Lte (div m n) m
= λ m. λ n. μ ih. m @(λ x: Nat. Lte (div x n) x) {
  | zero → β
  | succ l →
    [l' : Nat = to/Nat -isType/ih l]
  - σ<is/Bool> (lt (succ l') n)
    @(λ x: Bool.
      Lte (if x zero ((succ (div (minus l' (pred n)) n)))) (succ l')) {
    | true → β
    | false →
      [dif : Type/ih = minusCV -isType/ih l (pred n)]
    - [dif' : Nat = to/Nat -isType/ih dif]
    - [ih' : Lte (div dif' n) dif' = ih dif]
    - lteTrans (div dif' n) dif' l' ih' (lteMinus l' (pred n))
    }
  }.
```

Figure 5.5: Proof concerning division (course-of-values induction)

lteTrans are lemmas proving respectively that the result of subtraction is always less than the minuend and that *Lte* is a transitive relation.

In the main proof, *lteDiv*, we proceed by induction over the dividend m, with the induction hypothesis *ih* having the following type.

$$\Pi\, x : \texttt{Type/}ih.\ Lte\ (div\ (\texttt{to/}Nat\ \texttt{-isType/}ih\ x)\ n)\ (\texttt{to/}Nat\ \texttt{-isType/}ih\ x)$$

This type comes from the @(...) annotation, which is part of the syntax for μ- and σ-expressions. We have omitted this in previous Cedille code listings as it could be inferred; this annotation gives the *motive*, which is the type constructor into which we are eliminating. When m is the successor of some number, that number is bound as pattern variable l : `Type/`*ih*. In this branch, the goal type is the following.

$$Lte\ (div\ (succ\ (\texttt{to/}Nat\ \texttt{-isType/}ih\ l))\ n)\ (succ\ (\texttt{to/}Nat\ \texttt{-isType/}ih\ l))$$

So, in this type the appropriate retyping functions for the constructor arguments are automatically inserted. This is done using the telescope subtyping algorithm (see Figures 4.5 and 5.17).

To proceed with the proof, we must determine which branch of the conditional is taken in *div*, so we perform (dependent) case analysis on the Boolean comparison, aliasing the retyped predecessor as $l' : Nat$. In the case that $succ\ l'$ is not strictly less than the divisor n, we compute the difference *dif* : `Type/`*ih* between l and the predecessor of n using the type-preserving subtraction function *minusCV*. We introduce an alias $dif' : Nat$ for this quantity so that we may use the lemmas *lteMinus* and *lteTrans*. Crucially, however, when we invoke the induction hypothesis *ih*, *it is*

on dif.

5.1.3 Interface Between Surface and Target Languages

In this section, we specify axiomatically a *generic interface* that shall serve as the target for elaborating datatypes and course-of-values pattern matching and induction. We do this in order to simplify the coming formalisms. By "generic," we mean that the subjects of the typing rules are *parametric* in a type constructor $F : \star \to \star$ (we will see in the next section where this is instantiated with the elaborated datatype signatures from Chapter 4). This interface is presented in Figure 5.6 as a collection of typing and computation laws for CDLE definitions that mirror the Cedille features we saw in the previous section. The figure is separated into three parts, which we now detail.

Type least fixedpoint

The two typing rules listed in Figure 5.6a are relatively straightforward, being the formation and introduction rules for the type least fixedpoint operation. For the formation rule, nothing more is required than that the type constructor F has kind $\star \to \star$. For the introduction rule, we require that F is monotonic (see Section 3.3.3.2) to construct an expression of type μF from one of type $F \cdot \mu F$.

Before we discuss the remainder of the figure, we record a property of the type least fixedpoint μF (not listed in the figure) that is essential for providing a termination guarantee for datatype expressions in Cedille. Recall from our discussion of the meta-theoretic properties of CDLE (specifically, Section 3.2.2) that all closed terms whose types are included into a function type are call-by-name normalizing.

$$\frac{\Gamma \vdash F : \star \to \star}{\Gamma \vdash \mu F : \star}$$

$$\frac{\Gamma \vdash F : \star \to \star \quad \Gamma \vdash \mathit{Mono} \cdot F \ni t_1 \quad \Gamma \vdash F \cdot \mu F \ni t_2}{\Gamma \vdash \mathit{in} \cdot F \text{ -}t_1\ t_2 \in \mu F}$$

(a) Type least fixedpoint formation and introduction

$$\frac{\Gamma \vdash F : \star \to \star \quad \Gamma \vdash S : \star}{\Gamma \vdash \mathit{IsD} \cdot F \cdot S : \star} \qquad \frac{\Gamma \vdash F : \star \to \star \quad \Gamma \vdash \mathit{Mono} \cdot F \ni t}{\Gamma \vdash \mathit{isD} \cdot F \text{ -}t \in \mathit{IsD} \cdot F \cdot \mu F}$$

$$\frac{\Gamma \vdash F : \star \to \star \quad \Gamma \vdash S : \star \quad \Gamma \vdash \mathit{IsD} \cdot F \cdot S \ni s}{\Gamma \vdash \mathit{toD} \cdot F \cdot S \text{ -}s \in \mathit{Cast} \cdot S \cdot \mu F}$$

$$\begin{aligned} \mathit{toFD} \quad =_{\mathbf{df}} \quad & \chi\ (\forall\, X : \star \to \star.\, \mathit{Mono} \cdot X \Rightarrow \\ & \qquad \forall\, Y : \star.\, \mathit{IsD} \cdot X \cdot Y \Rightarrow \mathit{Cast} \cdot (X \cdot Y) \cdot (X \cdot \mu X)) \\ & \quad \text{-}\ \Lambda\, X.\, \Lambda\, x.\, \Lambda\, Y.\, \Lambda\, y. \\ & \qquad \mathit{intrCast}\ \text{-}(\mathit{elimCast}\ (x\ \text{-}(\mathit{toD} \cdot F \cdot S\ \text{-}y)))\ \text{-}(\lambda\, x.\, \beta) \end{aligned}$$

(b) Datatype course-of-values induction globals

$$\frac{\begin{array}{c}\Gamma \vdash F : \star \to \star \quad \Gamma \vdash \mathit{Mono} \cdot F \ni t_1 \quad \Gamma \vdash S : \star \\ \Gamma \vdash \mathit{IsD} \cdot F \cdot S \ni s \quad \Gamma \vdash t_2 \in S \quad \Gamma \vdash T : \kappa \quad \kappa \cong \mu F \to \star \\ \Gamma \vdash \Pi\, x : F \cdot S.\, T\ (\mathit{in} \cdot F \text{ -}t_1\ (\mathit{elimCast}\ \text{-}(\mathit{toFD} \cdot F \text{ -}t_1 \cdot S\ \text{-}s)\ x)) \ni t_3\end{array}}{\Gamma \vdash \mathit{sigma} \cdot F \text{ -}t_1 \cdot S\ \text{-}s\ t_2 \cdot T\ t_3 \in T\ (\mathit{toD} \cdot F \cdot S\ \text{-}s\ t_2)}\ *$$

$$\frac{\begin{array}{c}\Gamma \vdash F : \star \to \star \quad \Gamma \vdash \mathit{Mono} \cdot F \ni t_1 \\ \Gamma \vdash t_2 \in \mu F \quad \Gamma \vdash T : \kappa \quad \kappa \cong \mu F \to \star \quad x \notin \mathit{FV}(|t_3|) \\ \Theta =_{\mathbf{df}} (R : \star)(\text{-}\ x : \mathit{IsD} \cdot F \cdot R)(y : \Pi\, z : R.\, T\ (\mathit{elimCast}\ \text{-}(\mathit{toD} \cdot F \cdot R\ \text{-}x)\ z)) \\ \Gamma, \Theta \vdash \Pi\, z : F \cdot R.\, T\ (\mathit{in} \cdot F \text{ -}t_1\ (\mathit{elimCast}\ \text{-}(\mathit{toFD} \cdot F \text{ -}t_1 \cdot R\ \text{-}x)\ y)) \ni t_3\end{array}}{\Gamma \vdash \mathit{mu} \cdot F \text{ -}t_1\ t_2 \cdot T\ \Lambda\, R.\, \Lambda\, x.\, \lambda\, y.\, t_3 \in T\ t}\ *$$

$$\begin{array}{lcl} |\mathit{sigma}\ (\mathit{in}\ t_2)\ t_3| & \text{-}\text{-}\!\!\to^* & |t_3|\ |t_2| \\ |\mathit{mu}\ (\mathit{in}\ t_2)\ \lambda\, y.\, t_3| & \text{-}\text{-}\!\!\to^* & |t_3[(\lambda\, z.\, \mathit{mu}\ z\ \lambda\, y.\, t_3)/y]\ t_2| \end{array}$$

(c) Course-of-values pattern matching and induction

Figure 5.6: Axiomatic CDLE interface for Cedille datatype system

For all $F : \star \to \star$, μF is such a type.

Fact 5.3 (Inclusion of μF into a function type)**.** *For all $\emptyset \vdash F : \star \to \star$, there exists t, T_1, and T_2 such that $\emptyset \vdash t \in \mathit{Cast} \cdot \mu F \cdot (T_1 \to T_2)$.*

Datatype globals

Next, in Figure 5.6b we have constructs corresponding to Cedille's datatype globals for course-of-values pattern matching. In the surface language, for each datatype D we have globals `Is/`D, `is/`D, and `to/`D. In the target language, however, we can be parametric in the particular datatype signature and implement all such globals with *IsD*, *isD*, and *toD*. Expression $\mathit{isD} \cdot F$ -t is a proof of $\mathit{IsD} \cdot F \cdot \mu F$ when $F : \star \to \star$ is monotonic, as witnessed by $t : \mathit{Mono} \cdot F$. The first use of *IsD* is in the term *toD*, which turns any proof $s : \mathit{IsD} \cdot F \cdot S$ into a proof that S is included into μF (that is, a proof of $\mathit{Cast} \cdot S \cdot \mu F$; see Section 3.3.3). Finally, the auxiliary definition *toFD*, which is used in the typing rules for *sigma* and *mu*, combines this type inclusion with a monotonicity witness to yield an inclusion of $F \cdot S$ into $F \cdot \mu F$.

Course-of-values pattern matching and induction

Finally, Figure 5.6 gives the typing and computation laws for the CDLE interface underpinning σ- and μ-expressions in Cedille. We first note a subtlety concerning *mu* and *sigma*, one that warrants labeling their type inference with an asterisk: these are not terms by themselves, but rather names of two families of CDLE terms. Specifically, the situation for *mu* is for every F, t_1, t_2, T, and term of the form $\Lambda\, R.\, \Lambda\, x.\, \lambda\, y.\, t_3$ that satisfying the premises of the typing rule for *mu*, we have a term in the type-annotated target language $\mathit{mu} \cdot F$ -t_1 $t_2 \cdot T$ $\Lambda\, R.\, \Lambda\, x.\, \lambda\, y.\, t_3$; a similar situation holds for *sigma*. This caveat that *mu* and *sigma* are not stand-alone terms

is due to our need for call-by-name operational semantics in Cedille to be preserved in CDLE by elaboration (see Section 5.4). We could have introduced new notation for this, e.g., $mu\langle F, t_1, t_2, T, R, x, y, t_3\rangle$, but have chosen not to do so for the sake of cleaner notation. For a precise definition of these families, see Definitions 5.59 and 5.60.

In typing the rule for *sigma*, for a given monotonic datatype signature $F : \star \to \star$, the scrutinee t_2 may have any type $S : \star$ so long as we provide a proof s that the predicate $IsD \cdot F$ holds for S. Then, for a given predicate $T : \mu F \to \star$, to prove that T holds of the (retyping of) t_2, we must provide a proof t_3 (corresponding to the collection of case branches in σ-expressions) that, for every term of type $F \cdot S$, T holds of the μF term formed from the constructor *in* applied to the retyping of this term to type $F \cdot \mu F$. The computation law is essentially that of the case distinction scheme (see Section 2.2.2): we apply the collection of case branches t_3 to the revealed subdata. Note that we have a corresponding family of untyped terms $|sigma\ t_2\ t_3|$.

For the typing rule of μ, the scrutinee t_2 *must* have type μF. The type of the term t_3, corresponding to the collection of case branches in surface μ-expressions, is more complicated than for *sigma*. The type of t_3 is checked under a context extended by a type variable R, evidence that $IsD \cdot F$ holds for R, and an induction hypothesis y that the given predicate $T : \mu \to \star$ holds for (the retyping of) all subdata of type R; under this extended context, the type of t_3 says that it must return a proof that T holds for the image under constructor *in* of every F-collection of R subdata. The computation law for *mu* is that of Mendler-style iteration (see Section 2.2.3):

the collection of case branches is invoked on revealed subdata t_2 with an induction hypothesis that recursively invokes *mu* with those case branches. Again, observe that we have a corresponding family of untyped terms $|mu\ t_2\ \lambda\, y.\, t_3|$.

Remark 5.4. Recall that in Sections 3.4 and 3.5, we noted that the computational behavior of the CDLE-idiomatic formulation of course-of-values induction was akin to Mendler-style *iteration*, not Mendler-style course-of-values recursion. The same phenomenon that caused this is occurring here, too: terms playing the role of t_3 in the typing rule of *mu* are given erased evidence that *sigma* can be *used at a certain type.*

5.2 Static Semantics of Cedille (Pt. 1)

We now formally present the surface language Cedille using elaborating inference rules that use the interface listed in Figure 5.6. We begin with the grammar and erasure rules, shown in Figures 5.7, 5.8, and 5.9. To the grammar of type constructors, we add identifiers D for datatypes and, for each such identifier, a constant `Is/`D (formed from the concatenation of the string `Is/` with the identifier denoted by meta-variable D, see Remark 5.1). For convenience, in our presentation we shall treat datatype identifiers as distinct from other type constructor variables. To the grammar of terms, we add σ- and μ-expressions, constructors, and the families of constants `is/`D and `to/`D. Case trees bs, used by both σ- and μ-expressions, are either empty (written with whitespace) or a single branch followed by another case tree, where a branch comprises the token | (we use quotes to distinguish this from the meta-syntactic notation for BNF), a constructor c, a pattern variable sequence ϱ, and a term t (the branch body).

Definition 5.5 (Pattern variable sequence)**.** A pattern variable sequence ϱ is a sequence (see Definition 4.11) in which all entries are term or type variables. Meta-variable ϱ (along with primed and subscripted variants) ranges over pattern variable sequences.

type constructors	$S, T, F, T', T_1, \ldots$	$::=$	$\ldots$
datatypes			$\mid D$
datatype globals			$\mid \texttt{Is/}D$
terms	$s, t, t', t_1, \ldots$	$::=$	$\ldots$
case distinction			$\mid \sigma\langle s\rangle\ t\ @T\ \{\ bs\}$
recursion			$\mid \mu\ x.\ t\ @T\ \{\ bs\}$
constructors			$\mid c$
datatype globals			$\mid \texttt{is/}D \mid \texttt{to/}D$
case trees	bs	$::=$	
empty tree			
nonempty tree			$\mid$ ‘$\mid$‘ $c\ \varrho \to t\ bs$

Figure 5.7: Grammar of Cedille (extends Figure 3.1)

$$
\begin{array}{lcl}
|\sigma\langle s\rangle\ t\ @T\ \{\ c_i\ \varrho_i \to t_i\}_{i=\{1\ldots n\}}| & = & \sigma\ |t|\ \{\ c_i\ |\varrho_i| \to |t_i|\}_{i\in\{1\ldots n\}} \\
|\mu\ x.\ t\ @T\ \{\ c_i\ \varrho_i \to t_i\}_{i\in 1\ldots n}| & = & \mu\ x.\ |t|\ \{c_i\ |\varrho_i| \to |t_i|\}_{i\in\{1\ldots n\}} \\
|c| & = & c \\
|\texttt{is/}D| & = & \texttt{is/}D \\
|\texttt{to/}D| & = & \lambda\, x.\, x
\end{array}
$$

Figure 5.8: Erasure rules for annotated terms (extends Figure 3.8)

$$
\begin{array}{lcllcl}
|()| & = & () & |(t)\tau| & = & (|t|)|\tau| \\
|(\text{- } t)\tau| & = & |\tau| & |(\cdot T)\tau| & = & |\tau|
\end{array}
$$

Figure 5.9: Erasure rules for sequences

Pattern variable sequences are used primarily to express variables bound in constructor patterns. Analogous to our treatment of telescopes, we introduce notation for abstracting over pattern variable sequences in terms, and we will freely assume that we may rename entries such that they are distinct from each other and the declared variables of the local typing context.

Definition 5.6 (Pattern variable sequence abstraction)**.** Define $\lambda\,\varrho.\,t$ as follows.

$$\begin{array}{rcl} \lambda\,().\,t & = & t \\ \lambda\,(x)(\varrho).\,t & = & \lambda\,x.\,\lambda\,\varrho.\,t \\ \lambda\,(\text{-}\ x)\varrho.\,t & = & \Lambda\,x.\,\lambda\,\varrho.\,t \\ \lambda\,(\cdot X)\varrho.\,t & = & \Lambda\,X.\,\lambda\,\varrho.\,t \end{array}$$

In inference rules, it is useful to have a compact notation for the collection of branches that comprise a case tree. This motivates the following notational convention.

Notation 5.7 (Compact form for case trees)**.** We write $\{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}}$ for a case tree consisting of n case branches (where n is apparent from the syntax), and outside this notation c_i, ϱ_i, and t_i correspond to the ith constructor, pattern variable sequence, and branch body, respectively.

5.2.1 Datatype Declarations

We now bring together positivity checking and signature elaboration from Chapter 4 to give the inference rules that complete the elaboration of datatype declarations. This is shown in Figure 5.10. Following the convention established in Section 4.2.3, the main judgment $\Gamma \vdash_{\mathbf{data}} \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$, which we read "under Γ, declaration $\mathit{Decl}[D, \Delta]$ is well-formed and elaborates datatype information $\mathcal{D}$," elaborates a meta-linguistic record, and the meta-language functions we have defined for inductive signature constructor information, such as *isig* and *case*, are lifted in the expected way to records $\mathcal{D}$.

$$\boxed{\Gamma^- \vdash_{\mathbf{data}} \mathit{Decl}[D, \Delta]^- \searrow \mathcal{D}^+}$$

$$\boxed{\Gamma^- \vdash_{\mathbf{dcon}} (\mathcal{R}^-, i^-) \searrow (c^+, T^+)} \qquad \boxed{\Gamma^- \vdash_{\mathbf{dcon}} (\mathcal{R}^-, t^-, i^-) \searrow t'^+}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathbf{icon}} \mathit{Decl}[D, \Delta] \searrow \mathcal{R} \quad \Gamma \vdash +\mathit{isig}(\mathcal{R}) \rightsquigarrow t \\ (\Gamma \vdash_{\mathbf{dcon}} (\mathcal{R}, i) \searrow (c_i, T_i))_{i \in \{1 \ldots \#\Delta\}} \quad (\Gamma \vdash_{\mathbf{dcon}} (\mathcal{R}, t, i) \searrow t_i)_{i \in 1 \ldots \#\Delta} \\ \underset{i \leq \#\Delta}{\Delta'(i)} =_{\mathbf{df}} (c_i : T_i) \quad \underset{i \leq \#\Delta}{\theta(i)} =_{\mathbf{df}} (t_i)\end{array}}{\begin{array}{l}\Gamma \vdash_{\mathbf{data}} \mathit{Decl}[D, \Delta] \searrow \mathsf{record}\ \mathit{Data} \\ \quad \{\mathcal{R} = \mathcal{R}; \mathit{mono} = t; D = \mu(\mathit{isig}(\mathcal{R})) \\ \quad ; \Delta = \Delta'; \theta = \theta\}\end{array}}$$

$$\frac{(c_i : \Pi\, \Delta_i.\, \mathcal{R}.D) \in \mathcal{R}.\Xi}{\Gamma \vdash_{\mathbf{dcon}} (\mathcal{R}, i) \searrow (c_i, \Pi\, \Delta_i[\mu(\mathit{isig}(\mathcal{R}))/\mathcal{R}.D].\, \mu(\mathit{isig}(\mathcal{R})))}$$

$$\frac{(c_i : \Pi\, \Delta_i.\, \mathcal{R}.D) \in \mathcal{R}.\Xi}{\Gamma \vdash_{\mathbf{dcon}} (\mathcal{R}, t, i) \searrow \lambda\, \Delta_i.\, \mathit{in} \cdot \mathit{isig}(\mathcal{R})\ \text{-}t\ (\mathcal{R}.\gamma(i) \cdot \mu(\mathit{isig}(\mathcal{R}))\ \Delta_i)}$$

Figure 5.10: Datatype elaboration

Definition 5.8. In the meta-theory, define *Data* as the set of records whose fields are the inductive signature constructor information $\mathcal{R}$ (see Definition 4.49), a term *mono* (for the monotonicity witness), a type D (for the elaborated datatype), telescope Δ (for the constructor types), and sequence θ (for the elaborated datatype constructors). We let meta-variable $\mathcal{D}$ range over members of *Data*.

For each judgment listed in Figure 5.10 there is exactly one inference rule. In the first inference rule, to elaborate the subject $\mathit{Decl}[D, \Delta]$ we first invoke signature constructor elaboration (see Figure 4.13), obtaining $\mathcal{R}$, then invoke the positivity checker to confirm that the signature we obtain from this, $\mathit{isig}(\mathcal{R})$ (see Definition 4.41), is positive. Then, we elaborate the impredicative and lambda encodings of the constructors, which is accomplished by the auxiliary judgments. For these types, we replace all occurrences of the datatype identifier D with the elaboration of the datatype, and for these terms we abstract over each constructor's argument telescope Δ_i (where i is the index of the constructor) and precompose the generic constructor

in with the signature constructor $\mathcal{R}.\gamma(i)$.

As we did for signature elaboration (see Chapter 4), we define soundness for datatype elaboration, then prove (under the usual assumptions) that all elaborated datatype information is sound.

Definition 5.9 (Soundness of inductive datatype information)**.** Soundness of datatype elaboration is expressed by the following judgment and inference rule (see Definition 4.50 for the definition of soundness for elaboration inductive signature constructors).

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathbf{data}} \mathit{Decl}[D, \Delta] \searrow \mathcal{D} \quad \Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{D}.\mathcal{R})\ \mathit{valid} \\ \Gamma' \vdash \mathit{Mono} \cdot \mathit{isig}(\mathcal{D}) \ni \mathcal{D}.\mathit{mono} \quad \Gamma' \vdash \mathcal{D}.D : \star \quad \#\mathcal{D}.\Delta = \#\mathcal{D}.\theta \\ (\Gamma' \vdash \mathcal{D}.\theta(i) : \mathcal{D}.\Delta(i))_{i \in \{1 \ldots \#\mathcal{D}.\Delta\}}\end{array}}{\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{D})\ \mathit{valid}}$$

It is also straightforward to confirm that the inference rules of Figure 5.10 are mode-correct and syntax-directed, and that the judgment is decidable (again, up to assumptions).

Theorem 5.10 ($\vdash_{\mathbf{data}}$ soundness)**.** *Let* Γ*,* Γ'*, and* $\mathit{Decl}[D, \Delta]$ *be such that the conditions of Definition 4.26 (the assumptions for signature soundness). If* $\Gamma \vdash_{\boldsymbol{data}} \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$ *then* $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{D})$ *valid*

Proposition 5.11 ($\vdash_{\mathbf{data}}$ syntax-directedness and mode-correctness)**.** *The inference rules listed in Figure 5.10 are syntax-directed and mode-correct.*

Proof. By inspection of the rules. □

Theorem 5.12 ($\vdash_{\mathbf{data}}$ decidability)**.** *For all* Γ *and* $\mathit{Decl}[D, \Delta]$ *such that the assumption for signature decidability holds (Definition 4.32) it is decidable whether* $\Gamma \vdash_{\boldsymbol{data}} \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$*.*

For the proofs of Theorems 5.10 and 5.12, see Chapter C.

5.2.2 Contexts, Constants, and Constructors

To give the typing laws for μ- and σ-expressions in Cedille, we require judgments for looking up datatype declarations in the typing context and associating the appropriate types to constructors and constants. This is what we shall concern

$$\boxed{\vdash \Gamma^{-} \searrow \Gamma'^{+}}$$

$$\frac{}{\vdash \emptyset \searrow \emptyset} \quad \frac{\vdash \Gamma \searrow \Gamma' \quad \Gamma \vdash T : \star \searrow T'}{\vdash \Gamma, x{:}T \searrow \Gamma', x{:}T'} \quad \frac{\vdash \Gamma \searrow \Gamma' \quad \Gamma \vdash \kappa \searrow \kappa'}{\vdash \Gamma, X{:}\kappa \searrow \Gamma', X{:}\kappa'}$$

$$\frac{\vdash \Gamma \searrow \Gamma' \quad \Gamma \vdash_{\mathbf{data}} \mathit{Decl}[D, \Delta] \searrow \mathcal{D}}{\vdash \Gamma, \mathit{Decl}[D, \Delta] \searrow \Gamma'}$$

Figure 5.11: Context well-formedness and elaboration

ourselves with now. We will not address *how* datatype declarations are added to the typing context, as we have spilled no ink concerning the treatment of top-level definitions. We instead simply assume that they may be present in the typing context.

5.2.2.1 Context Elaboration

With this caveat given, we start with the judgment for checking and elaborating typing contexts. This is listed in Figure 5.11. The first three rules are congruence rules, with the empty context mapped to itself and contexts with declared variables preserving the identifier while associating with it the elaboration of the original classifier. Meanwhile, datatype declarations (which are not part of our target language) are removed completely.

Since the inference rules for context elaboration mainly defer to other judgments which have not yet been presented (recall that we are still in the middle of a mutual inductive definition of the source language), we leave for later the proofs that context elaboration is sound and decidable (see Section 5.5.1). However, we can at this point record the fact that context elaboration is syntax-directed and mode-correct.

Proposition 5.13 (Context elaboration syntax-directedness and mode-correctness)**.** *The inference rules listed in Figure 5.11 are syntax-directed and mode-correct.*

$$\boxed{\Gamma^- \vdash \mathit{Data}[D]^- : \mathit{Decl}[D,\Delta]^+ \searrow \mathcal{D}^+}$$

$$\frac{\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D,\Delta] \searrow \mathcal{D}}{\Gamma, x{:}T \vdash \mathit{Data}[D] : \mathit{Decl}[D,\Delta] \searrow \mathcal{D}} \quad \frac{\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D,\Delta] \searrow \mathcal{D}}{\Gamma, X{:}\kappa \vdash \mathit{Data}[D] : \mathit{Decl}[D,\Delta] \searrow \mathcal{D}}$$

$$\frac{D \neq D' \quad \Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D,\Delta] \searrow \mathcal{D}}{\Gamma, \mathit{Decl}[D',\Delta] \vdash \mathit{Data}[D] : \mathit{Decl}[D,\Delta] \searrow \mathcal{D}}$$

$$\frac{\Gamma \vdash_{\mathbf{data}} \mathit{Decl}[D,\Delta] \searrow \mathcal{D}}{\Gamma, \mathit{Decl}[D,\Delta] \vdash \mathit{Data}[D] : \mathit{Decl}[D,\Delta] \searrow \mathcal{D}}$$

$$\boxed{\Gamma^- \vdash \mathit{Con}[c]^- : \mathit{Decl}[D,\Delta]^+ \searrow (\mathcal{D}^+, i^+)}$$

$$\frac{\Gamma \vdash \mathit{Con}[c] : \mathit{Decl}[D,\Delta] \searrow (\mathcal{D}, i)}{\Gamma, x{:}T \vdash \mathit{Con}[c] : \mathit{Decl}[D,\Delta] \searrow (\mathcal{D}, i)} \quad \frac{\Gamma \vdash \mathit{Con}[c] : \mathit{Decl}[D,\Delta] \searrow (\mathcal{D}, i)}{\Gamma, X{:}\kappa \vdash \mathit{Con}[c] : \mathit{Decl}[D,\Delta] \searrow (\mathcal{D}, i)}$$

$$\frac{\Gamma \vdash (c, \mathit{Decl}[D',\Delta'], \#\Delta') : \mathit{Decl}[D,\Delta] \searrow (\mathcal{D}, i)}{\Gamma, \mathit{Decl}[D',\Delta'] \vdash \mathit{Con}[c] : \mathit{Decl}[D,\Delta] \searrow (\mathcal{D}, i)}$$

$$\boxed{\Gamma^- \vdash (c^-, \mathit{Decl}[D,\Delta]^-, j^-) : \mathit{Decl}[D',\Delta']^+ \searrow (\mathcal{D}^+, i^+)}$$

$$\frac{\Gamma \vdash \mathit{Con}[c] : \mathit{Decl}[D',\Delta'] \searrow (\mathcal{D}, i)}{\Gamma \vdash (c, \mathit{Decl}[D,\Delta], 0) : \mathit{Decl}[D',\Delta'] \searrow (\mathcal{D}, i)}$$

$$\frac{j \neq 0 \quad \exists T.\, \Delta(\#\Delta - j) = (c'{:}T) \quad c \neq c' \quad \Gamma \vdash (c, \mathit{Decl}[D,\Delta], j-1) \searrow (\mathcal{D}, i)}{\Gamma \vdash (c, \mathit{Decl}[D,\Delta], j) : \mathit{Decl}[D',\Delta'] \searrow (\mathcal{D}, i)}$$

$$\frac{j \neq 0 \quad \exists T.\, \Delta(\#\Delta - j) = (c{:}T) \quad \Gamma \vdash_{\mathbf{data}} \mathit{Decl}[D,\Delta] \searrow \mathcal{D}}{\Gamma \vdash (c, \mathit{Decl}[D,\Delta], j) : \mathit{Decl}[D,\Delta] \searrow (\mathcal{D}, j)}$$

Figure 5.12: Datatype declaration lookup

Proof. By inspection of the rules. □

5.2.2.2 Context Search

Our next step is looking up a datatype declaration using a given datatype identifier D, which we will need for typing μ- and σ-expressions as well as the datatype constants, and looking up a constructor. The rules for these lookup operations are

listed in Figure 5.12. The judgment $\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$ has a straightforward definition, with the inference rules decomposing the context until a datatype declaration for identifier D is found.

Remark 5.14. Some inference rules for datatype lookup shown in Figure 5.12 use the judgment $\Gamma \vdash_{\mathbf{data}} \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$ as a premise. A practical implementation of Cedille would store this information rather than redo the process of elaboration each time a search is performed. Since datatype lookup enjoys the property of *unicity* (the property that the outputs of the judgment are uniquely determined by the inputs; see Lemma E.8), the reader should understand such premises as being merely a notationally convenient way to refer to the elaborated datatype information.

For looking up constructors with judgment $\Gamma \vdash \mathit{Con}[c] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$, when we find a datatype declaration $\mathit{Decl}[D', \Delta']$ in the context we check the entries of Δ' with auxiliary judgment $\Gamma \vdash (c, \mathit{Decl}[D', \Delta'], j) : \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$, where j ranges from $\#\Delta'$ to 0. Each identifier of Δ' is checked to see if c is one of its declared variables until the index j reaches 0, in which case we continue looking through the typing context for another datatype declaration.

As we expect for judgments whose only aim is to retrieve information associated with a datatype in the typing context, the inference rules in Figure 5.12 are syntax-directed and mode-correct, and the judgments are decidable (assuming that the given context is well-formed).

Proposition 5.15 (Datatype lookup syntax-directedness and mode-correctness)**.** *The inference rules listed in Figure 5.12 are syntax-directed and mode-correct.*

Proof. By inspection of the rules.

- For $\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$, when we encounter a datatype declaration in the context, we decide which of the last two rules to apply based on a syntactic criterion (equality of identifiers).
- Immediate for the rules for $\Gamma \vdash \mathit{Con}[c] : \mathit{Decl}[D, \Delta] \searrow (\mathcal{D}, i)$.
- For judgment $\Gamma \vdash (c, \mathit{Decl}[D, \Delta], j) : \mathit{Decl}[D', \Delta'] \searrow (\mathcal{D}, i)$, we are testing a syntactic criterion (equality of identifiers) for the $(\#\Delta - j)$th entry.

□

$$\boxed{\Gamma^- \vdash t^- \in T^+ \searrow t'^+} \quad \boxed{\Gamma^- \vdash T^- \ni t^- \searrow t'^+}$$

$$\boxed{\Gamma^- \vdash T^- : \kappa^+ \searrow T'^+}$$

$$\frac{\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D}}{\Gamma \vdash D : \star \searrow \mathcal{D}.D} \quad \frac{\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D}}{\Gamma \vdash \texttt{Is/}D : \star \to \star \searrow \mathit{IsD} \cdot \mathit{isig}(\mathcal{D})}$$

$$\frac{\Gamma \vdash \mathit{Con}[c] : \mathit{Decl}[D, \Delta] \searrow (\mathcal{D}, i) \quad \Delta(i) = (c : \Pi\, \Delta'.\, D)}{\Gamma \vdash c \in \Pi\, \Delta'.\, D \searrow \mathcal{D}.\theta(i)}$$

$$\frac{\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D}}{\Gamma \vdash \texttt{is/}D \in \texttt{Is/}D \cdot D \searrow \mathit{isD} \cdot \mathit{isig}(\mathcal{D})\ \text{-}\mathcal{D}.\mathit{mono}}$$

$$\frac{\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D}}{\begin{array}{l}\Gamma \vdash \texttt{to/}D \in \forall\, X : \star.\, \texttt{Is/}D \cdot X \Rightarrow X \to D \\ \quad \searrow \chi\ (\forall\, X : \star.\, \mathit{IsD} \cdot \mathit{isig}(\mathcal{D}) \cdot X \Rightarrow X \to \mathcal{D}.D) \\ \quad\quad \text{-}\ \Lambda\, X.\, \Lambda\, x.\, \mathit{elimCast}\ \text{-}(\mathit{toD} \cdot \mathit{isig}(\mathcal{D}) \cdot X\ \text{-}x)\end{array}}$$

Figure 5.13: Elaboration of constants and constructors

Proposition 5.16 (Datatype lookup decidability)**.** *If* $\vdash \Gamma \searrow \Gamma'$ *then*

- *it is decidable whether* $\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$ *for some* Δ *and* $\mathcal{D}$*, and*
- *it is decidable whether* $\Gamma \vdash \mathit{Con}[c] : \mathit{Decl}[D, \Delta] \searrow (\mathcal{D}, i)$ *for some* D*,* Δ*, and* $i \in \mathbb{N}$*.*

Proof. Both proceed by induction on the derivation of $\vdash \Gamma \searrow \Gamma'$; for constructor lookup, we must prove by this by mutual induction with decidability of the auxiliary judgment $\Gamma \vdash (c, \mathit{Decl}[D, \Delta], j) : \mathit{Decl}[D', \Delta'] \searrow (\mathcal{D}, i)$. □

5.2.2.3 Elaborating Classification Rules

We at last are able to present inference rules for some of the main judgments: type synthesis and type checking for terms and kinding for type constructors. Only the rules for datatypes, constructors, and constants are presented in Figure 5.13. The type inference rules for the remaining language constructs of Cedille, as well as the statements of soundness and algorithmicness for these judgments, are presented in Sections 5.3 (μ- and σ-expressions) and 5.5 (everything else). We now discuss each rule in the order (left to right, top to bottom) they appear in the figure.

- Datatype identifier D has kind $\star$ when a declaration for D can be found in the context, elaborating datatype information $\mathcal{D}$, and the elaboration of D itself is $\mathcal{D}.D$ — this is the least fixedpoint of its (inductive) signature (see Figure 5.10).
- Constant $\texttt{Is/}D$ has kind $\star \to \star$ when D is a declared datatype, and its elaboration instantiates *IsD* from the interface (Figure 5.6a) with the datatype's elaborated signature.
- A constructor c is well-typed when a datatype declaration can be found introducing that constructor, and it synthesizes the type associated with it in the declaration. Its elaboration is given by the appropriate entry of field θ of the elaborated datatype information $\mathcal{D}$.
- Constant $\texttt{is/}D$ has type $\texttt{Is/}D \cdot D$ when D is a declared datatype, and its elaboration instantiates *isD* from the interface with the signature and its monotonicity witness elaborated from the declaration.
- Constant $\texttt{to/}D$ has type $\forall\, X : \star.\, \texttt{Is/}D \cdot X \Rightarrow X \to D$ (so, it retypes inhabitants of T to type D given evidence that $\texttt{Is/}D$ holds of T). Its elaboration is more involved than the other constructs in the figure: starting with a type ascription (so we may synthesize the type of this elaborated term), we abstract over types X and witnesses x and use *elimCast* to eliminate the type inclusion obtained from instantiating *toD* from the interface with x.

5.3 Static Semantics of Cedille (Pt. 2)

We now turn to the surface language constructs for course-of-values pattern matching and induction: σ- and μ-expressions. Compared to the other language

$$\boxed{\vdash \varrho^{-} := \Delta^{-}}$$

$$\frac{}{\vdash () := ()} \qquad \frac{\vdash \varrho := \Delta}{\vdash (y)\varrho := (x : T)\Delta}$$

$$\frac{\vdash \varrho := \Delta}{\vdash (\text{-}\ y)\varrho := (\text{-}\ x : T)\Delta} \qquad \frac{\vdash \varrho := \Delta}{\vdash (\cdot\ Y)\varrho := (X : \kappa)\Delta}$$

Figure 5.14: Valid pattern for telescope

constructs we have seen thus far, μ- and σ-expressions are significantly more complex. To make this complexity more manageable, we make use of a number of auxiliary judgments and definitions.

5.3.1 Validity of a Pattern Variable Sequence for a Telescope

Our first concern is ensuring that the branches in case trees are appropriately labeled. Specifically, for a branch labeled with $c\ \varrho$, where $c : \Pi\,\Delta.\,D$, we must ensure that pattern variable sequence ϱ provides a suitable binding for the constructor parameter telescope Δ. For example, case trees for datatype *Nat* should have the branch for *zero* binding neither terms nor types in the constructor pattern, and the branch for *succ* binding *exactly* one (unerased) term. We introduce the judgment $\vdash \varrho := \Delta$, to be read "pattern variable sequence ϱ is valid for telescope Δ," in Figure 5.14. It is inductively defined on ϱ and Δ, and can be affirmed precisely when $\#\varrho = \#\Delta$ and for each entry of ϱ, the corresponding entry of Δ has the same modality (unerased term, erased term, and type).

$$\boxed{\Gamma^- \vdash \mathfrak{M}^- \in \mathit{Decl}[D,\Delta]^+ \searrow \mathcal{M}^+}$$

$$\boxed{\Gamma^- \vdash (\mathit{Decl}[D,\Delta]^-, \mathfrak{M}^-, \mathfrak{L}^-) \ni \mathfrak{B}^- \searrow \mathcal{B}^+}$$

$$\frac{\begin{array}{c} y \notin \mathit{DV}(\Gamma) \quad \Gamma \vdash \mathfrak{M}.S : \star \searrow S' \quad \Gamma \vdash \mathfrak{M}.s \overset{\dashrightarrow^*}{\in} \mathtt{Is}/D \cdot S \searrow s' \quad S \cong \mathfrak{M}.S \\ \Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D,\Delta] \searrow \mathcal{D} \quad \Gamma \vdash \mathfrak{M}.T : \kappa \searrow T' \quad \kappa \cong D \to \star \end{array}}{\Gamma \vdash \mathfrak{M} \in \mathit{Decl}[D,\Delta] \searrow \mathsf{record}\ \mathit{Motive}\ \{\mathcal{D} = \mathcal{D}; S = S'; s = s'; T = T'; y = y\}}$$

$$\frac{\begin{array}{c} \mathfrak{B}.\gamma = \mathit{Seq}(\Delta) \quad (\vdash \mathfrak{B}.\varrho_i := \Delta_i)_{(c_i:\Pi\,\Delta_i.\,D)\in\Delta} \\ \Gamma \vdash \mathfrak{P}.S : \star \searrow S' \quad \Gamma \vdash \mathfrak{P}.s \overset{\dashrightarrow^*}{\in} \mathtt{Is}/D \cdot S \searrow s' \quad S \cong \mathfrak{P}.S \\ \left(\begin{array}{c} \Gamma \vdash \Delta_i[\mathfrak{P}.S/D] \leqslant_{\mathtt{to}/D\ \cdot\mathfrak{P}.s} \Delta_i \rightsquigarrow \xi_i \\ \Gamma, \Delta_i[\mathfrak{P}.S/D] \vdash \xi_i \odot \Delta_i : \Delta_i \searrow \xi_i' \\ \Gamma \vdash \Pi\,\Delta_i[\mathfrak{P}.S/D].\,\mathfrak{M}.T\ (c_i\ (\xi_i \odot \Delta_i)) \ni \lambda\,\mathfrak{B}.\varrho_i.\,\mathfrak{B}.t_i \searrow t_i' \end{array}\right)_{(c_i:\Pi\,\Delta_i.\,D)\in\Delta} \\ \tau(i) \underset{i\le\#\Delta}{=_{\mathbf{df}}} (t_i{}') \end{array}}{\Gamma \vdash (\mathit{Decl}[D,\Delta], \mathfrak{M}, \mathfrak{P}) \ni \mathfrak{B} \searrow \mathsf{record}\ \mathit{Branch}\ \{S = S'; s = s'; \tau = \tau; (\xi_i = \xi_i')_{i\in\{1\ldots\#\Delta\}}\}}$$

Figure 5.15: Typing of motives and case trees

5.3.2 Case Tree Motive and Branch Information

Our next pair of judgments, shown in Figure 5.15, factor out some tasks that are shared in the type inference and elaboration of both μ- and σ-expression. The first of these, the *motive information elaboration* judgment $\Gamma \vdash \mathfrak{M} \in \mathit{Decl}[D,\Delta] \searrow \mathcal{M}$, determines the datatype whose inhabitants the μ- or σ-expression acts upon; the second, $\Gamma \vdash (\mathit{Decl}[D,\Delta], \mathfrak{M}, \mathfrak{P}) \ni \mathfrak{B} \searrow \mathcal{B}$, confirms that case tree is well-typed for that datatype and motive information. These judgments make liberal use of meta-linguistic records. We start with $\mathfrak{M}$ and $\mathcal{M}$, the motive information for the source and target languages, respectively.

Definition 5.17 (Motive information (source language))**.** In the meta-theory, define *MotiveSrc* as the set of records whose fields are:

- a source type S (the type of the scrutinee);

- a source term s (a witness that connects the type S to some datatype); and
- a source type constructor T (the motive for elimination).

Meta-variable $\mathfrak{M}$ ranges over members of *MotiveSrc*

We construct members of *MotiveSrc* in the inference rules for μ- and σ-expressions (see Figures 5.16 and 5.17).

Example 5.18. *The source language motive information for predCV (Figure 5.2) is:*

- *type variable N;*
- *term variable is; and*
- *type constructor $\lambda\, x : Nat.\, N$.*

This would be written as record *MotiveSrc* $\{S = N; s = is; T = \lambda\, x : Nat.\, N\}$.

Definition 5.19 (Motive information (target language))**.** In the meta-theory, define *Motive* as the set of records whose fields are:

- datatype information $\mathcal{D}$;
- a target type S (the type of the scrutinee);
- a target term s (a witness that connects type S to some datatype);
- a target type constructor T (the motive for elimination); and
- a term variable y (used to instantiate the predicate of *case* in terms of T; see Definition 5.25).

Meta-variable $\mathcal{M}$ ranges over members of *Motive*.

We can now explain the sole inference rule for judgment $\Gamma \vdash \mathfrak{M} \in Decl[D, \Delta] \searrow \mathcal{M}$. First, we pick a variable y fresh with respect to the typing context Γ, and we elaborate the type of the scrutinee $\mathfrak{M}.S$, obtaining S'. Next, we synthesize the type of the witness $\mathfrak{M}.s$ and confirm this type reduces to a type of the form $\texttt{Is/}D \cdot S$ for some datatype identifier D and type S, elaborating s'; Type S must be convertible with $\mathfrak{M}.S$. Then, we look up the declaration of datatype D in the context, obtaining $Decl[D, \Delta]$ and elaborating $\mathcal{D}$. Finally, we elaborate the motive elimination $\mathfrak{M}.T$, obtaining T', and confirm that its kind is convertible with $D \to \star$ (for dependent

elimination of a datatype D, the motive must be a predicate over D). We gather $\mathcal{D}$, S', s', and T' into a *Motive* record, used in the elaboration of μ- and σ-expressions.

Inference for case trees

In judgment $\Gamma \vdash (Decl[D, \Delta], \mathfrak{M}, \mathfrak{P}) \ni \mathfrak{B} \searrow \mathcal{B}$, record $\mathfrak{B}$ stores the surface language information for the case branch bodies of the μ- or σ-expression, and record $\mathfrak{P}$ tells us how to type the recursive subdata bound in the constructor patterns — in μ-expressions, this differs from the corresponding fields of motive source information record, whereas in σ-expressions it is the same. The record $\mathcal{B}$ contains the elaboration of the case tree.

Definition 5.20 (Case branch bodies information (source language))**.** In the meta-theory, define *BranchSrc* as the set of records whose fields are:

- γ, a source term sequence consisting of constructors;
- a family of pattern variable sequences $(\varrho_i)_{i \in \{1 \ldots \#\gamma\}}$; and
- a family of source terms $(t_i)_{i \in \{1 \ldots \#\gamma\}}$ (the branch bodies).

Meta-variable $\mathfrak{B}$ ranges over members of *BranchSrc*.

Definition 5.21 (Pattern variable typing information (source language))**.** In the meta-theory, define *PattSrc* as the set of records whose fields are:

- a source type S (the type of the recursive subdata bound in constructor patterns); and
- a source term s (a witness connecting S to some declared datatype D).

Meta-variable $\mathfrak{P}$ ranges over members of *PattSrc*

Example 5.22. *The source language pattern variable typing information for minusCV is*

$$\mathsf{record}\ \mathit{PatternSrc}\ \{S = \texttt{Type/}m\text{-}\mathit{minus}; s = \texttt{isType/}m\text{-}\mathit{minus}\}$$

and the source language case branch bodies information for it is

$$\mathsf{record}\ \mathit{BranchSrc}\ \begin{array}{l}\{\gamma = (\mathit{zero})(\mathit{succ}); \varrho_1 = (); \varrho_2 = (n') \\ ; t_1 = m; t_2 = \mathit{predCV}\ \text{-}\mathit{is}\ (m\text{-}\mathit{minus}\ n')\}\end{array}$$

Definition 5.23 (Case tree information (target language))**.** In the meta-theory, define *Branch* as the set of records whose fields are:

- a target type S (the type of the recursive subdata);
- a target term s (a witness connecting S to the elaboration of some declared datatype D);

- a target term sequence τ (the case branches of the tree); and
- a family of sequences $(\xi_i)_{i \in \{1 \ldots \#\tau\}}$ (the retyped constructor pattern variables).

Meta-variable $\mathcal{B}$ ranges over *Branch*.

We now explain the inference rule for judgment $\Gamma \vdash (Decl[D, \Delta], \mathfrak{M}, \mathfrak{L}) \ni \mathfrak{B} \searrow \mathcal{B}$. First, we require that the sequence of constructors $\mathfrak{B}.\gamma$ exactly matches the sequence obtained from the constructor telescope Δ, and that each pattern variable sequence $\mathfrak{B}.\varrho_i$ is valid for the corresponding constructor parameter telescope Δ_i (where $(c_i : \Pi\, \Delta_i.\, D) \in \Delta$). Next, we elaborate the type $\mathfrak{P}.S$ of the recursive subdata in these constructor patterns, obtaining S', and synthesize the type of witness $\mathfrak{P}.s$. We confirm that this reduces to $\texttt{Is/}D \cdot S$ for some S convertible with $\mathfrak{P}.S$ and elaborate s'.

In order to type the case branches, we need to use the subtyping judgment to produce a family of sequences of retyping functions. Recall that this is because, in the types of the bound pattern variables, recursive occurrences of datatype D may be replaced with some other type (such as in *div* or *predCV* in Figure 5.2). For each $(c_i : \Pi\, \Delta_i.\, D) \in \Delta$, the case branch binds a sequence of pattern variables ϱ_i classified by telescope $\Delta_i[\mathfrak{P}.S/D]$, and we want a sequence of retyping functions ξ_i such that $\xi_i \odot \varrho_i$ has type Δ_i (see Definition 4.17 for the definition of $\odot$). This is obtained using the subtyping judgment for telescopes using the base subtyping assumption $\texttt{to/}D$ -$\mathfrak{P}.s$. We then elaborate as ξ_i' the retyping, using ξ_i, of the pattern variable sequences Δ_i (implicitly demoted to sequences, see Definition 4.15).

Finally, we check the type of each case branch. For the branch labeled with constructor c_i (where $(c_i : \Pi\, \Delta_i.\, D) \in \Delta$), we take the corresponding branch body $\mathfrak{B}.t_i$

and use it as the body of an abstraction over pattern variable sequence $\mathfrak{B}.\varrho_i$, forming the term $\lambda\,\mathfrak{B}.\varrho_i.\,\mathfrak{B}.t_i$. This captures the free occurrences of the pattern variable sequence. We then check that this term has type

$$\Pi\,\Delta_i[\mathfrak{P}.S/D].\,\mathfrak{M}.T\ (c_i\ (\xi_i \odot \Delta_i))$$

which is to say that each case branch is a proof that the predicate $\mathfrak{M}.T$ holds for the corresponding constructor form.

Remark 5.24. The branch information elaboration judgment of Figure 5.15 invokes the subtyping judgment for *source language types*. This is a straightforward extension of the subtyping rules of Figures 4.2 and 4.5 in which we treat datatype identifiers D and the family of constants $\mathtt{Is}/D$ as we do type variables: they are only in the subtyping relation if they are equal.

5.3.3 Elaboration of Case Trees

Before moving to the next section, the final piece we need is the assembly, from records $\mathcal{M}$ and $\mathcal{B}$ elaborated in Figure 5.15, of the target-language term given as the last argument to *sigma* and *mu* when elaborating σ- and μ-expressions. This is given below in Definition 5.25 as *ctree*.

Definition 5.25 (Elaboration of motives and case trees)**.**

$$\begin{array}{lll} mot(\mathcal{M},\mathcal{B}) & =_{\mathbf{df}} & \lambda\,\mathcal{M}.y\!:\!isig(\mathcal{M})\cdot\mathcal{B}.S.\\ & & \quad \mathcal{M}.T\ (in\cdot isig(\mathcal{M})\ \text{-}\mathcal{M}.mono\\ & & \qquad (elimCast\ \text{-}(toFD\cdot isig(\mathcal{M})\ \text{-}\mathcal{M}.mono\cdot\mathcal{B}.S\ \text{-}\mathcal{B}.s)\ \mathcal{M}.y))\\ ctree(\mathcal{M},\mathcal{B}) & =_{\mathbf{df}} & case(\mathcal{M})\cdot\mathcal{B}.S\cdot mot(\mathcal{M},\mathcal{B})\ \mathcal{B}.\tau \end{array}$$

Recall what we are trying to accomplish with *ctree*. In the source language, σ- and μ-expressions will need to reduce to the case branch labeled by the constructor from which the scrutinee is formed. In the target language, this is realized by invoking the (dependent) signature eliminator $case(\mathcal{M})$ (see Section 4.2.6; *case* is lifted to act on motive information $\mathcal{M}$ via projection, i.e., $case(\mathcal{M}.\mathcal{D}.\mathcal{R})$). Field $\mathcal{B}.S$ of the branch

information tells us how to instantiate the type of recursive subdata, then we use auxiliary definition $mot(\mathcal{M}, \mathcal{B})$ to instantiate the type constructor we are eliminating into (the motive), and we give the elaborated case branches $\mathcal{B}.\tau$ as the arguments to $case(\mathcal{M})$.

For the motive given to $case(\mathcal{M})$, our task is to use the original motive $\mathcal{M}.T : \mu(isig(\mathcal{M})) \to \star$ to form a type constructor of kind $isig(\mathcal{M}) \cdot \mathcal{B}.S \to \star$. This is obtained by precomposing $\mathcal{M}.T$ with the generic constructor in (instantiated with the datatype's signature $isig(\mathcal{M})$ and its proof of monotonicity $\mathcal{M}.mono$) precomposed with a retyping function of type $isig(\mathcal{M}) \cdot \mathcal{B}.S \to isig(\mathcal{M}) \cdot \mu(isig(\mathcal{M}))$. Note that the elaboration rules of Figure 5.15 ensure that the abstraction over variable $\mathcal{M}.y$ in $mot(\mathcal{M}, \mathcal{B})$ does not capture any free variables in its body.

We conclude this section by recording sufficient conditions for $mot(\mathcal{M}, \mathcal{B})$ being well-kinded and $ctree(\mathcal{M}, \mathcal{B})$ being well-typed.

Definition 5.26 (Branch and motive information soundness)**.** Soundness of elaborated motive information with respect to a datatype declaration and source motive information is expressed by the judgment $\Gamma \vdash (Decl[D, \Delta], \mathfrak{M}) \searrow (\Gamma', \mathcal{M})\ valid$, which is defined by the single inference rule given below.

$$\frac{\begin{array}{c}\Gamma \vdash Decl[D, \Delta] \searrow (\Gamma', \mathcal{M}.\mathcal{D})\ valid \quad \Gamma \vdash (Decl[D, \Delta], \mathfrak{M}) \searrow \mathcal{M} \quad \Gamma' \vdash \mathcal{M}.S : \star \\ \Gamma' \vdash IsD \cdot isig(\mathcal{D}) \cdot \mathcal{M}.S \ni \mathcal{M}.s \quad \Gamma' \vdash \mathcal{M}.T : \kappa' \quad \kappa' \cong \mathcal{D}.D \to \star\end{array}}{\Gamma \vdash (Decl[D, \Delta], \mathfrak{M}) \searrow (\Gamma', \mathcal{M})\ valid}$$

Soundness of the elaborated motive and case tree information with respect to a datatype declaration, source motive information, pattern variable typing information, and source branch bodies information is expressed by the judgment $\Gamma \vdash (Decl[D, \Delta], \mathfrak{M}, \mathfrak{P}, \mathfrak{B}) \searrow (\Gamma', \mathcal{M}, \mathcal{B})\ valid$, which is defined by the single inference

rule given below.

$$\frac{\begin{array}{c}\Gamma \vdash (\mathit{Decl}[D,\Delta],\mathfrak{M}) \searrow (\Gamma',\mathcal{M})\ \mathit{valid} \quad \Gamma \vdash (\mathit{Decl}[D,\Delta],\mathfrak{M},\mathfrak{P}) \ni \mathfrak{B} \searrow \mathcal{B} \\ \Gamma' \vdash \mathcal{B}.S : \star \quad \Gamma' \vdash \mathit{IsD} \cdot \mathit{isig}(\mathcal{M}) \cdot \mathcal{B}.S \ni \mathcal{B}.s \\ \left(\begin{array}{c}\Gamma' \vdash \mathcal{B}.\xi_i : \Delta_i[\mathcal{D}.D/D] \quad |\mathcal{B}.\xi_i| \cong |\mathit{Seq}(\Delta_i)| \\ \Gamma' \vdash \Pi\,\Delta_i[\mathcal{B}.S/D].\,\mathcal{M}.T\ (\mathcal{M}.\theta(i)\ \mathcal{B}.\xi_i) : \star \\ \Gamma' \vdash \Pi\,\Delta_i[\mathcal{B}.S/D].\,\mathcal{M}.T\ (\mathcal{M}.\theta(i)\ \mathcal{B}.\xi_i) \ni \tau(i)\end{array}\right)_{(c_i:\Pi\,\Delta_i.\,D \in \mathcal{M}.\Xi)}\end{array}}{\Gamma \vdash (\mathit{Decl}[D,\Delta],\mathfrak{M},\mathfrak{P},\mathfrak{B}) \searrow (\Gamma',\mathcal{M},\mathcal{B})\ \mathit{valid}}$$

Briefly, Definition 5.26 says the following.

- We have $\Gamma \vdash (\mathit{Decl}[D,\Delta],\mathfrak{M}) \searrow (\Gamma',\mathcal{M})\ \mathit{valid}$ when the datatype information $\mathcal{M}.\mathcal{D}$ is sound with respect to the declaration, $\mathcal{M}$ has been elaborated from $\mathfrak{M}$, the elaborated scrutinee type $\mathcal{M}.S$ is well-kinded, the elaborated witness $\mathcal{M}.s$ can be checked against the appropriate type, and the elaborated motive $\mathcal{M}.T$ is a predicate over the elaborated datatype.
- We have $\Gamma \vdash (\mathit{Decl}[D,\Delta],\mathfrak{M},\mathfrak{P},\mathfrak{B}) \searrow (\Gamma',\mathcal{M},\mathcal{B})\ \mathit{valid}$ when $\mathcal{M}$ is valid, $\mathcal{B}$ has been elaborated from $\mathfrak{M}$, $\mathfrak{P}$, and $\mathfrak{B}$, the elaborated type of the recursive subdata $\mathcal{B}.S$ is well-kinded and the witness $\mathcal{B}.s$ can be checked against the appropriate type, and, for each $(c_i : \Pi\,\Delta_i.\,D) \in \Delta$:
  - elaborations of the retyped constructor arguments $\mathcal{B}.\xi_i$ can be classified by telescope $\Delta_i[\mathcal{D}.D/D]$ and each of its non-erased entries are convertible with the corresponding identifier of Δ_i;
  - the type for the case branch, $\Pi\,\Delta_i[\mathcal{B}.S/D].\,\mathcal{M}.T\ (\mathcal{M}.\theta(i)\ \mathcal{B}.\xi)$ has kind $\star$ (this type says that the elaborated predicate holds for all values of the datatype constructed from the elaboration of c_i); and
  - each case branch can be checked against this type.

$$\boxed{\Gamma^- \vdash t^- \in T^+ \searrow t'^+}$$

$$\frac{\begin{array}{l}\Gamma \vdash t \in S \searrow t' \\ \mathfrak{M} =_{\mathbf{df}} \mathsf{record}\ \mathit{MotiveSrc}\ \{S = S; s = s; T = T\} \\ \mathfrak{P} =_{\mathbf{df}} \mathsf{record}\ \mathit{PattSrc}\ \{S = S; s = s\} \\ \mathfrak{B} =_{\mathbf{df}} \mathsf{record}\ \mathit{BranchSrc}\ \{\gamma_n(i) = (c_i); (\varrho_i = \varrho_i)_{i\in\{1\ldots n\}}; (t_i = t_i)_{i\in\{1\ldots n\}}\} \\ \Gamma \vdash \mathfrak{M} \in \mathit{Decl}[D, \Delta] \searrow \mathcal{M} \quad \Gamma \vdash (\mathit{Decl}[D, \Delta], \mathfrak{M}, \mathfrak{P}) \ni \mathfrak{B} \searrow \mathcal{B}\end{array}}{\begin{array}{l}\Gamma \vdash \sigma\langle s\rangle\ t\ @T\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \in T\ (\mathtt{to}/D\ \text{-}s\ t) \\ \quad \searrow \mathit{sigma} \cdot \mathit{isig}(\mathcal{M})\ \text{-}\mathcal{M}.\mathit{mono} \cdot \mathcal{M}.S\ \text{-}\mathcal{M}.s\ t' \cdot \mathcal{M}.T\ \mathit{ctree}(\mathcal{M}, \mathcal{B})\end{array}}$$

Figure 5.16: Course-of-values pattern matching: typing

Proposition 5.27. *If* $\Gamma \vdash (\mathit{Decl}[D, \Delta], \mathfrak{B}, \mathfrak{P}, \mathfrak{B}) \searrow (\Gamma', \mathcal{M}, \mathcal{B})$ *valid then*

1. $\Gamma' \vdash \mathit{mot}(\mathcal{M}, \mathcal{B}) : \Pi\, \mathcal{M}.y : \mathit{isig}(\mathcal{M}) \cdot \mathcal{B}.S.\star$
2. $\Gamma' \vdash \mathit{ctree}(\mathcal{M}, \mathcal{B}) \in \Pi\, \mathcal{M}.y : \mathit{isig}(\mathcal{M}) \cdot \mathcal{B}.S.\, \mathit{mot}(\mathcal{M}, \mathcal{B})\ y$

See Section F.3.1 for the proof of Proposition 5.27.

5.3.4 Course-Of-Values Pattern Matching

We now turn to the typing and elaboration of σ-expressions in Cedille, the source-language feature that enables course-of-values pattern matching. This is shown in Figure 5.16, which we now describe in detail. First, the scrutinee t has its type S synthesized and is elaborated to t'. Next, we organize the other components of the source σ-expression using the meta-language records we have previously defined.

- $\mathfrak{M}$ contains the source-language motive and scrutinee information: the type S of the scrutinee, the provided witness s permitting pattern matching on terms of type S, and the provided motive T (see Definition 5.17).
- $\mathfrak{P}$ contains the information needed for typing the pattern variable sequences bound in each case branch. Recall that course-of-values pattern matching pre-

serves the type of the scrutinee in the types of variables bound in constructor patterns. Thus, we populate this record with type S and witness s (see Definition 5.21).

- $\mathfrak{B}$ contains the case tree information: field γ contains the constructors labeling each branch, the family of fields $(\varrho_i)_{i\in\{1\ldots n\}}$ gives the pattern variable sequences bound in each branch, and the family $(t_i)_{i\in\{1\ldots n\}}$ gives the branch bodies (see Definition 5.20).

With premise $\Gamma \vdash \mathfrak{M} \in \mathit{Decl}[D, \Delta] \searrow \mathcal{M}$ we look up the declaration of the datatype we are eliminating and elaborate $\mathcal{M}$, the motive information for the target language. The last premise, $\Gamma \vdash (\mathit{Decl}[D, \Delta], \mathfrak{M}, \mathfrak{P}) \ni \mathfrak{B} \searrow \mathcal{B}$, checks that the case tree is well-typed with respect to declaration, motive information $\mathfrak{M}$, and pattern variable typing information $\mathfrak{P}$, and it elaborates the target language branch information $\mathcal{B}$. When these premises are affirmed, the type synthesized for the expression is T (`to`/D -s t) (that is, predicate T holds for the retyped scrutinee).

The term elaborated by this inference rule instantiates *sigma* from the interface of Figure 5.6 with the elaborated datatype signature and its monotonicity witness ($\mathit{isig}(\mathcal{M})$ and $\mathcal{M}.\mathit{mono}$, resp.), the elaborated type of the scrutinee and corresponding evidence connecting it to a declared datatype ($\mathcal{M}.S$ and $\mathcal{M}.s$, resp.), the elaboration of the scrutinee (t'), the elaborated motive ($\mathcal{M}.T$), and the elaboration of the case tree ($\mathit{ctree}(\mathcal{M}, \mathcal{B})$, see Definition 5.25).

5.3.5 Course-Of-Values Induction

The last new surface language construct, μ-expressions, enables course-of-values induction for Cedille datatypes. As we noted many times already, μ-expressions in Cedille allow the programmer to code inductive definitions *à la Mendler*, meaning that recursive subdata in constructor pattern sequences have an abstract type, with the handle for making a recursive call (or invoking the induction hypothesis) having this type as its domain. So, before explaining the elaborating type inference rule for μ-expressions, we first define the telescope with which we extend the local typing context when elaborating case branches.

Definition 5.28 (μ-expression locals)**.** For all datatype identifiers $D, \mathfrak{M} \in \mathit{MotiveSrc}$, and term identifiers x and y, define $\mathit{MuLoc}(D, \mathfrak{M}, x, y)$ as the following telescope.

$$(\texttt{Type/}x : \star)(\texttt{- isType/}x : \texttt{Is/}D \cdot \texttt{Type/}x)(x : \Pi\, y : \texttt{Type/}x.\, \mathfrak{M}.T\ (\texttt{to/}D\ \texttt{-isType/}x\ y))$$

When we use $\mathit{MuLoc}(D, \mathfrak{M}, x, y)$ in the typing of a μ-expression, D is the datatype we are eliminating, $\mathfrak{M}$ is the source-language motive and scrutinee information, x is the name of handle for the induction hypothesis, and y is a variable that is fresh with respect to the typing context and the identifiers declared in $\mathit{MuLoc}(D, \mathfrak{M}, x, y)$. The telescope $\mathit{MuLoc}(D, \mathfrak{M}, x, y)$ has as its entries: 1. the type variable standing in for recursive occurrences of the datatype in the types of constructor argument patterns; 2. a local witness connecting this type back to datatype D (in particular, allowing us to retype inhabitants of $\texttt{Type/}x$ to D and case analyze them further with σ-expressions); and 3. the handle for the induction hypothesis.

With Definition 5.28, we can explain the elaborating type inference rule for μ-expressions, shown in Figure 5.17.

$$\boxed{\Gamma^- \vdash t^- \in T^+ \searrow t'^+}$$

$$\frac{\begin{array}{c}\Gamma \vdash t \overset{\dashrightarrow^*}{\in} D \searrow t' \quad (\texttt{isType/}x \notin FV(|t_i|))_{i\in\{1\ldots n\}} \\ \mathfrak{M} =_{\mathbf{df}} \mathsf{record}\ \mathit{MotiveSrc}\ \{T = T; S = D; s = \texttt{is/}D\} \\ \mathfrak{P} =_{\mathbf{df}} \mathsf{record}\ \mathit{PattSrc}\ \{S = \texttt{Type/}x; s = \texttt{isType/}x\} \\ \mathfrak{B} =_{\mathbf{df}} \mathsf{record}\ \mathit{BranchSrc}\ \{\gamma_n(i) = (c_i); (\varrho_i = \varrho_i)_{i\in\{1\ldots n\}}; (t_i = t_i)_{i\in\{1\ldots n\}}\} \\ \Gamma \vdash \mathfrak{M} \in \mathit{Data}[D, \Delta] \searrow \mathcal{M} \quad y \notin \mathit{DV}(\Gamma) \cup \{\texttt{isType/}x, x\} \\ \Gamma, \mathit{MuLoc}(D, \mathfrak{M}, x, y) \vdash (\mathit{Data}[D, \Delta], \mathfrak{M}, \mathfrak{P}) \ni \mathfrak{B} \searrow \mathcal{B}\end{array}}{\begin{array}{l}\Gamma \vdash \mu\ x.\ t\ @T\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \in T\ t \\ \quad \searrow mu \cdot \mathit{isig}(\mathcal{M})\ \text{-}\mathit{mono}(\mathcal{M})\ t' \cdot \mathcal{M}.T\ (\lambda\, \mathit{MuLoc}(D, \mathfrak{M}, x, y).\ \mathit{ctree}(\mathcal{M}, \mathcal{B}))\end{array}}$$

Figure 5.17: Course-of-values induction: typing

- We first synthesize the type of the scrutinee t and elaborate t'. Unlike σ-expressions, the type of t must reduce to a datatype identifier D. We also check that the μ-local evidence $\texttt{isType/}x$ does not occur free within the erasures of the branch bodies.
- We next collect the motive information $\mathfrak{M}$, constructor pattern information $\mathfrak{P}$, and case tree information $\mathfrak{B}$. Observe that in μ-expressions, the type of the scrutinee recorded in $\mathfrak{M}$ (that is, D) differs from the type of recursive subdata (that is, $\texttt{Type/}x$) recorded in $\mathfrak{P}$.
- We look up the datatype that the μ-expression is eliminating with premise $\Gamma \vdash \mathfrak{M} \in \mathit{Decl}[D, \Delta] \searrow \mathcal{M}$, elaborating the target motive information, and generate a term variable y that is fresh with respect to the local typing context and term variables $\texttt{isType/}x$ and x.
- With premise $\Gamma, \mathit{MuLoc}(D, \mathfrak{M}, x, y) \vdash (\mathit{Decl}[D, \Delta], \mathfrak{M}, \mathfrak{P}) \ni \mathfrak{B} \searrow \mathcal{B}$, we check that the branches are well-typed, under the local typing context extended by

the telescope for μ-local identifiers, with respect to the datatype declaration and motive and pattern information, and elaborate target branch information $\mathcal{B}$.

- Finally, the elaboration of the μ-expression invokes *mu* from the interface listed in Figure 5.6. It is instantiated with the datatype's elaborated signature and monotonicity witness, the elaboration of the scrutinee (t') and motive $\mathcal{M}.T$, and the term corresponding to the case tree abstracts over the μ-local identifiers before invoking the elaborated signature's dependent eliminator with *ctree* (see Definition 5.25).

5.4 Dynamic Semantics of Cedille

Before we finish our presentation of Cedille's typing rules, we should explain the *dynamics* of the system, that is, reduction and elaboration of untyped (post-erasure) terms. As a dependent type theory, type checking in Cedille can involve checking convertibility of such terms. Now that we have added new constructs (untyped σ- and μ-expressions) to this term language, we can no longer rely solely on the meta-theoretic results of untyped lambda calculus for reasoning about computation.

The presences of *Top* and the φ axioms pose a significant difficulty in the design of the surface language dynamics. These are same features that cause CDLE's relatively weak termination guarantee (see Section 3.2.2). With the addition of σ- and μ-expressions, they now allow one to write well-typed Cedille terms which are neither values nor for which it is obvious how to assign an operational semantics.

Example 5.29. *Consider the following example.*

$$\chi\ (\{\mathit{true} \simeq \lambda\, x.\, x\} \Rightarrow \mathit{Bool})$$
$$\text{- } \Lambda\, x.\, \sigma\ (\varphi\ x\ \text{- }\ \mathit{true}\ \{\lambda\, x.\, x\})\ \{\ \mathit{true} \to \mathit{true} \mid \mathit{false} \to \mathit{false}\}$$

This term is well-typed, and its erasure is $\sigma\ (\lambda\, x.\, x)\ \{\ \mathit{true} \to \mathit{true} \mid \mathit{false} \to \mathit{false}\}$

In light of Example 5.29, we have the following choices for the design of Cedille.

1. Abandon an independent surface-language operational semantics and instead use the untyped lambda calculus by exposing elaborations of terms in the surface language.
2. Assign a surface-language operational semantics for such terms.
3. Be stuck with "stuck" terms.

We will, like in the current implementation of Cedille, proceed with the third option, though we evaluate the prospects for the other two in Section 5.7. This means that *progress*, an important aspect of type safety, fails to hold in Cedille.

Proposition 5.30 (Violation of Progress)**.** *There are closed well-typed terms in Cedille that are not values (under call-by-name operational semantics) and cannot be reduced further.*

We view Proposition 5.30 as reflecting an incompleteness of the surface language with respect to its target: though every reduction in the surface language corresponds can be simulated by convertibility in the target language, the same does not hold true in the opposite direction. In particular, while the surface language term shown in Example 5.29 is stuck, it's elaboration *can be reduce* because progress holds for the untyped lambda calculus.

$$\boxed{\Gamma^- \vdash ConU[\Delta']^- : Decl[D, \Delta]^+ \searrow \mathcal{D}^+}$$

$$\frac{\Gamma \vdash ConU[\Delta'] : Decl[D, \Delta] \searrow \mathcal{D}}{\Gamma, x{:}T \vdash ConU[\Delta'] : Decl[D, \Delta] \searrow \mathcal{D}} \quad \frac{\Gamma \vdash ConU[\Delta'] : Decl[D, \Delta] \searrow \mathcal{D}}{\Gamma, X{:}\kappa \vdash ConU[\Delta'] : Decl[D, \Delta] \searrow \mathcal{D}}$$

$$\frac{Seq(\Delta'') \neq Seq(\Delta') \quad \Gamma \vdash ConU[\Delta'] : Decl[D, \Delta]}{\Gamma, Decl[D'', \Delta''] \vdash ConU[\Delta'] : Decl[D, \Delta]}$$

$$\frac{Seq(\Delta) = Seq(\Delta') \quad \Gamma \vdash_{\mathbf{data}} Decl[D, \Delta] \searrow \mathcal{D}}{\Gamma, Decl[D, \Delta] \vdash ConU[\Delta'] : Decl[D, \Delta]}$$

Figure 5.18: Datatype lookup for untyped terms

5.4.1 Elaboration Of Untyped Terms

The next issue we tackle is the elaboration of untyped terms. Such terms now include constructors, σ- and μ-expressions, and the family of constants `is/`D, all of which are given meaning in CDLE by the elaborations of declared datatypes in the context. Therefore, though these terms are untyped, their elaboration must be stated with respect to a typing context so that we may look up the datatype declarations that govern the constructors and datatype constants.

The first judgment we consider, defined in Figure 5.18, is used to look up the datatype declaration that governs a case tree for σ- and μ-expressions. Read $\Gamma \vdash ConU[\Delta'] : Decl[D, \Delta] \searrow \mathcal{D}$ as "the identifiers present in Δ' match the constructors for declaration $Decl[D, \Delta]$ in Γ, which elaborates datatype information $\mathcal{D}$." The inference rules defining this judgment inspect the typing context until a datatype declaration whose constructor telescope Δ lists precisely the same identifiers as the telescope Δ' (which is moded as an input to the judgment).

$$\boxed{\Gamma_1^-; \Gamma_2^- \vdash t^- \searrow t'^+}$$

$$\frac{x \in DV(\Gamma_1, \Gamma_2)}{\Gamma_1; \Gamma_2 \vdash x \searrow x} \quad \frac{\Gamma_1; \Gamma_2, x : Top \vdash t \searrow t'}{\Gamma_1; \Gamma_2 \vdash \lambda\, x.\, t \searrow \lambda\, x.\, t'} \quad \frac{\Gamma_1; \Gamma_2 \vdash t_1 \searrow t_1' \quad \Gamma_1; \Gamma_2 \vdash t_2 \searrow t_2'}{\Gamma_1; \Gamma_2 \vdash t_1\ t_2 \searrow t_1'\ t_2'}$$

$$\frac{\Gamma_1 \vdash Con[c] : Decl[D, \Delta] \searrow (\mathcal{D}, i)}{\Gamma_1; \Gamma_2 \vdash c \searrow |\mathcal{D}.\theta(i)|} \quad \frac{\Gamma_1 \vdash Data[D] : Decl[D, \Delta] \searrow \mathcal{D}}{\Gamma_1; \Gamma_2 \vdash \texttt{is}/D \searrow |isD|}$$

$$\frac{\begin{array}{c}\Gamma_1; \Gamma_2 \vdash t \searrow t' \quad \Gamma_1 \vdash ConU \left[\underset{i \le n}{\Delta'(i) =_{\mathbf{df}} (c_i : Top)}\right] : Decl[D, \Delta] \searrow \mathcal{D} \\ (\#\varrho_i = \#|Seq(\Delta_i)| \text{ where } (c_i : \Pi\, \Delta_i.\, D) \in \Delta)_{i \in \{1 \ldots n\}} \\ (\Gamma_1; \Gamma_2 \vdash \lambda\, \varrho_i.\, t_i \searrow t_i')_{i \in \{1 \ldots n\}} \quad \underset{i \le n}{\tau(i) =_{\mathbf{df}} (t_i')}\end{array}}{\Gamma_1; \Gamma_2 \vdash \sigma\ t\ \{\ c_i\ \varrho_i \to t_i\}_{i \in \{1 \ldots n\}} \searrow |sigma\ t'\ (case(\mathcal{D})\ \tau)|}$$

$$\frac{\begin{array}{c}\Gamma_1; \Gamma_2 \vdash t \searrow t' \quad \Gamma_1 \vdash ConU \left[\underset{i \le n}{\Delta'(i) =_{\mathbf{df}} (c_i : Top)}\right] : Decl[D, \Delta] \searrow \mathcal{D} \\ (\#\varrho_i = \#|Seq(\Delta_i)| \text{ where } (c_i : \Pi\, \Delta_i.\, D) \in \Delta)_{i \in \{1 \ldots n\}} \\ (\Gamma_1; \Gamma_2, x : Top \vdash \lambda\, \varrho_i.\, t_i \searrow t_i')_{i \in \{1 \ldots n\}} \quad \underset{i \le n}{\tau(i) =_{\mathbf{df}} (t_i')}\end{array}}{\Gamma_1; \Gamma_2 \vdash \mu\ x.\ t\ \{c_i\ \varrho_i \to t_i\}_{i \in \{1 \ldots n\}} \searrow |mu\ t'\ (\lambda\, x.\, case(\mathcal{D})\ \tau)|}$$

Figure 5.19: Elaboration of untyped terms

As we expect, these inference rules for the lookup judgment describe algorithm.

Proposition 5.31 (Untyped datatype lookup syntax-directedness and mode-correctness)**.** *The inference rules listed in Figure 5.18 are syntax-directed and mode-correct.*

Proof. By inspection of the rules. In particular, note that the distinction between the last two rules (in which a datatype declaration is found) is whether identifier sequences differ syntactically. □

Proposition 5.32 (Untyped datatype lookup decidability)**.** *If* $\vdash \Gamma \searrow \Gamma'$*, then it is decidable whether* $\Gamma \vdash ConU[\Delta'] : Decl[D, \Delta] \searrow \mathcal{D}$ *for some* D*,* Δ*, and* $\mathcal{D}$*.*

Proof. By a straightforward induction on the assumed derivation of context elaboration. □

Next, in Figure 5.19 we turn to the elaboration of Cedille's untyped terms. The judgment uses two typing contexts: Γ_1, which contains the local typing information

at the point which we switched from typed to untyped terms, and Γ_2, whose sole purpose is to track lambda- and pattern-bound variables occurring in the untyped terms. Read judgment $\Gamma_1; \Gamma_2 \vdash t \searrow t'$ as "under contexts Γ_1 and Γ_2, untyped term t is well-formed and elaborates to t'."

We now describe the inference rules. Variables are checked to be declared in either Γ_1 or Γ_2 and elaborate to themselves. Lambda abstractions $\lambda\, x.\, t$ have the bound variable added to the second context with the type *Top* ascribed to them before elaborating the body t to t', and the elaboration is $\lambda\, x.\, t'$. The rule for applications $t_1\ t_2$ elaborates to $t_1'\ t_2'$ when t_1 elaborates to t_1' and t_2 elaborates to t_2'. For constructors c, we look up the datatype declaration $Decl[D, \Delta]$ declaring c and elaborate the erasure of its CDLE encoding $\mathcal{D}.\theta(i)$. For constant `is/`D, we merely ensure that datatype D is declared, and if so elaborate the erasure of isD.

The rules for elaborating σ- and μ-expressions share many of the same elaboration steps. First, the scrutinee t is elaborated to t'. Next, we look up the datatype declaration whose constructor telescope identifiers exactly match the identifiers $(c_i)_{i \in \{1 \ldots n\}}$; datatypes cannot be declared in untyped settings, so only the typing context Γ_1 is searched. Once the datatype declaration governing the case tree is found, we check with $\#\varrho_i = \#|Seq(\Delta_i)|$ that each pattern variable sequence ϱ_i bound in the case tree has the same number of entries as the *unerased* entries of the corresponding constructor argument telescope Δ_i. The final step in elaborating case trees is the elaboration of each of the branches, with the variables ϱ_i bound by the constructor patterns being lambda-abstracted over the branch bodies t_i, and the resulting elabo-

rated terms are collected into a sequence τ; for μ-expressions, this is done with the untyped context extended by the μ-bound variable x (the other μ-local identifiers are erased). For σ-expressions, the resulting elaboration is $|sigma\ t'\ (case(\mathcal{D})\ \tau)|$, and for μ-expressions it is $|mu\ t'\ (\lambda\, x.\ case(\mathcal{D})\ \tau)|$ (the μ-bound x in the source is replaced by the λ-bound x in the target).

Like lookup, the inference rules that define the judgment for elaborating untyped terms describes an algorithm.

Proposition 5.33 (Untyped term elaboration syntax-directedness and mode-correctness)**.** *The inference rules listed in Figure 5.19 are syntax-directed and mode-correct.*

Proof. By inspection of the rules. □

Proposition 5.34 (Untyped term elaboration decidability)**.** *If* $\vdash \Gamma_1, \Gamma_2 \searrow \Gamma'$*, then (for all untyped* t*) it is decidable whether* $\Gamma_1; \Gamma_2 \vdash t \searrow t'$ *for some* t'*.*

Proof sketch. By a straightforward structural induction on t, appealing to Proposition 5.16 and Proposition 5.32 where appropriate. □

5.4.2 Operational Semantics

Our final step in describing the dynamics of Cedille is to provide the operational semantics, shown in Figure 5.20. The main judgment, $t_1 \dashrightarrow t_2$, is a deterministic reduction relation that reduces the bodies of lambda-abstractions and case branches. Being such enables the following definition of convertibility.

Definition 5.35 (Convertibility)**.** Define $t_1 \cong t_2$ as the property that there exists untyped t_3 and t_4 such that $t_3 \not\dashrightarrow$, $t_4 \not\dashrightarrow$, $t_1 \dashrightarrow^* t_3$, $t_2 \dashrightarrow^* t_4$, and t_3 and t_4 are η-convertible.

Reduction relation $t_1 \dashrightarrow t_2$ is defined in terms of auxiliary judgments $t_1 \dashrightarrow_{\mathbf{a}} t_2$, which is reduction in an application term context, and $\tau_1 \dashrightarrow \tau_2$, which is reduction of a term sequence.

$$\boxed{t \dashrightarrow_{\mathbf{a}} t'} \quad \boxed{t \dashrightarrow t'} \quad \boxed{\tau \dashrightarrow \tau'}$$

$$\frac{\exists\, j \in \{1 \ldots n\}.\, c = c_j \quad \#\tau = \#\varrho_j}{\sigma\ (c\ \tau)\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{a}} t_j[\tau/\varrho_i]}\ \textsc{Sigma}$$

$$\frac{t \dashrightarrow_{\mathbf{a}} t'}{\sigma\ t\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{a}} \sigma\ t'\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}}}\ *_1$$

$$\frac{t \not\dashrightarrow_{\mathbf{a}} \quad \underset{i\le n}{\tau(i)} =_{\mathbf{df}} (t_i) \quad \tau \dashrightarrow \tau'}{\sigma\ t\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{a}} \sigma\ t\ \{\ c_i\ \varrho_i \to \tau'(i)\}_{i\in\{1\ldots n\}}}\ *_1$$

$$\frac{\begin{array}{c}\exists\, j \in \{1 \ldots n\}.\, c = c_j \quad \#\tau = \#\varrho_j \\ y \notin \bigcup_{i\in\{1\ldots n\}} (DV(\varrho_i) \cup FV(t_i)) \quad t_{\mathbf{m}} =_{\mathbf{df}} \lambda\, y.\, \mu\ x.\ y\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}}\end{array}}{\mu\ x.\ (c\ \tau)\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{a}} t_j[t_{\mathbf{m}}/x][\tau/\varrho_j]}\ \textsc{Mu}$$

$$\frac{t \dashrightarrow_{\mathbf{a}} t'}{\mu\ x.\ t\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{a}} \mu\ x.\ t'\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}}}\ *_2$$

$$\frac{t \not\dashrightarrow_{\mathbf{a}} \quad \underset{i\le n}{\tau(i)} =_{\mathbf{df}} (t_i) \quad \tau \dashrightarrow \tau'}{\mu\ x.\ t\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{a}} \mu\ x.\ t\ \{c_i\ \varrho_i \to \tau'(i)\}_{i\in\{1\ldots n\}}}\ *_2$$

$$\frac{}{(\lambda\, x.\, t_1)\ t_2 \dashrightarrow_{\mathbf{a}} t_1[t_2/x]}\ \textsc{Lambda}$$

$$\frac{t_1 \dashrightarrow_{\mathbf{a}} t_1'}{t_1\ t_2 \dashrightarrow_{\mathbf{a}} t_1'\ t_2}\ *_3 \qquad \frac{t_1 \not\dashrightarrow \quad t_2 \dashrightarrow t_2'}{t_1\ t_2 \dashrightarrow_{\mathbf{a}} t_1\ t_2'}\ *_3$$

$$\frac{t \dashrightarrow t'}{\lambda\, x.\, t \dashrightarrow \lambda\, x.\, t'} \qquad \frac{\Gamma \vdash t \dashrightarrow_{\mathbf{a}} t'}{\Gamma \vdash t \dashrightarrow t'}\ *_4$$

$$\frac{t \dashrightarrow t'}{(t)\tau \dashrightarrow (t')\tau} \qquad \frac{t \not\dashrightarrow \quad \tau \dashrightarrow \tau'}{(t)\tau \dashrightarrow (t)\tau'}$$

Figure 5.20: Full reduction for Cedille terms

We now describe the reduction rules, which for the sake of presentation make use of side conditions (which we explain as well).

- The main rule governing the reduction of σ-expressions is rule SIGMA. This rule applies when the scrutinee is of the form $c\ \tau$, where c is one of the constructors $(c_i)_{i\in\{1\ldots n\}}$ labeling the case tree and the length of τ matches the length of the pattern variable sequence at that label.
- Side condition $*_1$ is that rule SIGMA does not apply. In this case, to reduce a σ-expression either the scrutinee or one of the branch bodies must reduce. To reduce the branch bodies, we use auxiliary judgment $\tau \to \tau'$, which reduces the first entry of τ that it can.
- The main rule governing the reduction of μ-expressions is rule MU. The conditions under which this rule is applicable are the same as those for SIGMA. When a μ-expression reduces, the μ-bound variable occurring within the selected branch body is substituted by a lambda-expression that abstracts over the scrutinee with a fresh variable and duplicates the case tree.
- Side condition $*_2$ is that the rule MU does not apply, in which case to reduce the μ-expression either the scrutinee or one of the branch bodies must reduce.
- Rule LAMBDA is the familiar β-reduction rule.
- Side condition $*_3$ is that rue LAMBDA does not apply, in which case either the term in the function or the argument position must reduce for an application to reduce.
- Finally, we have the two rules for the main reduction relation $t_1 \dashrightarrow t_2$. If the

term on the left-hand side is a λ-abstraction, we reduce the bodies under the abstraction. Side condition $*_4$ is that the term is not a λ-abstraction, in which case we reduce using the auxiliary relation $\dashrightarrow_{\mathbf{a}}$.

Proposition 5.36 (Reduction unicity)**.** *If* $\vdash \Gamma_1, \Gamma_2 \searrow \Gamma'$ *then:*

1. $\Gamma_1; \Gamma_2 \vdash t \searrow t'$ *and* $t \dashrightarrow t_1$ *and* $t \dashrightarrow t_2$ *then* $t_1 = t_2$*; and*
2. *if* $\Gamma_1; \Gamma_2 \vdash t \searrow t'$ *and* $t \dashrightarrow_{\boldsymbol{a}} t_1$ *and* $t \dashrightarrow_{\boldsymbol{a}} t_2$ *then* $t_1 = t_2$
3. *assuming that for all* $i \in \{1 \ldots \#\tau\}$ *we have* $\Gamma_1; \Gamma_2 \vdash \tau(i) \searrow \tau'(i)$ *(for some* τ' *where* $\#\tau' = \#|\tau'| = \#\tau$*), if* $\tau \dashrightarrow \tau_1$ *and* $\tau \dashrightarrow \tau_2 m$ *then* $\tau_1 = \tau_2$*.*

Proof sketch. By mutual induction on the structure t and τ, making use of the fact that, when reducing t, the side conditions and mutually exclusive premises force that only one rule is applicable at a time □

Proposition 5.37 (Reduction decidability)**.** *If* $\vdash \Gamma_1, \Gamma_2 \searrow \Gamma'$*, then:*

1. *if* $\Gamma_1; \Gamma_2 \vdash t \searrow t'$*, it is decidable whether* $t \dashrightarrow t_1$ *for some* t_1*;*
2. *if* $\Gamma_1; \Gamma_2 \vdash t \searrow t'$*, it is decidable whether* $t \dashrightarrow_{\boldsymbol{a}} t_1$ *for some* t_1*; and*
3. *if (for all* $i \in \{1 \ldots \#\tau\}$*)* $\Gamma_1; \Gamma_2 \vdash \tau(i) \searrow \tau'(i)$ *(for some* τ' *with* $\#\tau' = \#|\tau'| = \#\tau$*), it is decidable whether* $\tau \dashrightarrow \tau_1$ *for some* τ_1*.*

Proof sketch. By a straightforward mutual induction on the structure of t and τ. □

5.4.3 Preservation and Dynamic Soundness

We have two main judgments for untyped terms, one for their elaboration and one for their reduction. We now must consider how they relate to each other. Our first concern is whether reduction preserves well-formedness of a term, i.e., if we reduce an untyped term that we are able to elaborate, can we guarantee that the result can be elaborated as well? Theorem 5.38 below answers this in the affirmative (for proofs of this and Theorem 5.39, see Section F.2 in the appendix).

Theorem 5.38 (Preservation)**.** *If* $\Gamma_1; \Gamma_2 \vdash t_1 \searrow t_1'$ *and* $t_1 \dashrightarrow t_2$ *then* $\Gamma_1; \Gamma_2 \vdash t_2 \searrow t_2'$ *for some* t_2'*.*

Our next concern is that the operational semantics of the surface language is *sound* with respect to the operational semantics of the target language. This concern

is two-fold: first, we wish to ensure that convertible terms in the surface language are elaborated to convertible terms in the target language; second, since we wish to import CDLE's termination guarantee (Theorem 3.19) into Cedille, we also require soundness of elaboration with respect to *call-by-name* operational semantics.

For the first concern, Theorem 5.39 establishes that each reduction in the surface preserves convertibility in the target (for the proof of this, see Section F.2). It then follows by an easy corollary that convertibility is preserved by elaboration.

Theorem 5.39 (Dynamic soundness (convertibility))**.** *If* $\Gamma_1; \Gamma_2 \vdash t_1 \searrow t_1'$ *and* $t_1 \dashrightarrow t_2$ *and* $\Gamma_1; \Gamma_2 \vdash t_2 \searrow t_2'$ *then* $t_1' =_{\beta\eta} t_2'$

Proposition 5.40 (η-conversion soundness)**.** *If* $\Gamma_1; \Gamma_2 \vdash t_1 \searrow t_1'$, $\Gamma_1; \Gamma_2 \vdash t_2 \searrow t_2'$, *and* t_1 *and* t_2 *are* η*-convertible, then* t_1' *and* t_2' *are* η*-convertible.*

Corollary 5.41. *If* $\Gamma_1; \Gamma_2 \vdash t_1 \searrow t_1'$ *and* $t_1 \cong t_2$ *and* $\Gamma_1; \Gamma_2 \vdash t_2 \searrow t_2'$ *then* $t_1' =_{\beta\eta} t_2'$.

For the second concern, we give a precise definition of call-by-name reduction in Cedille in Figure 5.21. Intuitively, $\dashrightarrow_{\mathbf{cbn}}$ is a restriction on the reduction relation that does not reduce the bodies of lambda-abstractions, nor arguments in applications, nor branch bodies in case trees. Theorem 5.21 below establish that each such reduction corresponds to one or more call-by-name reductions for the elaborated terms.

Theorem 5.42 (Dynamic soundness (simulation of call-by-name reduction))**.** *If* $\Gamma_1; \Gamma_2 \vdash t_1 \searrow t_1'$ *and* $t_1 \dashrightarrow_{cbn} t_2$ *and* $\Gamma_1; \Gamma_2 \vdash t_2 \searrow t_2'$ *then* $t_1' \dashrightarrow^+ t_2'$.

5.5 Static Semantics of Cedille (Pt. 3)

With our analysis of the dynamics concluded, we finally conclude our discussion of the surface with the remaining inference rules for its statics: the elaboration of kinds, type constructors, and the remaining annotated terms. This is shown in Figures 5.22 and 5.23. Most of the inference rules listed in these figures are congruence

$$\boxed{t_1^- \dashrightarrow_{\mathbf{cbn}} t_2^+}$$

$$\frac{\exists\, j \in \{1 \ldots n\}.\, c = c_j \quad \#\tau = \#\varrho_j}{\sigma\ (c\ \tau)\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{cbn}} t_j[\tau/\varrho_i]}\ \textsc{Sigma}$$

$$\frac{t \dashrightarrow_{\mathbf{cbn}} t'}{\sigma\ t\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{cbn}} \sigma\ t'\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}}}\ *_1$$

$$\frac{\begin{array}{c}\exists\, j \in \{1 \ldots n\}.\, c = c_j \quad \#\tau = \#\varrho_j \\ y \notin \bigcup_{i\in\{1\ldots n\}} (DV(\varrho_i) \cup FV(t_i)) \quad t_{\mathbf{m}} =_{\mathbf{df}} \lambda\, y.\, \mu\ x.\ y\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}}\end{array}}{\mu\ x.\ (c\ \tau)\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{cbn}} t_j[t_{\mathbf{m}}/x][\tau/\varrho_j]}\ \textsc{Mu}$$

$$\frac{t \dashrightarrow_{\mathbf{cbn}} t'}{\mu\ x.\ t\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{cbn}} \mu\ x.\ t'\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}}}\ *_2$$

$$\frac{}{(\lambda\, x.\, t_1)\ t_2 \dashrightarrow_{\mathbf{cbn}} t_1[t_2/x]}\ \textsc{Lambda}$$

$$\frac{t_1 \dashrightarrow_{\mathbf{cbn}} t_1'}{t_1\ t_2 \dashrightarrow_{\mathbf{cbn}} t_1'\ t_2}\ *_3$$

Figure 5.21: Call-by-name reduction for Cedille terms

rules, so rather than detail each of them we shall draw the reader's attention to the handful of rules in which we use elaboration of untyped terms. The reader may find it helpful to compare with the inference rules we presented for CDLE in Chapter 3 (Figures 3.2 and 3.7).

Remark 5.43. In Section 3.1.4, we introduced notation $\Gamma \vdash t \overset{\leftrightarrow}{\in} T$ to express that we are ranging over both the type synthesis and type checking judgments. Here, we do the same with the notation $\Gamma \vdash t \overset{\leftrightarrow}{\in} T \searrow t'$, except now we range over elaborating type synthesis and checking judgments.

- In the formation rule for equality types $\{t_1 \simeq t_2\}$, we require that the (erasures of) terms t_1 and t_2 are well-formed untyped terms.
- In the introduction form for equality, $\beta\{t_3\}$, we require that $|t_3|$ be a well-formed untyped term.

$$\boxed{\Gamma^- \vdash \kappa^- \searrow \kappa'^+}$$

$$\frac{}{\Gamma \vdash \star \searrow \star} \quad \frac{\Gamma \vdash T : \star \searrow T' \quad \Gamma, x{:}T \vdash \kappa \searrow \kappa'}{\Gamma \vdash \Pi\, x{:}T.\, \kappa \searrow \Pi\, x{:}T'.\, \kappa'} \quad \frac{\Gamma \vdash \kappa_1 \searrow \kappa_1' \quad \Gamma, X{:}\kappa \vdash \kappa_2 \searrow \kappa_2'}{\Gamma \vdash \Pi\, X{:}\kappa_1.\, \kappa_2 \searrow \Pi\, X{:}\kappa_1'.\, \kappa_2'}$$

(a) Elaboration of kinds

$$\boxed{\Gamma^- \vdash T^- : \kappa^+ \searrow T'^+}$$

$$\frac{(X{:}\kappa) \in \Gamma}{\Gamma \vdash X : \kappa \searrow X} \qquad \frac{\Gamma \vdash \kappa \searrow \kappa' \quad \Gamma, X{:}\kappa \vdash T : \star \searrow T'}{\Gamma \vdash \forall\, X{:}\kappa.\, T : \star \searrow \forall\, X{:}\kappa'.\, T'}$$

$$\frac{\Gamma \vdash T_1 : \star \searrow T_1' \quad \Gamma, x{:}T_1 \vdash T_2 : \star \searrow T_2'}{\Gamma \vdash \forall\, x{:}T_1.\, T_2 : \star \searrow \forall\, x{:}T_1'.\, T_2'} \qquad \frac{\Gamma \vdash T_1 : \star \searrow T_1' \quad \Gamma, x{:}T_1 \vdash T_2 : \star \searrow T_2'}{\Gamma \vdash \Pi\, x{:}T_1.\, T_2 : \star \searrow \Pi\, x{:}T_1'.\, T_2'}$$

$$\frac{\Gamma \vdash T_1 : \star \searrow T_1' \quad \Gamma, x{:}T_1 \vdash T_2 : \kappa \searrow T_2'}{\Gamma \vdash \lambda\, x{:}T_1.\, T_2 : \Pi\, x{:}T_1.\, \kappa \searrow \lambda\, x{:}T_1'.\, T_2'} \qquad \frac{\Gamma \vdash \kappa_1 \searrow \kappa_1' \quad \Gamma, X{:}\kappa_1 \vdash T : \kappa_2 \searrow T'}{\Gamma \vdash \lambda\, X{:}\kappa_1.\, T : \Pi\, X{:}\kappa_1.\, \kappa_2 \searrow \lambda\, X{:}\kappa_1'.\, T'}$$

$$\frac{\Gamma \vdash T : \Pi\, x{:}T_1.\, \kappa \searrow T' \quad \Gamma \vdash T_1 \ni t \searrow t'}{\Gamma \vdash T\ t : \kappa[t/x]^{T_1} \searrow T'\ t'} \qquad \frac{\Gamma \vdash T_1 : \Pi\, X{:}\kappa_2.\, \kappa_1 \searrow T_1' \quad \Gamma \vdash T_2 : \kappa_3 \searrow T_2' \quad \kappa_2 \cong \kappa_3}{\Gamma \vdash T_1 \cdot T_2 : \kappa_1[T_2/X] \searrow T_1' \cdot T_2'}$$

$$\frac{\Gamma \vdash T_1 : \star \searrow T_1' \quad \Gamma, x{:}T_1 \vdash T_2 : \star \searrow T_2'}{\Gamma \vdash \iota\, x{:}T_1.\, T_2 : \star \searrow \iota\, x{:}T_1'.\, T_2'} \qquad \frac{\Gamma; \emptyset \vdash |t_1| \searrow t_1' \quad \Gamma; \emptyset \vdash |t_2| \searrow t_2'}{\Gamma \vdash \{t_1 \simeq t_2\} : \star \searrow \{t_1' \simeq t_2'\}}$$

(b) Elaboration of type constructors

Figure 5.22: Elaboration of Cedille classifiers (congruence rules)

$$\boxed{\Gamma^- \vdash t^- \in T^+ \searrow t'^+} \quad \boxed{\Gamma^- \vdash T^- \ni t^- \searrow t'^+}$$

$$\frac{(x{:}T) \in \Gamma}{\Gamma \vdash x \in T \searrow x} \qquad \frac{\Gamma \vdash t \in T' \searrow t' \quad T' \cong T}{\Gamma \vdash T \ni t \searrow t'} \ *$$

$$\frac{T \dashrightarrow^* \Pi\, x{:}T_1.\, T_2 \quad \Gamma, x{:}T_1 \vdash T_2 \ni t \searrow t'}{\Gamma \vdash T \ni \lambda\, x.\, t \searrow \lambda\, x.\, t'} \qquad \frac{\Gamma \vdash t \overset{\dashrightarrow^*}{\in} \Pi\, x{:}T_1.\, T_2 \searrow t' \quad \Gamma \vdash T_1 \ni t_1 \searrow t_1'}{\Gamma \vdash t\ t_1 \in T_2[t_1/x]^{T_1} \searrow t'\ t_1'}$$

$$\frac{T' \dashrightarrow^* \forall\, X{:}\kappa.\, T \quad \Gamma, X{:}\kappa \vdash T \ni t \searrow t'}{\Gamma \vdash T' \ni \Lambda\, X.\, t \searrow \Lambda\, X.\, t'} \qquad \frac{\Gamma \vdash t \overset{\dashrightarrow^*}{\in} \forall\, X{:}\kappa.\, T_2 \searrow t' \quad \Gamma \vdash T_1 : \kappa_1 \searrow T_1' \quad \kappa_1 \cong \kappa}{\Gamma \vdash t \cdot T_1 \in T_2[T_1/X] \searrow t' \cdot T_1'}$$

$$\frac{T \dashrightarrow^* \forall\, x{:}T_1.\, T_2 \quad \Gamma, x{:}T_1 \vdash T_2 \ni t \searrow t' \quad x \notin FV(|t|)}{\Gamma \vdash T \ni \Lambda\, x.\, t \searrow \Lambda\, x.\, t'} \qquad \frac{\Gamma \vdash t_2 \overset{\dashrightarrow^*}{\in} \forall\, x{:}T_1.\, T_2 \searrow t_2' \quad \Gamma \vdash T_1 \ni t_1 \searrow t_1'}{\Gamma \vdash t_2\ \text{-}t_1 \in T_2[t_1/x]^{T_1} \searrow t_2'\ \text{-}t_1'}$$

$$\frac{T \dashrightarrow^* \iota\, x{:}T_1.\, T_2 \quad \Gamma \vdash T_1 \ni t_1 \searrow t_1' \quad \Gamma \vdash T_2[t_1/x]^{T_1} \ni t_2 \searrow t_2' \quad |t_1| \cong |t_2|}{\Gamma \vdash T \ni [t_1, t_2] \searrow [t_1', t_2']}$$

$$\frac{\Gamma \vdash t \overset{\dashrightarrow^*}{\in} \iota\, x{:}T_1.\, T_2 \searrow t'}{\Gamma \vdash t.1 \in T_1 \searrow t'.1} \qquad \frac{\Gamma \vdash t \overset{\dashrightarrow^*}{\in} \iota\, x{:}T_1.\, T_2 \searrow t'}{\Gamma \vdash t.2 \in T_2[t.1/x] \searrow t'.2}$$

$$\frac{T \dashrightarrow^* \{t_1 \simeq t_2\} \quad \Gamma; \emptyset \vdash |t_3| \searrow t_3' \quad |t_1| \cong |t_2|}{\Gamma \vdash T \ni \beta\{t_3\} \searrow \beta\{t_3'\}}$$

$$\frac{\Gamma \vdash t \in T_1 \searrow t' \quad T_1 \cong \{\lambda\, x.\, \lambda\, y.\, x \simeq \lambda\, x.\, \lambda\, y.\, y\}}{\Gamma \vdash T \ni \delta\ \text{-}\ t \searrow \delta\ \text{-}\ t'}$$

$$\frac{\begin{array}{c}\Gamma \vdash t \overset{\dashrightarrow^*}{\in} \{t_1 \simeq t_3\} \searrow t' \quad \Gamma; \emptyset \vdash |t_2| \searrow t_2' \quad |t_3| \cong |t_2| \\ \Gamma \vdash T_1[t_2/x] : \star \searrow T_1' \quad \Gamma \vdash T_1[t_2/x] \ni t_1 \searrow t_1' \quad T_1[t_1/x] \cong T\end{array}}{\Gamma \vdash T \ni \rho\ t\ @x\langle t_2\rangle.T_1\ \text{-}\ t_1 \searrow \rho\ t'\ @x\langle t_2'\rangle.T_1'\ \text{-}\ t_1'}$$

$$\frac{\Gamma \vdash t_2 \overset{\leftrightarrow}{\in} T \searrow t_2' \quad \Gamma; \emptyset \vdash |t_3| \searrow t_3' \quad \Gamma \vdash \{t_2 \simeq t_3\} \ni t_1 \searrow t_1'}{\Gamma \vdash \varphi\ t_1\ \text{-}\ t_2\ \{t_3\} \overset{\leftrightarrow}{\in} T \searrow \varphi\ t_1'\ \text{-}\ t_2'\ \{t_3'\}}$$

$$\frac{\Gamma \vdash T : \star \searrow T' \quad \Gamma \vdash T \ni t \searrow t'}{\Gamma \vdash \chi\ T\ \text{-}\ t \in T \searrow \chi\ T'\ \text{-}\ t'}$$

$$\frac{\Gamma \vdash T_1 : \star \searrow T_1' \quad \Gamma \vdash T_1 \ni t_1 \searrow t_1' \quad \Gamma \vdash t_2[t_1/x]^{T_1} \overset{\leftrightarrow}{\in} T_2 \searrow t_2'}{\Gamma \vdash [x : T_1 = t_1]\ \text{-}\ t_2 \overset{\leftrightarrow}{\in} T_2 \searrow t_2'}$$

$*$ where t is not a λ- or Λ-abstraction, intersection or equality introduction, or a δ, ρ, φ, or let expression

Figure 5.23: Elaboration of Cedille terms (congruence rules)

- In the elimination form for equality, $\rho\ t\ @x.\langle t_2\rangle.T_1$ - t_1, we require that the term t_2 given as part of the guide be a well-formed after erasure.
- In the φ axiom, $\varphi\ t_1$ - $t_2\ \{t_3\}$, we require that the erasure of t_3 (to which the entire expression erases) be well-formed.

Remark 5.44. In Cedille, as in CDLE, call-by-name reduction of type constructors does not require reduction of terms. We therefore do not spell out in detail the reduction or convertibility relations for kinds and type constructors in Cedille. For convertibility of Cedille kinds and type constructors, we take the formulation presented in Figure 3.6 and replace the relation $=_{\beta\eta}$ for terms with $\cong$.

We noted in Section 3.2 that type inference in CDLE is undecidable due to the presence of non-normalizing terms, and the same is true for Cedille. Nonetheless, we can at least verify that the inference rules comprising the judgments for kind formation, kinding of type constructors, and typing of terms are subject-directed and mode-correct.

Proposition 5.45 (Elaboration syntax-directedness and mode-correctness)**.** *The inference rules comprising judgments* $\Gamma \vdash \kappa \searrow \kappa'$, $\Gamma \vdash T : \kappa \searrow T'$, $\Gamma \vdash t \in T \searrow t'$, *and* $\Gamma \vdash T \ni t \searrow t'$ *(spanning Figures 5.13, 5.16, 5.17, 5.22, and 5.23) are syntax-directed and mode-correct.*

Proof. By inspection of the rules. Note that for inference rule mediating the exchange (reading bottom-up) from checking to synthesis, we have a syntactic criterion as a side condition (noted by label $\star$) that prevents any overlap between it and other rules for the type checking judgment. □

5.5.1 Static Soundness

Currently, we have an erasure function mapping annotated terms to untyped terms, and judgments for elaborating type-annotated and untyped terms. We are therefore interested in confirming that the erasure function preserves our ability to elaborate a term. This confirmation is given in Theorem 5.46 (see Section F.3.3 for the proof).

Theorem 5.46 (Erasure soundness)**.** *If* $\vdash \Gamma \searrow \Gamma'$*, then*

- *if* $\Gamma \vdash t \in T \searrow t'$ *then* $\Gamma; \emptyset \vdash |t| \searrow |t'|$*; and*
- *if* $\Gamma \vdash T : \star \searrow T'$ *and* $\Gamma \vdash T \ni t \searrow t'$ *then* $\Gamma; \emptyset \vdash |t| \searrow |t'|$.

With this, we obtain as a corollary dynamic (convertibility) soundness for the erasures of annotated terms.

Corollary 5.47 (Dynamic soundness (convertibility, annotated terms))**.** *If* $\vdash \Gamma \searrow \Gamma'$*, and:*

- *we have* t_1*,* t_1'*, and* T_1 *such that either* $\Gamma \vdash t_1 \in T_1 \searrow t_1'$*, or* $\Gamma \vdash T_1 : \star \searrow T_1'$ *and* $\Gamma \vdash T_1 \ni t_1 \searrow t_1'$*; and*
- *we have* t_2*,* t_2'*, and* T_2 *such that either* $\Gamma \vdash t_2 \in T_2 \searrow t_2'$*, or* $\Gamma \vdash T_2 : \star \searrow T_2'$ *and* $\Gamma \vdash T_2 \ni t_2 \searrow t_2'$*; and*
- $|t_1| \cong |t_2|$*;*

then $|t_1'| =_{\beta\eta} |t_2'|$.

Proof. By Theorem 5.46, $\Gamma; \emptyset \vdash |t_1| \searrow |t_1'|$ and $\Gamma; \emptyset \vdash |t_2| \searrow |t_2'|$. By Corollary 5.41, $|t_1'| =_{\beta\eta} |t_2'|$. □

The last property we consider for elaboration is *static soundness*: a term being well-typed in the surface language implies that its elaboration can be classified by the elaboration of its (checked or synthesized) type. Of course, for this to be true we also require that the same holds at the level of type constructors and kinds. This is affirmed by Theorem 5.48 below (see Section F.3.4 for the proof).

Theorem 5.48 (Static soundness)**.** *Assuming* $\vdash \Gamma \searrow \Gamma'$*, then:*

1. *if* $\Gamma \vdash \kappa \searrow \kappa'$ *then* $\Gamma' \vdash \kappa'$*;*
2. *if* $\Gamma \vdash T : \kappa \searrow T'$ *then exists* κ' *where* $\Gamma \vdash \kappa \searrow \kappa'$ *and* $\Gamma' \vdash T' : \kappa'$*;*
3. *if* $\Gamma \vdash t \in T \searrow t'$ *then exists* T' *where* $\Gamma \vdash T : \star \searrow T'$ *and* $\Gamma \vdash t' \in T'$*; and*
4. *if* $\Gamma \vdash T \ni t \searrow t'$ *and* $\Gamma \vdash T : \star \searrow T'$ *then* $\Gamma \vdash T' \ni t'$.

Together, Theorems 5.42 and 5.48 allow us to import CDLE's logical consistency and qualified termination guarantee into Cedille.

Proposition 5.49 (Logical consistency)**.**
There is no term t *such that* $\emptyset \vdash t \in \forall\, X : \star.\, X \searrow t'$ *(for some* t'*).*

Proof. Assume such a t and t' exist. By Theorem 5.48 and inversion, $\emptyset \vdash t' \in \forall\, X : \star.\, X$. This contradicts Theorem 3.18. □

Proposition 5.50 (Termination guarantee)**.** *If $\emptyset \vdash t \in T \searrow t'$ and either*

1. *T is of the form $\Pi\, x : T_1.\, T_2$, or*
2. *T is of the form D where $\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$,*

then $|t|$ is call-by-name normalizing.

Proof. By contradiction. Assume that $|t|$ is non-normalizing under call-by-name reduction. By Theorem 5.48, $\emptyset \vdash t' \in T'$ where $\emptyset \vdash T : \star \searrow T'$, and furthermore by Theorem 5.46 $\emptyset; \emptyset \vdash |t| \searrow |t'|$. By a straightforward coinductive proof that invokes Theorem 5.42, t' is non-normalizing under call-by-name reduction as well.

We now consider the cases.

1. By inversion, $T' = \Pi\, x : T_1'.\, T_2'$ for some T_1', T_2'. So, $\emptyset \vdash t' \in \Pi\, x : T_1'.\, T_2'$, and by Theorem 3.19 $|t'|$ is call-by-name normalizing — a contradiction.
2. By inversion, $T' = \mu(\mathit{isig}(\mathcal{D}))$. So, $\emptyset \vdash t' \in \mu(\mathit{isig}(\mathcal{D}))$. By Fact 5.3, there exists a type inclusion of this type into $T_1' \to T_2'$ for some T_1', T_2'. By Theorem 3.21, t' is call-by-name normalizing — a contradiction.

□

5.6 Implementing the CDLE Interface for Course-Of-Values Induction

This section brings to conclusion the elaboration of inductive datatypes in Cedille to CDLE by providing an implementation of the interface for course-of-values induction described in Section 5.1.3. We begin by summarizing the generic derivation of induction by Firsov et al. [34] for efficient Mendler-style encodings, then motivate the derivation of a form of restricted existential type through a discussion of the tension between the course-of-values iteration scheme and the requirement that datatype signatures be positive. After these prerequisites, we conclude with the derivation of Mendler-style course-of-values induction in CDLE.

5.6.1 Efficient Mendler-Style Encodings

Figure 5.24 summarizes the generic derivation of Firsov et al. [34] using a handful of inference rules and one erasure rule (we will explain why only the erasure of

$$\boxed{\Gamma^- \vdash T^- : \star^+} \quad \boxed{\Gamma^- \vdash t^- \in T^+}$$

$$\frac{\Gamma \vdash F : \star \to \star}{\Gamma \vdash \mu_{\mathrm{D}} F : \star}$$

$$\frac{\Gamma \vdash F : \star \to \star \quad \Gamma \vdash \mathit{Mono} \cdot F \ni t_1 \quad \Gamma \vdash F \cdot \mu_{\mathrm{D}} F \ni t_2}{\Gamma \vdash \mathit{in}_{\mathrm{D}} \cdot F \text{ -}t_1\ t_2 \in \mu_{\mathrm{D}} F}$$

$$\begin{array}{lcl} \mathit{PrfAlg}_{\mathrm{D}} & : & \Pi\, X : \star \to \star.\, \mathit{Mono} \cdot X \to (\mu_{\mathrm{D}} F \to \star) \to \star \\ \mathit{PrfAlg}_{\mathrm{D}} & =_{\mathbf{df}} & \lambda\, X : \star \to \star.\, \lambda\, x : \mathit{Mono} \cdot X.\, \lambda\, Y : \mu_{\mathrm{D}} F \to \star. \\ & & \quad \forall\, R : \star.\, \forall\, x' : \mathit{Cast} \cdot R \cdot \mu_{\mathrm{D}} F.\, (\Pi\, z : R.\, Y\ (\mathit{elimCast}\ \text{-}x'\ z)) \\ & & \quad\quad \to \Pi\, y : F \cdot R.\, Y\ (\mathit{in}_{\mathrm{D}} \cdot X\ \text{-}x\ (\mathit{elimCast}\ \text{-}(x\ \text{-}x')\ y)) \end{array}$$

$$\frac{\Gamma \vdash F : \star \to \star \quad \Gamma \vdash T : \mu_{\mathrm{D}} F \to \star \quad \Gamma \vdash \mathit{Mono} \cdot F \ni t_1 \quad \Gamma \vdash \mathit{PrfAlg}_{\mathrm{D}} \cdot F\ t_1 \cdot T \ni t_2}{\Gamma \vdash \mathit{induction}_{\mathrm{D}} \cdot F \cdot T\ \text{-}t_1\ t_2 \in \Pi\, x : \mu_{\mathrm{D}} F.\, T\ x}$$

$$|\mathit{in}_{\mathrm{D}}| \quad = \quad \lambda\, x.\, \lambda\, y.\, y\ (\lambda\, z.\, z\ y)\ x$$

Figure 5.24: Summary of the derivation by Firsov et al. [34]

$|\mathit{in}_{\mathrm{D}}|$ is shown shortly). To disambiguate for the constructs presented in Section 5.1.3, we label the type least fixedpoint, and its constructor and eliminator, in the present figure with the subscript "D" (after the first initial of the first author of [34]).

The first two inference rules (for type formation and introduction) are straightforward, being essentially a relabeling of the corresponding rules in Figure 5.6. The main difference is in inference rule of the dependent eliminator, $\mathit{induction}_{\mathrm{D}}$: terms playing the role of t_2 in the rule are allowed to assume the existence of a type inclusion $\mathit{Cast} \cdot \mathit{Rp} \cdot \mu_{\mathrm{D}} F$, but do *not* have access to a destructor for terms of type R — the crucial component that enables course-of-values induction.

The final listing of the figure is the untyped lambda calculus term that realizes the constructor in_{D}. This is the term we would expect to obtain if we applied

the recipe described in Section 4.2.2 for obtaining impredicative encodings of types and lambda encodings of their constructors to the typing and computation rule of Mendler-style iteration shown in Section 2.2.3 (this is left to the reader as an exercise). However, we do not show the same for $induction_{\mathrm{D}}$ — though it satisfies the computation law for Mendler-style iteration, the particular lambda term that realizes $induction_{\mathrm{D}}$ in Firsov et al. [34] (which in their paper is called *inductionM*) is rather more complex than what we might expect from following the recipe in Section 4.2.2, and thus for our purposes somewhat unwieldy.

Fortunately (and perhaps surprisingly), this relative shortcoming is easily bypassed. Because CDLE allows us to reflect typing derivations into the type system itself with *View*, we can use $induction_{\mathrm{D}}$ to prove that the untyped lambda term that we *expect* to realize the dependent eliminator for $\mu_{\mathrm{D}}F$ *actually does!*

Proposition 5.51. *There exists a CLDE term ind_D satisfying the following typing rule and erasure.*

$$\frac{\Gamma \vdash F : \star \to \star \quad \Gamma \vdash T : \mu_D F \to \star \quad \Gamma \vdash \mathit{Mono} \cdot F \ni t_1 \quad \Gamma \vdash \mathit{PrfAlg}_D \cdot F\ t_1 \cdot T \ni t_2}{\Gamma \vdash \mathit{ind}_D \cdot F \cdot T\ \text{-}t_1\ t_2 \in \Pi\, x : \mu_D F.\, T\ x}$$

$$|\mathit{ind}_D| = \lambda\, y.\, \lambda\, x.\, x\ y$$

Proof. Given by the CDLE program below. Note that the signature F and its monotonicity witness *mono* are parameters to the definitions below, as well as to the definitions of $PrfAlg_{\mathrm{D}}$, in_{D}, and $induction_{\mathrm{D}}$

```
indView : ∀ P : μ_D F → ⋆. Π a: PrfAlg_D ·P. Π x : μ_D F. View ·(P x) β{x a}
= Λ P. λ a.
  induction_D ·(λ x: μ_D. View ·(P x) β{x a})
   Λ R. Λ c. λ ih. λ xs.
   intrView ·(P (in (elimCast -(mono -c) xs)))
     β{a (λ x. x a) xs}
     -(a ·R -c (λ x. elimView ·(P (elimCast -c x)) β{x a} -(ih x)) xs)
     -β .

ind : ∀ P : μ_D F → ⋆. PrfAlg_D ·P → Π x : μ_D. P x
= Λ P. λ a. λ x. elimView ·(P x) β{x a} -(indView ·P a x) .
```

□

Firsov et al. [34] also provide for their generic encoding an efficient (i.e., constant time) destructor, and characterize it by proving that it satisfies *Lambek's lemma* (Lambek [54]). This destructor is implemented in using $induction_{\mathrm{D}}$; since we shall be using ind_{D} instead, we define our own and similarly prove Lambek's lemma.

Proposition 5.52. *There exists a CDLE term out_D satisfying the following typing rule and erasure.*

$$\frac{\Gamma \vdash F : \star \to \star \quad \Gamma \vdash \mathit{Mono} \cdot F \ni t}{\Gamma \vdash \mathit{out}_D \cdot F \text{ -}t : \mu_D F \to F \cdot \mu_D F}$$

$$|\mathit{out}| = \lambda\, x.\, x\ (\lambda\, y.\, \lambda\, z.\, z)$$

Furthermore, Lambek's lemma holds, i.e.,

1. *For all t, $|\mathit{out}_D\ (\mathit{in}_D\ t)|$ $\text{--}\!\!\to^*$ $|t|$*
2. *For all $t : \mu_D F$, the equality $\{\mathit{in}_D\ (\mathit{out}_D\ t) \simeq t\}$ is provable.*

Proof. For the typing and erasure rule, we exhibit the CDLE term below.

```
outD : μDF → F ·μDF
= indD ·(λ x: μDF. F ·μDF) Λ R. Λ c. λ ih. elimCast -(mono -c) .
```

Note that it is inessential that the erasure is only equal up to $\alpha\beta$-equivalence to that given in the statement of the property, as the φ axiom allows us to assign the type $\mu_{\mathrm{D}} F \to F \cdot \mu_{\mathrm{D}} F$ to $\lambda\, x.\, x\ (\lambda\, y.\, \lambda\, z.\, z)$ directly.

For Lambek's lemma,

1. holds by reduction alone.
2. holds through a proof by induction, shown below.

```
lambek2 ◁ Π x: μDF. { inD (outD x) ≃ x }
= indD ·(λ x: μDF. { inD (outD x) ≃ x })
    Λ R. Λ c. λ ih. λ xs. β .
```

□

5.6.2 Course-Of-Values Iteration and Signature Positivity

Course-of-values iteration is translated into the Mendler style of coding recursion by defining recursive functions with respect to functions of the following type.

$$\forall\, R{:}\star.\, (R \to T) \to (R \to F \cdot R) \to F \cdot R \to T$$

We might, therefore, think that to derive course-of-values induction from ordinary induction, we could augment the datatype signature F with such a destructor. To see what we mean, observe that the type above is isomorphic to the following (assuming we have binary products, which are derivable in CDLE).

$$\forall\, R{:}\star.\, (R \to T) \to (R \to F \cdot R) \times F \cdot R \to T$$

So, we could form a new type constructor $G =_{\mathbf{df}} \lambda\, X{:}\star.\, (X \to F \cdot X) \times F \cdot X$, and the typing law for Mendler-style course-of-values iteration for F reduces to the typing of Mendler-style iteration for G.

Unfortunately, using this observation to attempt to derive course-of-values induction faces two difficulties.

- *G is not positive*, which we require in order to use the framework of Firsov et al. [34] to implement course-of-values induction. While Mendler-style inductive types can be safely formed from mixed-variant signatures, the expected case distinction scheme for such types is not definable in logically consistent type theories (see the discussion in Section 2.1.2.3).
- *This signature defines a* superset *of the intended datatype,* as the first component is not restricted to the canonical datatype destructor and leaves open the

$$\frac{\Gamma \vdash H : \star \to \star \quad \Gamma \vdash F : \star \to \star}{\Gamma \vdash \mathit{RExt} \cdot H \cdot F : \star} \qquad \frac{\Gamma \vdash S : \star \quad \Gamma \vdash s : H \cdot S \quad \Gamma \vdash t : F \cdot S}{\Gamma \vdash \mathit{pack} \cdot S \text{ -}s\ t : \mathit{RExt} \cdot H \cdot F}$$

$$\frac{\Gamma \vdash P : \mathit{RExt} \cdot H \cdot F \to \star \quad \Gamma \vdash t : \forall\, X{:}\star.\, \forall\, x{:}H \cdot X.\, \Pi\, y{:}F \cdot X.\, P\ (\mathit{pack} \cdot X\ \text{-}x\ y)}{\Gamma \vdash \mathit{unpack} \cdot P\ t : \Pi\, z{:}\mathit{RExt} \cdot H \cdot F.\, P\ z}$$

$$\begin{array}{rcl} |\mathit{pack}| & = & \lambda\, x.\, \lambda\, y.\, y\ x \\ |\mathit{unpack}| & = & \lambda\, y.\, \lambda\, x.\, x\ y \end{array}$$

Figure 5.25: Derived implicitly restricted existential types

possibility that predecessors are tupled together with different "destructors." This interferes with characterizing course-of-values induction.

Uustalu and Vene [90] solved the first difficulty. They defined a superset of course-of-values datatypes by forming, with restricted existentials, the covariant signature $G =_{\mathbf{df}} \lambda\, X{:}\star.\, \exists\, R{:}\star.\, (R \to X) \times (X \to F \cdot X) \times F \cdot X$. To solve the second difficulty, we use *View* types, which enable us to precisely specify that the abstract destructor is the canonical one.

Remark 5.53. Credit for the derivation of implicitly restricted existentials in CDLE belongs to Denis Firsov and Larry Diehl (see Firsov et al. [35]).

5.6.3 Implicitly Restricted Existentials

Figure 5.25 axiomatically summarizes the derivation of *RExt*, a family of *implicitly* restricted existential types. The type $\mathit{RExt} \cdot H \cdot F$ existentially quantifies over types S such that both $F \cdot S$ *and* the restriction $H \cdot S$ holds. Differing from the usual presentation of restricted existentials, the evidence for the restriction $H \cdot S$ is erased and cannot be used in computation.

To form this type, type constructors H and F must have kind $\star \to \star$. The

introduction form *pack* takes a type argument S, a term s of type $H \cdot S$ as an erased argument, and a term t of type $F \cdot S$. The (dependent) elimination form *unpack* takes a property P over $\mathit{RExt} \cdot H \cdot F$ and a proof that, for all $X : \star$, P holds of $\mathit{pack} \cdot X$ *-x y*, erased $x : H \cdot S$, and $y : F \cdot S$. The whole expression is a proof that P holds for all $\mathit{RExt} \cdot H \cdot F$. Finally, the erasure rules for implicitly restricted existentials allow us to confirm the expected computation law: for all t_1 and t_2, $|\mathit{unpack}\ t_1\ (\mathit{pack}\ t_2)| \dashrightarrow^* |t_1\ t_2|$.

Proposition 5.54. *There exists CDLE type constructor RExt and terms pack and unpack satisfying the typing and erasure rules of Figure 5.25.*

Proof. The derivation can be found in Section D.1 of the Appendix, and is originally due to Firsov et al. [35]. It follows a recipe similar to that given by Stump [80] and described in Sections 4.2.2 and 4.2.4 of this dissertation. □

5.6.4 Course-Of-Values Induction

To begin our derivation of datatypes with course-of-values induction, notice that there is subtle circularity in proposing to define a signature CV for datatypes by augmenting an elaborated signature F with of the very destructor $\mathit{out} : \mu_{\mathrm{D}} CV \to F \cdot \mu_{\mathrm{D}} CV$, which is typed *in terms of CV itself!* To break this circularity, we start with the following observations.

- CLDE is committed to an *untyped* ontology of terms. It is *a priori* possible to define CV using some term (perhaps untyped) that is equal to the typed definition of *out* we give later.
- Destructor $\mathit{out}_{\mathrm{D}}$, which when instantiated with CV has type $\mu_{\mathrm{D}} CV \to CV \cdot \mu_{\mathrm{D}} CV$, performs no computation specific to the datatype signature — in particular, the monotonicity witness is *erased*. No operations defined specifically

for CV are needed to express the computation of out_{D}.

The CDLE derivation is given in Figure 5.26, with syntax modified to aid readability (e.g., sugar for derived product types $S \times T$ with projections π_1 and π_2, and removal of some annotations). The code is parameterized in a type scheme $F : \star \to \star$ and monotonicity proof $m : \mathit{Mono} \cdot F$. We summarize the key aspects.

Generic Course-of-Values Datatype Signature

The *Top*-typed definition *outU* gives the underlying computational content of the destructor for course-of-values datatypes. Reading it with the intended typing, we have a function of type $\mu_{\mathrm{D}} CV \to F \cdot \mu_{\mathrm{D}} CV$ that acts by destructing its input with out_{D} (giving a result of type $CV \cdot \mu_{\mathrm{D}} CV$) and unpacking the resulting restricted existential to return the predecessor values. Next, we define the type family of restrictions RCV, whose first argument R is the parameter in which we will soon take a fixedpoint (i.e., R is a placeholder for $\mu_{\mathrm{D}} CV$) and whose second argument X is the existentially quantified type variable. The body of RCV is the product of proofs that X is included into R and that *outU* has type $X \to F \cdot X$.

Finally, we form the signature CV and give a proof mCV that it is monotonic. We note the critical point in mCV where the existential is re-packed: there, transitivity of type inclusions is used (*castTrans*, presented in Figure 3.11) to combine the hypothetical inclusion $c : \mathit{Cast} \cdot X \cdot Y$ given as part of any proof of monotonicity (refer to Figure 3.12) with the inclusion $\pi_1\ r : \mathit{Cast} \cdot Z \cdot X$ that is part of the erased evidence r of the restriction, where Z is the existential type variable.

```
outU : Top = β{ λ x. unpack (λ xs. xs) (out_D x) } .

RCV : ⋆ → ⋆ → ⋆
= λ R: ⋆. λ X: ⋆. (Cast ·X ·R) × View ·(X → F ·X) outU .

CV : ⋆ → ⋆ = λ R: ⋆. RExt ·(RCV ·R) ·F .

mCV : Mono ·CV
= Λ X. Λ Y. Λ c.
  intrCast -(unpack Λ Z. Λ r. λ xs.
        pack -((castTrans -(π_1 r) -c), π_2 r) xs)
    -(unpack Λ Z. Λ r. λ xs. β)) .

out : μ_DCV → F ·μ_DCV
= λ x. unpack (Λ X. Λ r. λ xs. elimCast -(mono -(π_1 r)) xs) (out_D -mCV x) .

in : F ·μ_DCV → μ_DCV
= λ xs. in_D -mCV (pack -(castRefl, intrView outU -out -β) xs) .

PrfAlg : (μ_DCV → ⋆) → ⋆
= λ P: μ_DCV → ⋆.
  ∀ R: ⋆. ∀ c: Cast ·R ·μ_DCV. View ·(R → F ·R) outU ⇒
  (Π x: R. P (elimCast -c x)) → Π xs: F ·R. P (in (elimCast -(m -c) xs)) .

ind : ∀ P: μ_DCV → ⋆. PrfAlg ·P → Π x: μ_DCV. P x
= Λ P. λ a.
  ind_D -mCV
    Λ R. Λ c. λ ih.
    unpack Λ X. Λ r. λ xs.
      a ·X -(castTrans -(π_1 r) -c) -(π_2 r)
        (λ x. ih (elimCast -(π_1 r) x)) xs .
```

Figure 5.26: Generic encoding of course-of-values datatypes in CDLE

Datatype Destructor and Constructor

We can now define an informatively typed destructor *out* that is definitionally equal to $outU$. With *out*, we next define the constructor *in* for datatypes with course-of-values induction. Within it, we use in_{D} and are required to give a restricted existential $CV \cdot \mu_{\mathrm{D}}\, CV$. We chose for the existentially quantified type $\mu_{\mathrm{D}}\, CV$, provide a proof that $\mu_{\mathrm{D}}\, CV$ is included in itself (*castRefl*), and a proof that $outU$ can be viewed at type $\mu_{\mathrm{D}}\, CV \to F \cdot \mu_{\mathrm{D}}\, CV$, given with $intrView$. These two proofs are tupled together as erased evidence of the restriction, and packed together with $xs : F \cdot \mu_{\mathrm{D}}\, CV$.

Course-of-Values Induction

The generic course-of-values induction principle, *ind*, is defined as a composition of ind_{D} (see Proposition 5.51) and the eliminator for restricted existentials (to understand the type signature of *ind*, see the discussion in Section 3.4 concerning the CDLE-idiomatic course-of-values induction principle). After applying both and introducing assumptions, the goal is to show P holds for in_{D} (*pack xs*) (convertible with *in xs*; erased arguments are here omitted), accomplished by giving parameter a:

- a proof of inclusion of the existentially quantified X into $\mu_{\mathrm{D}}\, CV$, using transitivity on the inclusion of X into R and of R into $\mu_{\mathrm{D}}\, CV$;
- a proof that $outU$ has type $X \to F \cdot X$.
- and an induction hypothesis formed by restricting the domain of *ih* to X.

5.6.4.1 Characterization

Before implementing Cedille's surface language features, we take a moment to answer two questions of semantics. First, since our derivation of course-of-values

$$\frac{\Gamma \vdash t_2 : \mu_{\mathrm{D}} CV \qquad \Gamma \vdash t_1 : \forall\, R{:}\star.\, (R \to T) \to (R \to \mu_{\mathrm{D}} CV) \to (R \to F \cdot R) \to F \cdot R \to T}{\Gamma \vdash \mathit{mcovr}\ t_1\ t_2 : T}$$

$$\mathit{mcovr}\ t_1\ (\mathit{in}\ t_2')\ \dashrightarrow\ t_1 \cdot \mu F\ (\mathit{mcovr}\ t_1)\ (\lambda\, x.\, x)\ \mathit{out}\ t_2'$$

$$\frac{\Gamma \vdash t_1 : \forall\, R{:}\star.\, (R \to T) \to (R \to \mu_{\mathrm{D}} CV) \to (R \to F \cdot R) \to F \cdot R \to T \quad \Gamma \vdash t_2 : \mu_{\mathrm{D}} CV \quad \Gamma \vdash h : \mu_{\mathrm{D}} CV \to T \quad \Gamma, x{:}F \cdot \mu_{\mathrm{D}} CV \vdash h\ (\mathit{in}\ x) \simeq t_1 \cdot \mu_{\mathrm{D}} CV\ h\ (\lambda\, x.\, x)\ \mathit{out}\ x}{\Gamma \vdash h\ t_2 \simeq \mathit{mcovr}\ t_1\ t_2}$$

Figure 5.27: Course-of-values recursion: typing, computation, extension

induction principle computes the way Mendler-style *iteration* does, what is the connection between *ind* and Mendler-style course-of-values recursion as we discussed in Section 2.2.6? Second, although we know that Lambek's lemma holds for the generic framework of Firsov et al. [34], our derivation of course-of-values induction is not a direct instantiation of that framework, so we would like to confirm that it holds for the given definitions of *in* and *out*.

To answer the first of these, we start by noticing is immediate that *ind* is an implementation of the CDLE-idiomatic course-of-values induction combinator that we described in Section 3.4.

Proposition 5.55. *The combinator for course-of-values induction described in Figure 3.13 is implemented by the types and terms of Figure 5.26 by taking:*

1. *the least fixedpoint μF of a given signature $F : \star \to \star$ as $\mu_D CV$ (where F is a parameter in the definition of CV),*
2. *the generic constructor as in, and*
3. *the combinator mcovi as ind.*

Proof. Satisfaction of the typing rule is immediate. For the computation rule, it follows from the erasures of $\mathit{ind}_{\mathrm{D}}$, in_{D}, *pack*, and *unpack*, and by β-reduction. □

We can now characterize the CDLE combinator for course-of-values induction by showing that it is strong enough to implement the typing, computation, and extensionality law of Mendler-style course-of-values recursion, with some qualifications. For reference, we repeat these laws here in Figure 5.27.

- We expect the computation law to hold **not** by a single step, but in a constant number of β-reduction steps.
- We will interpret the informal $\simeq$ equality relation used to express the extensionality law with CDLE's primitive equality type. However, for the statement that results from this to be provable, we (at least appear to) require an additional assumption on terms playing the role of t_1, which is in essence that they respect extensional functional equality in their first argument. This assumption is intimately connected with the categorical semantics of Mendler algebras as *natural transformations*.

Definition 5.56 (*MCoVRAlgExt*)**.** We introduce the following CDLE terms for use in the proof of Theorem 5.57 below.

```
MCoVRAlg : ⋆ → ⋆
= λ X: ⋆.
  ∀ R: ⋆. (R → X) → (R → μD CV) → (R → F ·R) → F ·R → X .

MCoVRAlgExt : Π X: ⋆. MCoVRAlg ·μD CV ·X → ⋆
= λ X: ⋆. λ a: MCoVRAlg ·D ·X.
  ∀ R1: ⋆. ∀ f1: R1 → X.
  ∀ R2: ⋆. ∀ f2: R2 → X.
  (Π x1: R1. Π x2: R2. { x1 ≃ x2 } ⇒ { f1 x1 ≃ f2 x2 })
  → Π xs1: F ·R1. Π xs2: F ·R2.
  { xs1 ≃ xs2 } ⇒ { a f1 (λ x. x) out xs1 ≃ a f2 (λ x. x) out xs2 } .
```

Type constructor *MCoVRAlg* gives us a name for the family of types that terms playing the role of t_1 have in Figure 5.27. Type constructor *MCoVRAlgExt* is a predicate over such terms that states that given two functions $f_1 : R_1 \to X$ and $f_2 : R_1 \to X$, if they map equal inputs to equal results, then functions $t_1\ f_1\ (\lambda\, x.\, x)\ \mathit{out}$ and $t_1\ f_2\ (\lambda\, x.\, x)\ \mathit{out}$ do as well.

```
mcovr : ∀ X: ⋆. MCoVRAlg ·X → μ_D CV → X
= Λ X. λ a.
  ind ·(λ x: μ_D CV. X)
    Λ R. Λ c. Λ o. λ ih. λ xs.
    a ·R ih (elimCast -c) (elimView outU -o) xs .

mcovrBeta
: ∀ X: ⋆. ∀ a: MCoVRAlg ·X.
  ∀ R: ⋆. ∀ xs: F ·R. { mcovr a (in xs) ≃ a (mcovr a) (λ x. x) out xs }
= Λ X. Λ a. Λ R. Λ xs. β .

mcovEta
: ∀ X: ⋆. ∀ a: MCoVRAlg ·X. MCoVRAlgExt ·X a
  → ∀ h: μ_D CV → X. (∀ R: ⋆. ∀ xs: F ·R. { h (in xs) ≃ a h (λ x. x) out xs })
  → Π x: μ_D CV. { h x ≃ mcovr a x } .
= Λ X. Λ a. λ ext. Λ h. λ hom.
  ind ·(λ x: μ_D CV. { h x ≃ mcovr a x })
    Λ R. Λ c. Λ o. λ ih. λ xs.
    ρ (hom ·R -xs) @x.{ x ≃ a (mcovr a) (λ x. x) out xs }
  - ext ·R -(λ x. h (cast -c x)) ·R -(λ x. mcovr a (elimCast -c x))
      (λ x1. λ x2. Λ eq.
         ρ eq @x. { h x ≃ mcovr a x2 }
       - ih x2)
      xs xs -β .
```

Figure 5.28: Proof of Theorem 5.57

Theorem 5.57. *There exists CDLE types and terms which satisfy the laws of Figure 5.27 for course-of-values recursion for all $T : \star$, $t_2 : \mu_D CV$, and $t_1 : MCoVRAlg{\cdot}T$ such that $MCoVRAlgExt{\cdot}T\ t_1$ holds, with the computation law holding up to a constant number of reduction steps and the extensionality law holding via CDLE's primitive equality type.*

Proof. We exhibit CDLE terms listed in Figure 5.28 as proof. □

For the second question regarding the status of Lambek's lemma, the proof is shorter.

Proposition 5.58 (Lambek's lemma)**.**

1. *For all t, $|out\ (in\ t)|$ $\dashrightarrow^*$ $|t|$.*
2. *For all $t : \mu_D CV$, the equality $\{in\ (out\ t) \simeq t\}$ is provable.*

Proof.

1. holds by β-reduction alone.
2. is proven by the following CDLE term.

```
lambek2 : Π x: μ_D CV. { in (out x) ≃ x }
= ind ·(λ x: μ_D CV. { in (out x) ≃ x })
    Λ R. Λ c. Λ o. λ ih. λ xs. β .
```

□

5.6.5 Implementing μ- and σ-expressions

We end this section by exhibiting the CDLE definitions used to implement the interface listed back in Figure 5.6. This is shown in Figure 5.29, which we now describe. This CDLE code listing has as parameters type constructor $F : \star \to \star$ and monotonicity proof $mono : Mono \cdot F$.

The type least fixedpoint μF is implemented using the fixedpoint μ_{D} of Firsov et al. 5.24 and the signature CV defined in Section 5.6.4 (recall that F is a parameter to the definition of CV). The constructor for μF is simply *in* from Figure 5.26.

For the datatype globals, type constructor IsD is defined as $RCV \cdot \mu F$, that is, as a family of binary pair types: for each type T, $IsD \cdot T$ is the type of pairs whose first component is a proof of inclusion of T into μF and whose second component is a proof that the destructor *out* can be assigned the type $T \to F \cdot T$. The witness *isD* that this pair is inhabited for μF is straightforward to exhibit, as type inclusions are reflexive and *out* already has type $\mu F \to F \cdot \mu F$. For the proof of type inclusion *toD*, evidence of $IsD \cdot T$ for some $T : \star$ already entails evidence of a type inclusion $Cast \cdot T \cdot \mu F$, so the only "trick" is exhibiting this inclusion from *erased* evidence of its existence. Since the introduction and elimination form for *Cast* only take erased

```
μF : ⋆ = μ_D CV .

IsD : ⋆ → ⋆ = RCV ·μF .

isD : IsD ·μF = (castRefl ·μF , intrView outU -out -β) .

toD : ∀ X: ⋆. IsD ·X ⇒ Cast ·X ·μF
= Λ X. Λ i. intrCast -(elimCast -(π_1 i)) -(λ x. β) .

toFD : ∀ X: ⋆. IsD ·X ⇒ Cast ·(F ·X) ·(F ·μF)
= Λ X. Λ i. intrCast -(elimCast -(mono -(toD -i))) -(λ x. β) .

CTree : Π R: ⋆. IsD ·R → (μF → ⋆) → ⋆
= λ R: ⋆. λ i: IsD ·R. λ P: μF → ⋆.
  Π xs: F ·R. P (in (elimCast -(toFD -i) xs)) .

sigmaFam
: ∀ R: ⋆. ∀ i: IsD ·R. Π x: R. ∀ P: μF → ⋆.
  CTree ·R i ·P → P (elimCast -(toD -i) x)
= Λ R. Λ i. λ x. Λ P. λ ct.
  ind ·(λ x: μF. View ·R β{x} ⇒ P x)
    (Λ R'. Λ c. Λ o. λ ih. λ xs. Λ v.
     - ct (elimView outU -(π_2 i) (elimView β{in xs} -v)))
    (elimCast -(toD -i) x) -(intrView β{x} -x -β) .

muFam
: Π x: μF. ∀ P: μF → ⋆.
  (∀ R: ⋆. ∀ i: IsD ·R.
   (Π y: R. P (elimCast -(toD -i) y)) → CTree ·R i ·P)
  → P x
= λ x. Λ P. λ a.
  ind ·P (Λ R. Λ c. Λ o. λ ih. a ·R -(c , o) ih) x .

sigmaEq : ∀ t1: Top. ∀ t2: Top. { sigmaFam t1 t2 ≃ t1 (λ ih. λ xs. xs t2) }
= Λ t1. Λ t2. β .

muEq : ∀ t1: Top. ∀ t2: Top. { muFam t1 t2 ≃ t1 (λ ih. λ xs. xs (t2 ih)) }
= Λ t1. Λ t2. β .
```

Figure 5.29: Implementation of Figure 5.6

arguments, this poses no difficulty.

Finally, we reach the implementation of *sigmaFam* and *muFam*, the term definitions that implement the families *sigma* and *mu*. To simplify the type signatures of both, we make an auxiliary definition *CTree* for the family of types for the elaborations of case trees. *CTree* is parameterized by a type R for the type of recursive subdata in constructor arguments, a witness $IsD \cdot R$, and predicate $P : \mu F \to \star$ into which the case tree is eliminating.

Definition *sigmaFam* (whose type signature we explained in Section 5.1.3) is implemented using the induction principle *ind*, though note that it does not use the induction hypothesis. We eliminate into an implicit product $View \cdot R\ \beta\{x\} \Rightarrow P\ x$ so that we can make use of the knowledge that the scrutinee has type R (to invoke *ind* in the first place, we must retype the scrutinee to type μF). In the proof algebra given to *ind*, we use the typing information for *out* (given by $\pi_2\ i : View \cdot (R \to F \cdot R)\ outU$) and the scrutinee (given by $v : View \cdot R\ \beta\{in\ xs\}$) to obtain a term of type $F \cdot R$ that is definitionally equal to xs, and this is what we give to the case tree ct.

The definition of *muFam* is much simpler, being an almost direct use of the induction principle *ind*. The only difference is that in the argument to *ind*, we uncurry the erased arguments $c : Cast \cdot R \cdot \mu F$ and $o : View \cdot (R \to F \cdot R)\ outU$, forming an erased pair.

The last two definitions of the figure, *sigmaEq* and *muEq*, show the untyped lambda expressions that realize, for all terms t_1 and t_2, each instantiation of the *mu* and *sigma* families. With these proofs (which hold by reduction alone), we make

the following definitions of *mu* and *sigma* (note that, as with the code listing, the signature and monotonicity witness are parameters to these definitions).

Definition 5.59 (*sigma*)**.**

- For all $\vdash \Gamma$, t_1 and t_2 with $FV(t_1\ t_2) \subseteq DV(\Gamma)$, define

$$|sigma\ t_1\ t_2| =_{\mathbf{df}} |t_1|\ (\lambda\, x.\, \lambda\, xs.\, xs\ |t_2|)$$

  We implicitly choose variables x, xs fresh with respect to Γ.

- For all $\vdash \Gamma$, $\Gamma \vdash S : \star$, $\Gamma \vdash s \in IsD \cdot F \cdot S$, $\Gamma \vdash S \ni t$, $\Gamma \vdash T : \kappa$ with $\kappa \cong \mu F \to \star$, and $\Gamma \vdash CTree \cdot S\ s \cdot T \ni t_2$, define

$$sigma \cdot F\ \text{-}m \cdot S\ \text{-}s\ t_1 \cdot T\ t_2 =_{\mathbf{df}} \varphi\ \beta\ \text{-}\ (sigmaFam \cdot F\ \text{-}m \cdot S\ \text{-}s\ t_1 \cdot T\ t_2)\ \{|sigma\ t_1\ t_2|\}$$

Definition 5.60 (*mu*)**.**

- For all $\vdash \Gamma$, $x \notin DV(\Gamma)$, t_1 and t_2 with $FV(t_1) \subseteq DV(\Gamma)$ and $FV(t_2) \subseteq DV(\Gamma) \cup \{x\}$, define

$$|mu\ t_1\ \lambda\, x.\, t_2| =_{\mathbf{df}} |t_1|\ (\lambda\, x.\, \lambda\, xs.\, xs\ |t_2|)$$

  We implicitly chose variable xs fresh with respect to Γ.

- For all $\vdash \Gamma$, $\Gamma \vdash t_1 \overset{\dashrightarrow^*}{\in} \mu F$, $\Gamma \vdash T : \kappa$ with $\kappa \cong \mu F \to \star$, variables R, x, and y with $\{R, x, y\} \cap DV(\Gamma) = \emptyset$, and $\Gamma, (R : \star)(x : IsD \cdot F \cdot R)(y : \Pi\, z : R.\, T\ (elimCast\ \text{-}(toD \cdot F \cdot R\ \text{-}x)\ z)) \ni t_2$, define

$$\begin{aligned} mu \cdot F\ \text{-}m\ t_1\ \Lambda\, R.\, \Lambda\, x.\, \lambda\, y.\, t_2 =_{\mathbf{df}}\ & \varphi\ \beta \\ & \text{-}\ (muFam\ t_1 \cdot T\ \Lambda\, R.\, \Lambda\, x.\, \lambda\, y.\, t_3) \\ & \{|mu\ t_1\ \Lambda\, R.\, \Lambda\, x.\, \lambda\, y.\, t_3|\} \end{aligned}$$

Proposition 5.61. *There exists CDLE type constructors, terms, and term families that implement the interface listed in Figure 5.6.*

Proof. It is clear that classification rules are satisfied. For the computation rules, consider the following reduction sequences.

- For all t_1 and t_2,

$$\begin{aligned} & |sigma\ (in\ t_1)\ t_2| \\ = \ & |in|\ |t_1|\ (\lambda\, x.\, \lambda\, xs.\, xs\ t_2) \\ = \ & (\lambda\, xs.\, \lambda\, a.\, a\ (\lambda\, y.\, y\ a)\ (|pack|\ xs))\ |t_1|\ (\lambda\, x.\, \lambda\, xs.\, xs\ |t_2|) \\ \dashrightarrow^* \ & (\lambda\, x.\, \lambda\, xs.\, xs\ |t_2|)\ (\lambda\, y.\, y\ (\lambda\, x.\, \lambda\, xs.\, xs\ t_2))\ (|pack|\ |t_1|) \\ \dashrightarrow^* \ & |pack|\ |t_1|\ |t_2| \\ = \ & (\lambda\, x.\, \lambda\, f.\, f\ x)\ |t_1|\ |t_2| \\ \dashrightarrow^* \ & |t_2|\ |t_1| \end{aligned}$$

- For all t_1, y, and t_2,

$$\begin{array}{rl}
 & |mu\ (in\ t_1)\ \lambda\, y.\, t_2| \\
= & |in|\ |t_1|\ (\lambda\, y.\, \lambda\, xs.\, xs\ |t_2|) \\
= & (\lambda\, xs.\, \lambda\, a.\, a\ (\lambda\, y.\, y\ a)\ (|pack|\ xs))\ |t_1|\ (\lambda\, y.\, \lambda\, xs.\, xs\ |t_2|) \\
\dashrightarrow^* & (\lambda\, y.\, \lambda\, xs.\, xs\ |t_2|)\ (\lambda\, x.\, x\ (\lambda\, y.\, \lambda\, xs.\, xs\ |t_2|))\ (|pack|\ |t_1|) \\
\dashrightarrow & \lambda\, xs.\, xs\ |t_2[(\lambda\, x.\, x\ (\lambda\, y.\, \lambda\, xs.\, xs\ |t_2|))/y]|\ (|pack|\ |t_1|) \\
\dashrightarrow & |pack|\ |t_1|\ |t_2[(\lambda\, x.\, x\ (\lambda\, y.\, \lambda\, xs.\, xs\ |t_2|))/y]| \\
\dashrightarrow^* & |t_2[(\lambda\, x.\, x\ (\lambda\, y.\, \lambda\, xs.\, xs\ |t_2|))/y]|\ t_1 \\
= & |t_2[(\lambda\, x.\, mu\ x\ \lambda\, y.\, t_2)/y]|\ |t_1|
\end{array}$$

□

5.7 Chapter Conclusions

In this chapter, we have formally described the surface language features of Cedille that enable *course-of-values induction*, an expressive definition form for recursive functions and inductive proofs, and shown that these may be elaborated to CDLE in a way that preserves the static and dynamic semantics of the source language. Together, the results in this chapter and in Chapter 4 show that Cedille's datatype system is sound with respect to CDLE, allowing us to import important meta-theoretic properties of the latter — namely, logical consistency and a qualified termination guarantee — into the former. This accomplishment is a major step towards realizing Stump's vision, as expressed in the chapter epigraph, of a practical interactive theorem prover based on a pure typed lambda calculus.

Of course, this vision is not yet achieved, and in Chapter 6 we give a broad outline of promising future directions for this work in the long term. For the remainder of the present chapter, we focus on related work in type based termination checking and future work (in the short term) on improving Cedille's existing features.

5.7.1 Related Work

Sized types

Abel [1] showed how to extend Agda's type theory with *sized types*, allowing datatypes to be annotated with size information and the type system guaranteeing that recursive calls are made on arguments of decreasing size. At the time of writing, using sized types in Agda requires that the programmer define alternative, size-indexed versions of datatypes, and also requires an extension of the underlying theory.

This is in contrast to Cedille, where every datatype declaration is defined with the usual notation and automatically supports course-of-values induction. Furthermore, it requires no extension to the underlying theory: the Mendler style of coding recursion, on which Cedille's termination checking is based, can be expressed using polymorphic typing alone, making it suitable for a paradigm of *strong functional programming* [88] (c.f. [84] for a demonstration of this Cedille), and the generalization to dependent types is neatly dealt (particularly in the case of course-of-values induction) by using CDLE's ability to internalize typing.

On the other hand, *prima facie* sized types allow for even more powerful forms of recursive definitions. In particular, indices may express *relationships* between the sizes of different datatypes, such as in a length function returns sized naturals no larger than the given sized list. It is not clear how this could be implemented using the Mendler style of coding recursion, even in combination with CDLE's facilities for internalizing typing.

Chan [19] demonstrated that a sized dependent type system with higher-rank and bounded size polymorphism — both of which are implemented in Agda — can be soundly translated to the *Extensional Calculus of Inductive Constructions* (ECIC). This is a substantial step towards the design of an ITP with type-based termination checking that is derived from a simpler kernel theory, a goal shared by this dissertation. However, due to its incompatibility with infinite sizes, it does not give a suitable semantics for sized types as they are implemented in Agda.

Coding recursion *á la* Mendler

The Nax language, described by Ahn [4], also takes an approach to termination based on the Mendler style. In Nax, recursive functions are defined in terms of Mendler style recursion schemes, including course-of-values recursion, whereas in Cedille the μ-operator supports course-of-values *induction*. On the other hand, Nax soundly permits datatype definitions with *negative* recursive occurrences, possible because mixed-variant datatypes in Nax lack the standard destructor. We opt for the more traditional approach of restricting declarations to positive datatypes, guaranteeing the existence of datatype destructors.

5.7.2 Future Work

Repairing Progress

In Section 5.4, we saw that progress fails to hold for the Cedille surface language, and listed two alternatives for its repair: abandoning an independent operational semantics for the surface language, or introducing new reduction rules for the stuck terms. The first approach would see definitional equality of surface language

terms defined as equality of their elaborations. This is, of course, undesirable from the perspective of the user, as debugging a type error due to a failure of conversion checking would require knowledge of the elaboration rules.

We find the second approach more promising, though admittedly still rather unorthodox — it amounts to a depth-one elaboration of the stuck surface language construct. For example, the "nonsense" Cedille term below,

$$\sigma\ (\lambda\, x.\, x)\ \{\ |\ \mathit{zero} \to \mathit{zero}\ |\ \mathit{succ}\ n \to n\}$$

can soundly (with respect to convertibility) reduce to the following.

$$(\lambda\, x.\, x)\ (\lambda\, i.\, \lambda\, xs.\, xs\ (\lambda\, z.\, z\ \mathit{zero}\ (\lambda\, n.\, n)))$$

This has the advantage of exposing the underlying lambda encodings *only* when there is no other way to make sense of the surface language term.

Type coercions and subtyping

Another usability concern of the surface language is the proliferation of explicit type coercions in the case branches of μ and σ expressions, for example in the definition of *div* in Figure 5.2. It would be desirable for the type system to infer where such type coercions are necessary, rather than requiring the user to insert them explicitly. Fortunately, there is already machinery that is suitable to put to use for this task: the subtyping judgment (discussed in Section 4.2). Indeed, already we automatically infer the necessary type coercions for constructor arguments in the *expected type* of case branches (see Figure 5.15). Coercive subtyping *á la* Luo [57] offers a promising guide for this direction of work, with the benefit that *coherence* (the property that

all coercions between the same two types are equal) comes for free, as we only use retyping functions in the subtyping judgment.

CHAPTER 6
CONCLUSION

> There is no real ending. It's just the place where you stop the story.
>
> Frank Herbert

What this dissertation aims towards, ultimately, is a practical and feature-rich interactive theorem prover with CDLE as its tiny trusted computing base. In Chapter 1, we saw why this goal is worthwhile: while ITPs are tools for building trust in software, providing one of the most robust methods for gaining trust in the form of machine-checked verification, as software artifacts they are complex systems, and the features they include to facilitate verification may at the same time, when not carefully considered, undermine the very basis for this trust. Through a verified, compositional translation to a smaller kernel theory — a process called "elaboration" — we may regain trust in the proofs carried out in ITPs making use such features. Cedille, the surface language which this dissertation describes and examines, has had its datatype system designed from the start with elaboration in mind.

The collection of features comprising the scope of this dissertation revolve around *inductive definitions*, a feature with near-ubiquity in ITPs and functional languages alike. With nearly the same ubiquity, this feature is treated as a primitive part of the theory, and such treatment is at odds with establishing trust through elaboration. In particular, we can identify two aspects of inductive types that contribute to complexity and which axiomatic treatment of datatypes make irreducible: positivity checking for declarations and termination checking for recursive functions

and inductive proofs.

As a kernel theory, CDLE is particularly well-suited as a target for elaborating inductive definitions (indeed, this was one of its chief design goals). In Chapter 2 we informally overviewed some surface language features for inductive definitions that CDLE makes possible: not only inductive datatypes, but an expressive induction scheme called *course-of-values induction* with type-based termination checking based on Mendler-style recursion schemes and *course-of-values pattern matching*, a novel surface language feature that leverages CDLE's ability to internalize the typing judgment.

The bulk of this dissertation's contributions, presented in Chapters 4 and 5, show the process by which we can elaborate positivity checking, datatype signatures, termination checking, and course-of-values pattern matching to CDLE. The culmination of this are the proofs that this translation preserves the static and dynamic semantics of the source language, allowing us to prove logical consistency and a qualified termination guarantee for the surface language by appealing to the corresponding results established in CDLE's meta-theory.

Though this dissertation draws towards an end, by no means is it the end of the story for the project of justifying existing features of, and developing novel ones for, ITPs. In what remains, we look towards the future directions for the surface language that this work can be used to realize by surveying some existing CDLE developments.

Beyond course-of-values induction

Chapter 2 presented a lengthy treatment of four recursion schemes: iteration, primitive recursion, course-of-values iteration, and course-of-values recursion. This barely covers the "zoo" of recursion schemes that have been studied in the wider literature (see Yang and Wu [96] for an approachable discussion of the subject). Even considering research on deriving expressive recursion and induction schemes in CDLE only, the current state of the art is beyond the surface language we have studied here: Stump et al. [84] described an extension that captures a form of nested recursion, and demonstrated its expressive capability by using it to solve a well-studied challenge problem in the literature on termination checking. Abreu et al. [3] extend this result further with a type-based approach to checking termination of divide-and-conquer algorithms such as *mergesort*; though this work is presented using the Coq proof assistant, it was proceeded by unpublished work, carried out within CDLE, that derived an impredicative encoding that supports this recursion scheme and corresponding induction principle.

Coinductive definitions

Modern ITPs usually support definitional forms other than inductive ones. We mentioned one such form already in Chapter 1: *coinductive definitions*. Though in some settings (such as Haskell) the distinction between the two collapses, roughly speaking, in most settings where nontermination is (at least) restricted coinductive datatypes are dual to inductive datatypes in the following sense. Where inductive datatypes describe necessarily (trans)finite values that are eliminated by terminat-

ing recursion schemes, coinductive datatypes describe possibly infinite values that are *generated* by productive corecursion schemes ("productive" meaning here that all finite observations are well-defined). Jenkins et al. [49] proposed an efficient Mendler-style encoding of coinductive types in CDLE supporting course-of-values corecursion, and gave a sketch of a surface language where programming and proving with coinductive types could be carried out with type-based productivity checking.

Zero-cost data, program, and proof reuse

A novel application of CDLE's extrinsic approach to typing lies in the observation that distinct datatype declarations may denote, through elaboration and erasure, the same set of untyped lambda calculus terms. A thorough examination of this was first given by Diehl et al. [31], who presented several generic combinators for different kinds of reuse for data and the programs and proofs defined for them. A key advantage of this work is that reuse is achieved through retyping functions (called "Curry-style identity functions" in their work), meaning that the type coercions have no significant run-time cost.

Concerning surface language support for this feature, one modest step would be to extend definitional equality so that constructors of different datatypes are considered equal under conditions sufficient to ensure that their elaborated lambda expressions are equal. This would allow users to derive reuse *manually* for datatypes, programs, and proofs. More ambitiously, a higher level syntax for reuse (such as *ornaments* [61, 29]) could allow programmers to define, e.g., length-indexed lists in terms of ordinary lists by describing the function or relation on terms of the latter to

the indices of the former. Such definitions could then be elaborated to CDLE using the generic zero-cost reuse combinators of Diehl et al [31].

Another form of reuse possible in CDLE, and one that builds upon the work of Diehl et al. [31], is constructor subtyping [10], a form of subtyping between datatypes in which datatype D_1 is considered a subtype of D_2 when every constructor of the signature of D_1 is a constructor of the signature for D_2. Marmaduke et al. [59] describe a method of impredicative encoding for datatype signatures that enjoys flexibility in the allowed ordering of constructor lists, and prove generically a sufficient condition for guaranteeing the subtyping relation holds for datatypes formed using these signatures.

Large eliminations and generic programming

From the point of view of functional programming, one of the major appeals of dependent type theories is in their ability to express datatype- and arity-generic programming [94]. Unlike parametric polymorphism, in which functions act uniformly on values of different types, in these forms of generic programming functions can inspect the types of their inputs and vary their behavior accordingly. In dependent type theories, this is accomplished through the use of type-level computations called *large eliminations*, as they are eliminations of inductive types into the (large) universe of types. Some datatype expression, carried at runtime, acts as a "code" for the type, and inspecting this code by dependent case distinction reveals more information about the type. A standard example (in Agda-style pseudocode) of a large elimination over Booleans is given below with *isTrue*.

```
isTrue : Bool → ⋆
isTrue true  = Unit
isTrue false = ∀ X: ⋆. X
```

Large eliminations are typically closely tied to a theory's treatment of its primitive inductive types, and are thus not expected within polymorphic pure typed lambda calculi, in which inductive types are derived via impredicative encodings. However, Jenkins et al. [44] showed that such large eliminations can be *simulated* in CDLE using a type equivalence defined as two-way type inclusion. While they note that the simulation is incomplete in some respects, they demonstrate through the use of extended case studies that it is capable of expressing a wide range of arity- and datatype-generic programming

APPENDIX A
POSITIVITY CHECKER PROOFS

A.1 Soundness

Proof (of Theorem 4.3). By mutual induction on the assumed subtyping derivations.

Case:

$$\frac{S \dashrightarrow^* S' \not\dashrightarrow \quad T \dashrightarrow^* T' \not\dashrightarrow \quad \Gamma \vdash S' \leqslant_s^{\mathbf{n}} T' \rightsquigarrow t}{\Gamma \vdash S \leqslant_s T \rightsquigarrow t}$$

By Proposition 3.25, $\Gamma \vdash S' : \star$ and $\Gamma \vdash T' : \star$. By the IH, $\Gamma \vdash S' \to T' \ni t$ and $|t| =_{\beta\eta} \lambda\, x.\, x$. By Lemma 16 of [83], $\Gamma \vdash S \to T \ni t$, as desired.

Case:

$$\frac{}{\Gamma \vdash X \leqslant_s^{\mathbf{n}} X \rightsquigarrow \lambda\, x.\, x}$$

Immediate.

Case:

$$\frac{R_1 \neq R_2 \quad \Gamma \vdash s \in R_1 \to R_2}{\Gamma \vdash R_1 \leqslant_s^{\mathbf{n}} R_2 \rightsquigarrow s}$$

Immediate from the second premise of the rule and the assumption $|s| =_{\beta\eta} \lambda\, x.\, x$.

Case:

$$\frac{S\ t \cong T\ t'}{\Gamma \vdash S\ t \leqslant_s^{\mathbf{n}} T\ t' \rightsquigarrow \lambda\, x.\, x}$$

Since $|\lambda\, x.\, x| = \lambda\, x.\, x$, the following typing derivation concludes the proof.

$$\frac{\dfrac{\Gamma, x : S\ t \vdash x \in S\ t \quad S\ t \cong T\ t'}{\Gamma, x : S\ t \vdash T\ t' \ni x}}{\Gamma \vdash S\ t \to T\ t' \ni \lambda\, x.\, x}$$

(the case for type applications is similar, so it is omitted.)

Case:

$$\frac{\kappa_1 \cong \kappa_2 \quad \Gamma, X : \kappa_1 \vdash T_1 \leqslant_s T_2 \rightsquigarrow s_2 \quad s_2' =_{\mathbf{df}} \chi\ (T_1 \to T_2)\ \text{-}\ s_2}{\Gamma \vdash \forall\, X : \kappa_1.\, T_1 \leqslant_s^{\mathbf{n}} \forall\, X : \kappa_2.\, T_2 \rightsquigarrow \lambda\, f.\, \Lambda\, X.\, s_2'\ (f \cdot X)}$$

By inversion of the typing derivations for the subjects of subtyping,

$$\frac{\Gamma \vdash \kappa_i \quad \Gamma, X : \kappa_i \vdash T_i : \star}{\Gamma \vdash \forall\, X : \kappa_i.\, T_i : \star}$$

for $i \in \{1, 2\}$. With weakening and exchange, we apply Lemma 20 of [83], obtaining $\Gamma, X : \kappa_2 \vdash T_1[S_1 \cdot X/X] : \star$. By the IH, $\Gamma, X : \kappa_2 \vdash T_1[S_1 \cdot X/X] \to T_2 \ni s_2$ with $|s_2| =_{\beta\eta} \lambda\, x.\, x$.

For the erasure,

$$\begin{array}{lcll} |\lambda\, f.\, \Lambda\, X.\, s_2'\ (f \cdot X)| & = & \lambda\, f.\, |s_2|\ f & \text{erasure} \\ & =_{\beta\eta} & \lambda\, f.\, (\lambda\, x.\, x)\ f & \text{IH} \\ & =_{\beta\eta} & \lambda\, f.\, f & \beta\text{-reduction} \end{array}$$

For the typing, let $\Gamma' =_{\mathbf{df}} \Gamma, f : \forall\, X : \kappa_1.\, T_1, X : \kappa_2$, and conclude with the derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{D}_1 :: \Gamma' \vdash s_2' \in T_1 \to T_2 \quad \mathcal{D}_2 :: \Gamma' \vdash T_1 \ni f \cdot X}{\Gamma' \vdash s_2'\ (f \cdot X) \in T_2} \quad T_2 \cong T_2}{\Gamma' \vdash T_2 \ni s_2'\ (f \cdot X)}}{\Gamma, f : \forall\, X : \kappa_1.\, T_1 \vdash \forall\, X : \kappa_2.\, T_2 \ni \Lambda\, X.\, s_2'\ (f \cdot X)}}{\Gamma \vdash (\forall\, X : \kappa_1.\, T_1) \to \forall\, X : \kappa_2.\, T_2 \ni \lambda\, f.\, \Lambda\, X.\, s_2'\ (f \cdot X)}$$

where $\mathcal{D}_1$ is the derivation

$$\dfrac{\dfrac{\Gamma, X : \kappa_2 \vdash T_1 \to T_2 \ni s_2}{\Gamma' \vdash T_1 \to T_2 \ni s_2} \quad \dfrac{\Gamma, X : \kappa_2 \vdash T_1 \to T_2 : \star}{\Gamma' \vdash T_1 \to T_2 : \star}}{\Gamma' \vdash \chi\ (T_1 \to T_2)\ \text{-}\ s_2 \in T_1 \to T_2}$$

and where $\mathcal{D}_2$ is the derivation

$$\dfrac{\dfrac{\dfrac{(f : \forall\, X : \kappa_1.\, T_1) \in \Gamma'}{\Gamma' \vdash f \in \forall\, X : \kappa_1.\, T_1} \quad \dfrac{(X : \kappa_2) \in \Gamma'}{\Gamma' \vdash X : \kappa_2} \quad \kappa_1 \cong \kappa_2}{\Gamma' \vdash f \cdot X \in T_1} \quad T_1 \cong T_1}{\Gamma' \vdash T_1 \ni f \cdot X}$$

Case:

$$\dfrac{\begin{array}{ll} \Gamma \vdash S_2 \leqslant_s S_1 \rightsquigarrow s_1 & s_1' =_{\mathbf{df}} \chi\ (S_2 \to S_1)\ \text{-}\ s_1 \\ \Gamma, x : S_2 \vdash T_1[s_1'\ x/x] \leqslant_s T_2 \rightsquigarrow s_2 & s_2' =_{\mathbf{df}} \chi\ (T_1[s_1'\ x/x] \to T_2)\ \text{-}\ s_2 \end{array}}{\Gamma \vdash \Pi\, x : S_1.\, T_1 \leqslant_s^{\mathbf{n}} \Pi\, x : S_2.\, T_2 \rightsquigarrow \lambda\, f.\, \lambda\, x.\, s_2'\ (f\ (s_1'\ x))}$$

By inversion of the assumed kinding derivation of the subjects of subtyping,

$$\dfrac{\Gamma \vdash S_i : \star \quad \Gamma, x : S_i : T_i : \star}{\Gamma \vdash \Pi\, x : S_i.\, T_i : \star}$$

for $i \in \{1, 2\}$.

Below, let $\Gamma' =_{\mathbf{df}} \Gamma, f : \Pi\, x : S_1.\, T_1, x : S_2$.

- Obtain $\Gamma \vdash S_2 \to S_1 \ni s_1$ with $|s_1| =_{\beta\eta} \lambda\, x.\, x$ from the IH applied to $\Gamma \vdash S_2 \leqslant_s S_1 \rightsquigarrow s_1$ and $\Gamma \vdash S_i : \star$.
- Obtain $\Gamma' \vdash s_1' \in S_2 \to S_1$ with $|s_1'| =_{\beta\eta} \lambda\, x.\, x$ from $\Gamma \vdash S_2 \to S_1 \ni s_1$ with $|s_1| = \lambda\, x.\, x$, $\Gamma \vdash S_i : \star$, weakening, and the typing and erasure rule for χ.
- Obtain $\Gamma, y : S_2 \vdash T_1[s_1'\ x/x] : \star$ from $\Gamma, x : S_1 \vdash T_1 : \star$, weakening, $\Gamma \vdash s_1' \in S_2 \to S_1$, and Lemma 20 of [83].

- Obtain $\Gamma, x\!:\!S_2 \vdash T_1[s_1'\ x/x] \to T_2 \ni s_2$ with $|s_2| =_{\beta\eta} \lambda\, x.\, x$ from the IH applied to $\Gamma, x : S_2 \vdash T_1[s_1'\ x/x] \leqslant_s T_2 \rightsquigarrow s_2$ and $\Gamma, x : S_2 \vdash T_1[s_1'\ x/x] : \star$, and (from weakening) $\Gamma, x\!:\!S_2 \vdash s \in R_1 \to R_2$.
- Obtain $\Gamma' \vdash s_2' \in T_1[s_1'\ x/x] \to T_2$ with $|s_2'| =_{\beta\eta} \lambda\, x.\, x$ from $\Gamma, x : S_2 \vdash T_1[s_1'\ x/x] \to T_2 \ni s_2$ with $|s_2| =_{\beta\eta} \lambda\, x.\, x$, weakening, and the typing and erasure rule for χ.

  For the erasure,

$$\begin{array}{rcll} |\lambda\, f.\, \lambda\, x.\, s_2'\ (f\ (s_1'\ x))| & =_{\beta\eta} & \lambda\, f.\, \lambda\, x.\, (\lambda\, x.\, x)\ (f\ ((\lambda\, x.\, x)\ x)) & \text{erasure, IH} \\ & =_{\beta\eta} & \lambda\, f.\, \lambda\, x.\, f\ x & \beta\text{-reduction} \\ & =_{\beta\eta} & \lambda\, f.\, f & \eta\text{-contraction} \end{array}$$

  For the typing,

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma' \vdash s_2' \in T_1[s_1'\ x/x] \to T_2 \quad \mathcal{D}_1 :: \Gamma' \vdash T_1[s_1'\ x/x] \ni f\ (s_1'\ x)}{\Gamma' \vdash s_2'\ (f\ (s_1'\ x)) \in T_2}}{\Gamma' \vdash T_2 \ni s_2'\ (f\ (s_1'\ x))}}{\Gamma, f\!:\!\Pi\, x\!:\!S_1.\, T_1 \vdash \Pi\, x\!:\!S_2.\, T_2 \ni \lambda\, x.\, s_2'\ (f\ (s_1'\ x))}}{\Gamma \vdash (\Pi\, x\!:\!S_1.\, T_1) \to \Pi\, x\!:\!S_2.\, T_2 \ni \lambda\, f.\, \lambda\, x.\, s_2'\ (f\ (s_1'\ x))}$$

where $\mathcal{D}_1$ is the derivation

$$\dfrac{\dfrac{(f\!:\!\Pi\, x\!:\!S_1.\, T_1) \in \Gamma'}{\Gamma' \vdash f \in \Pi\, x\!:\!S_1.\, T_1} \qquad \dfrac{\dfrac{\Gamma' \vdash s_1' \in S_2 \to S_1 \quad \dfrac{\dfrac{(x\!:\!S_2) \in \Gamma'}{\Gamma' \vdash x \in S_2}}{\Gamma' \vdash S_2 \ni x}}{\Gamma' \vdash s_1'\ x \in S_1}}{\Gamma' \vdash S_1 \ni s_1'\ x}}{\dfrac{\Gamma' \vdash f\ (s_1'\ x) \in T_1[s_1'\ x/x]}{\Gamma' \vdash T_1[s_1'\ x/x] \ni f\ (s_1'\ x)}}$$

Case:

$$\dfrac{\begin{array}{ll} \Gamma \vdash S_2 \leqslant_s S_1 \rightsquigarrow s_1 & s_1' =_{\mathbf{df}} \chi\ (S_2 \to S_1)\ \text{-}\ s_1 \\ \Gamma, x\!:\!S_2 \vdash T_1[s_1'\ x/x] \leqslant_s T_2 \rightsquigarrow s_2 & s_2' =_{\mathbf{df}} \chi\ (T_1[s_1'\ x/x] \to T_2)\ \text{-}\ s_2 \end{array}}{\Gamma \vdash \forall\, x\!:\!S_1.\, T_1 \leqslant_s^{\mathbf{n}} \forall\, x\!:\!S_2.\, T_2 \rightsquigarrow \lambda\, f.\, \Lambda\, x.\, s_2'\ (f\ \text{-}(s_1'\ x))}$$

Similar to the case above, though now in the typing rule we must confirm (easily) that $x \notin FV(|s_2'\ (f\ \text{-}(s_1'\ x))|)$, and we do not need η-contraction when we reason about the erasure of the produced coercion.

Case:

$$\dfrac{\begin{array}{ll} \Gamma \vdash S_1 \leqslant_s S_2 \rightsquigarrow s_1 & s_1' =_{\mathbf{df}} \chi\ (S_1 \to S_2)\ \text{-}\ s_1 \\ \Gamma, x\!:\!S_1 \vdash T_1 \leqslant_s T_2[s_1'\ x/x] \rightsquigarrow s_2 & s_2' =_{\mathbf{df}} \chi\ (T_1 \to T_2[s_1'\ x/x])\ \text{-}\ s_2 \end{array}}{\Gamma \vdash \iota\, x\!:\!S_1.\, T_1 \leqslant_s^{\mathbf{n}} \iota\, x\!:\!S_2.\, T_2 \rightsquigarrow \lambda\, x.\, [s_1'\ x.1, s_2'[x.1/x]\ x.2]}$$

By inversion of the assumed kinding derivation of the subjects of subtyping,

$$\frac{\Gamma \vdash S_i : \star \quad \Gamma, x : S_i : T_i : \star}{\Gamma \vdash \iota\, x : S_i .\, T_i : \star}$$

for $i \in \{1, 2\}$.

Below, let $\Gamma' =_{\mathbf{df}} \Gamma, x : \iota\, x : S_1 .\, T_1$.

- Obtain $\Gamma \vdash S_1 \to S_2 \ni s_1$ with $|s_1| =_{\beta\eta} \lambda\, x.\, x$ from the IH applied to $\Gamma \vdash S_1 \leqslant_s S_2 \rightsquigarrow s_1$ and $\Gamma \vdash S_i : \star$.
- Obtain $\Gamma' \vdash s_1' \in S_1 \to S_2$ with $|s_1'| =_{\beta\eta} \lambda\, x.\, x$ from $\Gamma \vdash S_1 \to S_2 \ni s_1$ with $|s_1| = \lambda\, x.\, x$, $\Gamma \vdash S_i : \star$, weakening, and the typing and erasure rule for χ.
- Obtain $\Gamma, x : S_1 \vdash T_2[s_1'\ x/x] : \star$ from $\Gamma, x : S_2 \vdash T_2 : \star$, $\Gamma \vdash s_1' \in S_1 \to S_2$, and Lemma 20 of [83].
- Obtain $\Gamma, x : S_1 \vdash T_1 \to T_2[s_1'\ x/x] \ni s_2$ with $|s_2| =_{\beta\eta} \lambda\, x.\, x$ from the IH applied to $\Gamma, x : S_1 \vdash T_1 \leqslant_s T_2[s_1'\ x/x] \rightsquigarrow s_2$ and $\Gamma, x : S_1 \vdash T_2[s_1'\ x/x] : \star$, and (from weakening) $\Gamma, x : S_1 \vdash s \in R_1 \to R_2$.
- Obtain $\Gamma' \vdash s_2'[x.1/x] \in T_1[x.1/x] \to T_2[s_1'\ x.1/x]$ with $|s_2'[x.1/x]| =_{\beta\eta} \lambda\, x.\, x$ from $\Gamma, x : S_1 \vdash T_1 \to T_2[s_1'\ x/x] \ni s_2$ with $|s_2| =_{\beta\eta} \lambda\, x.\, x$ and weakening and Lemma 20 of [83] and the typing and erasure rules for χ.

For the erasure,

$$\begin{array}{lll} |\lambda\, x.\, [s_1'\ x.1, s_2'[x.1/x]\ x.2]| & =_{\beta\eta} & \lambda\, x.\, (\lambda\, x.\, x)\ x \\ & = & \lambda\, x.\, x \qquad\qquad \beta\text{-reduction} \end{array}$$

For the typing, observe

$$|s_1'\ x.1| =_{\beta\eta} x =_{\beta\eta} |s_2'[x.1/x]\ x.2|$$

And now deduce

$$\dfrac{\dfrac{\mathcal{D}_1 :: \Gamma' \vdash s_1'\ x.1 \in S_2}{\Gamma' \vdash S_2 \ni s_1'\ x.1} \quad \dfrac{\mathcal{D}_2 :: \Gamma' \vdash s_2'[x.1/x]\ x.2 \in T_2[s_1'\ x.1/x]}{\Gamma' \vdash T_2[s_1'\ x.1/x] \ni s_2'[x.1/x]\ x.2} \quad |s_1'\ x.1| =_{\beta\eta} |s_2'[x.1/x]\ x.2|}{\dfrac{\Gamma' \vdash \iota\, x : S_2 .\, T_2 \ni [s_1'\ x.1, s_2'[x.1/x]\ x.2]}{\Gamma \vdash (\iota\, x : S_1 .\, T_1) \to \iota\, x : S_2 .\, T_2 \ni \lambda\, x.\, [s_1'\ x.1, s_2'[x.1/x]\ x.2]}}$$

where $\mathcal{D}_1$ is the derivation

$$\dfrac{\Gamma' \vdash s_1' \in S_1 \to S_2 \qquad \dfrac{\dfrac{\dfrac{(x : \iota\, x : S_1 .\, T_1) \in \Gamma'}{\Gamma' \vdash x \in \iota\, x : S_1 .\, T_1}}{\Gamma' \vdash x.1 \in S_1}}{\Gamma' \vdash S_1 \ni x.1}}{\Gamma' \vdash s_1'\ x.1 \in S_2}$$

and $\mathcal{D}_2$ is the derivation

$$\dfrac{\Gamma' \vdash s_2'[x.1/x] \in T_1[x.1/x] \to T_2[s_1'\ x.1/x] \quad \dfrac{\dfrac{\dfrac{(x : \iota\, x : S_1 .\, T_1) \in \Gamma'}{\Gamma' \vdash x \in \iota\, x : S_1 .\, T_1}}{\Gamma' \vdash x.2 \in T_1[x.1/x]}}{\Gamma' \vdash T_1[x.1/x] \ni x.2}}{\Gamma' \vdash s_2'[x.1/x]\ x.2 \in T_2[s_1'\ x.1/x]}$$

□

Proof (of Corollary 4.4). By cases on the derivation of the assumed positivity judgement.

Case:

$$\frac{F \dashrightarrow^* \lambda\, R{:}\star.\, T \quad \Gamma' =_{\mathbf{df}} \Gamma, R_1{:}\star, R_2{:}\star, z{:}\mathit{Cast}\cdot R_1\cdot R_2 \quad \Gamma' \vdash T[R_1/R] \leqslant_{\mathit{elimCast}\ \text{-}z} T[R_2/R] \rightsquigarrow t}{\Gamma \vdash +F \rightsquigarrow \Lambda\, R_1.\, \Lambda\, R_2.\, \Lambda\, z.\, \mathit{intrCast}\ \text{-}t\ \text{-}(\lambda\, x.\, \beta)}$$

By Proposition 3.25, since $\Gamma \vdash F : \star \to \star$ and $F \dashrightarrow^* \lambda\, R : \star.\, T$, we have $\Gamma \vdash \lambda\, R : \star.\, T : \star \to \star$. By inversion of this, we have $\Gamma, R : \star \vdash T : \star$. Using Theorem 4.3, we obtain $\Gamma' \vdash T[R_1/R] \to T[R_2/R] \ni t$ and $|t| =_{\beta\eta} \lambda\, x.\, x$ (note that $\Gamma' \vdash \mathit{elimCast}\ \text{-}z \in R_1 \to R_2$ and $|\mathit{elimCast}\ \text{-}z| = \lambda\, x.\, x$).

It is clear by the erasure rule for *intrCast* (Figure 3.10) that the retyping function produced in the conlcusion of the rule is convertible with $\lambda\, x.\, x$ modulo erasure. We now construct a typing derivation to finish the proof (recall the definition of *Mono* in Figure 3.12).

$$\dfrac{\dfrac{\dfrac{\mathcal{D} :: \Gamma' \vdash \mathit{Cast}\cdot(F\cdot R_1)\cdot(F\cdot R_2) \ni \mathit{intrCast}\ \text{-}t\ \text{-}(\lambda\, x.\, \beta)}{\Gamma, R_1{:}\star, R_2{:}\star \vdash \mathit{Cast}\cdot R_1\cdot R_2 \Rightarrow \mathit{Cast}\cdot(F\cdot R_1)\cdot(F\cdot R_2) \ni \Lambda\, z.\, \mathit{intrCast}\ \text{-}t\ \text{-}(\lambda\, x.\, \beta)}}{\Gamma, R_1{:}\star \vdash \forall\, R_2{:}\star.\, \mathit{Cast}\cdot R_1\cdot R_2 \Rightarrow \mathit{Cast}\cdot(F\cdot R_1)\cdot(F\cdot R_2) \ni \Lambda\, R_2.\, \Lambda\, z.\, \mathit{intrCast}\ \text{-}t\ \text{-}(\lambda\, x.\, \beta)}}{\Gamma \vdash \mathit{Mono}\cdot F \ni \Lambda\, R_1.\, \Lambda\, R_2.\, \Lambda\, z.\, \mathit{intrCast}\ \text{-}t\ \text{-}(\lambda\, x.\, \beta)}$$

where $\mathcal{D}$ is the derivation

$$\dfrac{\dfrac{\Gamma' \vdash T[R_1/R] \to T[R_2/R] \ni t \qquad \dfrac{\dfrac{|t\ x| =_{\beta\eta} x}{\Gamma', x{:}T[R_1/R] \vdash \{t\ x \simeq x\} \ni \beta}}{\Gamma' \vdash \Pi\, x{:}T[R_1/R].\, \{t\ x \simeq x\} \ni \lambda\, x.\, \beta}}{\Gamma' \vdash \mathit{intrCast}\ \text{-}t\ \text{-}(\lambda\, x.\, \beta) \in \mathit{Cast}\cdot T[R_1/R]\cdot T[R_2/R]} \qquad \ldots \cong \ldots}{\Gamma' \vdash \mathit{Cast}\cdot(F\cdot R_1)\cdot(F\cdot R_2) \ni \mathit{intrCast}\ \text{-}t\ \text{-}(\lambda\, x.\, \beta)}$$

□

A.2 Decidability

Proof (of Proposition 4.5, syntax-directedness). By inspection of the inference rules. For judgments $\Gamma \vdash +F \rightsquigarrow t$ and $\Gamma \vdash S \leqslant_s T \rightsquigarrow t$, there is only ever one inference rule that can apply. For judgment $\Gamma \vdash S \leqslant_s^{\mathbf{n}} T \rightsquigarrow t$, every rule except the two for variables requires that the subjects of subtyping (S and T) are formed using a particular type construct, and these do not overlap with each other. In the case for variables, in the second rule we have a premise that requires the two nominally distinct meta-variables denote *actually* distinct type variables, which disambiguates it from the reflexive variable rule.

□

Proof (of Proposition 4.5, mode-correctness). By inspection of the inference rules. We walk through one particular case.

Case:

$$\frac{\begin{array}{ll}\Gamma \vdash S_2 \leqslant_s S_1 \rightsquigarrow s_1 & s_1' =_{\mathbf{df}} \chi\ (S_2 \to S_1)\ \text{-}\ s_1 \\ \Gamma, y : S_2 \vdash T_1[s_1'\ y/x] \leqslant_s T_2[y/x] \rightsquigarrow s_2 & s_2' =_{\mathbf{df}} \chi\ (T_1[s_1'\ y/x] \to T_2[y/x])\ \text{-}\ s_2\end{array}}{\Gamma \vdash \Pi\, x : S_1.\, T_1 \leqslant_s \Pi\, x : S_2.\, T_2 \rightsquigarrow \lambda\, f.\, \lambda\, x.\, s_2'[x/y]\ (f\ (s_1'\ x))}$$

- We assume that context Γ and types $\Pi\, x : S_1.\, T_1$ and $\Pi\, x : S_2.\, T_2$ are inputs.
- For $\Gamma \vdash S_2 \leqslant S_1 \rightsquigarrow s_1$, the inputs Γ, S_2, S_1 are readily constructed, and we assume the output s_1 is available.
- We can construct s_1' from what is available: S_2, S_1, and s_1.
- For $\Gamma, y : S_2 \vdash T_1[s_1'\ y/x] \leqslant_s T_2[y/x] \rightsquigarrow s_2$, the inputs $\Gamma, y : S_2$, $T_1[s_1'\ y/x]$, and $T_2[y/x]$ are readily constructed, and we assume output s_2 is available.

  Here we are assuming (uncontroversially) that it is possible to pick a variable name y such that $y \notin DV(\Gamma)$.
- We can construct s_2' from what is available: $T_1[s_1'\ y/x]$, $T_2[y/x]$, and s_2.
- Finally, for the output of the conclusion, we can construct $\lambda\, f.\, \lambda\, x.\, s_2'[x/y]\ (f\ (s_1'\ x))$ (again assuming we can pick variable name f with $f \notin DV(\Gamma)$).

□

Definition A.1 (Measure for subtyping decidability)**.**

$$\begin{array}{lcl}
\|T\|_{\kappa \neq \star} & = & 0 \\
\|X\|_\star & = & 0 \\
\|T\ t\|_\star & = & 0 \\
\|T_1 \cdot T_2\|_\star & = & 0 \\
\|\{t_1 \simeq t_2\}\|_\star & = & 0 \\
\|\forall\, X : \kappa.\, T\|_\star & = & 1 + \|\kappa\| + \|T\|_\star \\
\|\Pi\, x : T_1.\, T_2\|_\star & = & 1 + \|T_1\|_\star + \|T_2\|_\star \\
\|\forall\, x : T_1.\, T_2\|_\star & = & 1 + \|T_1\|_\star + \|T_2\|_\star \\
\|\iota\, x : T_1.\, T_2\|_\star & = & 1 + \|T_1\|_\star + \|T_2\|_\star \\
\\
\|\star\| & = & 0 \\
\|\Pi\, x : T.\, \kappa\| & = & 1 + \|T\|_\star + \|\kappa\| \\
\|\Pi\, X : \kappa_1.\, \kappa_2\| & = & 1 + \|\kappa_1\| + \|\kappa_2\|
\end{array}$$

Lemma A.2. *For all* $\vdash \Gamma_1, x : T, \Gamma_2$ *and* $\Gamma_1 \vdash t : T$*,*

- *For all* κ *with* $\Gamma_1, x : T, \Gamma_2 \vdash \kappa$*,* $\|\kappa[t/x]\| = \|\kappa\|$
- *For all* S *and* κ *with* $\Gamma_1, x : T, \Gamma_2 \vdash S : \kappa$*,* $\|S[t/x]\|_{\kappa[t/x]} = \|S\|_\kappa$

Proof. By a straightforward inductive argument on the assumed typing and kinding derivation. □

Proof (of Theorem 4.6). By mutual well-founded induction on the lexicographic ordering of the triple (n, w), where

- $n \in \mathbf{N}$, where $n = \|\mathbf{N}(S)\|_\star + \|\mathbf{N}(T)\|_\star$.

- $w \in \{0, 1\}$, with $w = 1$ when we are considering the rule for judgment $\Gamma \vdash S \leqslant_s T \rightsquigarrow t$ and $w = 0$ when we are considering the rules for judgment $\Gamma \vdash S \leqslant_s^{\mathbf{n}} T \rightsquigarrow t$.

We proceed by simultaneous cases on the subjects of the subtyping judgements. Observe that decidability of positivity checking follows as a directly consequence of decidability of subtyping.

Case: No assumptions on S and T
(Measure: $\|\mathbf{N}(S)\|_\star + \|\mathbf{N}(T)\|_\star, 1$).

There is only one rule which can possibly apply for judgment $\Gamma \vdash S \leqslant_s T \rightsquigarrow t$.

$$\frac{S \dashrightarrow^* S' \not\dashrightarrow \quad T \dashrightarrow^* T' \not\dashrightarrow \quad \Gamma \vdash S' \leqslant_s^{\mathbf{n}} T' \rightsquigarrow t}{\Gamma \vdash S \leqslant_s T \rightsquigarrow t}$$

First, observe that S' and T' can always be acquired as reduction to weak head normal form is terminating. Next, note $\mathbf{N}(S) = \mathbf{N}(S')$ and $\mathbf{N}(T) = \mathbf{N}(T')$, with S' and T' satisfying the condition of termination under full β-reduction (since S and T are terminating). We invoke the IH on the final premise (measure: $(\|\mathbf{N}(S')\|_\star + \|\mathbf{N}(T')\|_\star, 0) < (\|\mathbf{N}(S)\|_\star + \|\mathbf{N}(T)\|_\star, 1)$), and the final decision is yes iff we obtain "yes" for the premise.

Case: Both S and T are not in weak head normal form.
(Measure: $\|\mathbf{N}(S)\|_\star + \|\mathbf{N}(T)\|_\star, 0$)

Reject the derivation, as the judgment $\Gamma \vdash S \leqslant_s^{\mathbf{n}} T \rightsquigarrow t$ can obly be derived if both S and T are in weak head normal form.

Case: $S = Y$ and T is in weak head normal form.
(Measure: $(\|\mathbf{N}(T)\|_\star, 0)$).

Proceed by cases on T.

Subcase: $T = Y$
(Measure: $(0, 0)$)

We give an affirmative decision by applying the rule

$$\frac{}{\Gamma \vdash Y \leqslant_s^{\mathbf{n}} Y \rightsquigarrow \lambda\, x.\, x}$$

Subcase: $T = Z$ with $Z \neq Y$
(Measure: $(0, 0)$).

It is decidable whether $Y \to Z \cong R_1 \to R_2$.

- If yes, affirm the derivation with rule

$$\frac{Z \neq Y \quad \Gamma \vdash s \in Y \to Z}{\Gamma \vdash Y \leqslant_s Z \rightsquigarrow s}$$

- If no, reject the derivation. Assuming $\Gamma \vdash Y \leqslant_s Z \rightsquigarrow t$ for some t, inversion of the rule tells us either $Y = Z$ (it cannot) or $\Gamma \vdash s \in Y \to Z$ (it cannot).

Subcase: T is not a variable
(Measure: $(\|\mathbf{N}(T)\|_\star, 0)$).

Reject the derivation (this judgment only relates type variables to type variables).

Case: $S = S'\ t_1$ and T in weak head normal form
(Measure: $(0 + \|T\|_\star, 0)$).

There is only one applicable rule of the subtyping judgement, and it requires that T is an application of a type to a term.

Subcase: $T = T'\ t_2$
(Measure: $(0, 0)$).

We must affirm or refute a derivation ending with the rule

$$\frac{S'\ t_1 \cong T'\ t_2}{\Gamma \vdash S'\ t_1 \leqslant^{\mathbf{n}}_s T'\ t_2 \rightsquigarrow s'}$$

Since S and T are terminating, it is decidable whether $S'\ t_1 \cong T'\ t_2$. Affirmation or refutation of this implies affirmation or refutation of the whole derivation, resp.

Case: $S = S_1 \cdot S_2$ and T in weak head normal form
(Measure: $(0 + \|T\|_\star, 0)$).

Similar to the above case.

Case: $S = \{s_1 \simeq s_2\}$
(Measure: $(0 + \|T\|_\star, 0)$).

There is only one applicable rule of the subtyping judgement, and it requires that T is an equality type.

Subcase: $T = \{t_1 \cong t_2\}$.
(Measure: $(0, 0)$.)

Since S and T are terminating, it is decidable whether $\{s_1 \simeq s_2\} \cong \{t_1 \simeq t_2\}$. Affirmation or refutation of this implies affirmation or refutation of the whole derivation, resp.

Case: $S = \forall\, X : \kappa_1.\, T_1$ and T in weak head normal form.
(Measure: $(1 + \|\mathbf{N}(\kappa_1)\| + \|\mathbf{N}(T_1)\|_\star + \|\mathbf{N}(T)\|_\star, 0)$.)

There is only one applicable rule of the subtyping judgment, and it requires that T is quantification over a type constructor. If this is not the case for T, we refute the derivation of $\Gamma \vdash S \leqslant^{\mathbf{n}}_s T \rightsquigarrow t$: no rule applies that does not contradict this fact or the assumption that T is in weak head normal form.

Subcase: $T = \forall\, X : \kappa_2.\, T_2$.
(Measure: $(2 + \|\mathbf{N}(\kappa_1)\| + \|\mathbf{N}(T_1)\|_\star + \|\mathbf{N}(\kappa_2)\| + \|\mathbf{N}(T_2)\|_\star, 0)$)

We must affirm or refute a derivation which ends with an application of the rule

$$\frac{\kappa_1 \cong \kappa_2 \quad \Gamma, X : \kappa_1 \vdash T_1 \leqslant_s T_2 \rightsquigarrow s_2 \quad s_2' =_{\mathbf{df}} \chi\ (T_1 \to T_2)\ \text{-}\ s_2}{\Gamma \vdash \forall\, X : \kappa_1.\, T_1 \leqslant^{\mathbf{n}}_s \forall\, X : \kappa_2.\, T_2 \rightsquigarrow \lambda\, f.\, \Lambda\, X.\, s_2'\ (f \cdot X)}$$

- By the assumption that S and T are terminating, $\kappa_1 \cong \kappa_2$ is decidable.
  If this is refuted, refute the whole derivation by inversion of the subtyping derivation.
- Invoke the IH to decide $\Gamma, X : \kappa_1 \vdash T_1 \leqslant_s T_2 \rightsquigarrow s_2$ (measure: $(\|\mathbf{N}(T_1)\|_\star + \|\mathbf{N}(T_2)\|_\star, 1) < (2 + \|\mathbf{N}(\kappa_1)\| + \|\mathbf{N}(T_1)\|_\star + \|\mathbf{N}(\kappa_2)\| + \|\mathbf{N}(T_2)\|_\star, 0)$).
  If this is refuted, we refute the whole derivation by inversion of the subtyping derivation.

Now that all of the premises have been affirmed, we can affirm the derivation by applying the rule.

Case: $S = \Pi\, x : S_1.\, T_1$.
(Measure: $(1 + \|\mathbf{N}(S_1)\|_\star + \|\mathbf{N}(T_1)\|_\star + \|\mathbf{N}(T)\|_\star, 0)$)

There is only one applicable rule of the subtyping judgement, and it requires that T is of the form $\Pi\, x : S_2.\, T_2$.

Subcase: $T = \Pi\, x : S_2.\, T_2$.
(Measure: $(2 + \|\mathbf{N}(S_1)\|_\star + \|\mathbf{N}(T_1)\|_\star + \|\mathbf{N}(S_2)\|_\star + \|\mathbf{N}(T_2)\|_\star, 0)$)

We must affirm or refute a derivation ending with a use of the rule

$$\frac{\begin{array}{ll}\Gamma \vdash S_2 \leqslant_s S_1 \rightsquigarrow s_1 & s_1' =_{\mathbf{df}} \chi\ (S_2 \to S_1)\ \text{-}\ s_1 \\ \Gamma, x : S_2 \vdash T_1[s_1'\ x/x] \leqslant_s T_2 \rightsquigarrow s_2 & s_2' =_{\mathbf{df}} \chi\ (T_1[s_1'\ x/x] \to T_2)\ \text{-}\ s_2\end{array}}{\Gamma \vdash \Pi\, x : S_1.\, T_1 \leqslant_s^{\mathbf{n}} \Pi\, x : S_2.\, T_2 \rightsquigarrow \lambda\, f.\, \lambda\, x.\, s_2'\ (f\ (s_1'\ x))}$$

- Since S and T are terminating, so are S_1, S_2, T_1, and T_2.
- By the IH, it is decidable whether $\Gamma \vdash S_2 \leqslant_s S_1 \rightsquigarrow s_1$ (measure: $(\|S_2\|_\star + \|S_1\|_\star, 1) < (2 + \|\mathbf{N}(S_1)\|_\star + \|\mathbf{N}(T_1)\|_\star + \|\mathbf{N}(S_2)\|_\star + \|\mathbf{N}(T_2)\|_\star, 0)$). If this is refuted, then so is the whole derivation.
- By Theorem A.1, $|s| =_{\beta\eta} \lambda\, x.\, x$. Therefore, $T_1[s_1'\ x/x]$ is terminating.
- By Lemma A.2, $\|T_1[s_1'\ x/x]\|_\star = \|T_1\|_\star$.
- By Lemma 20 of [83], $\Gamma, x : S_2 \vdash T_1[s_1'\ x/x] : \star$.
- By the IH, it is decidable whether $\Gamma, x : S_2 \vdash T_1[s_1'\ x/x] \leqslant_s T_2 \rightsquigarrow s_2$ (measure: $(\|T_2\|_\star + \|T_1\|_\star, 1) < (2 + \|\mathbf{N}(S_1)\|_\star + \|\mathbf{N}(T_1)\|_\star + \|\mathbf{N}(S_2)\|_\star + \|\mathbf{N}(T_2)\|_\star, 0)$). If this is refuted, then so is the whole derivation.

Conclude in affirmation by applying the rule.

Case: $S = \forall\, x : S_1.\, T_1$.
Similar to the above case.

Case: $S = \iota\, x : S_1.\, T_1$.
Similar to the above case. □

Proof (of Theorem 4.18). By induction on the assumed derivation.

Case:

$$\frac{}{\Gamma \vdash ()\ \leqslant_s\ ()\ \rightsquigarrow\ ()}$$

1. $\#() = \#() = \#() = 0$, and the implication with antecedent $i \in \emptyset$ is vacuous.

2. By inversion, given $\Gamma \vdash \tau : ()$ we know $\tau = ()$. Since $() \odot () = ()$ by definition, conclude $\Gamma \vdash () : ()$ by rule.

Case:

$$\frac{\Gamma \vdash S \leqslant_s T \rightsquigarrow s_1 \quad \Gamma, x{:}S \vdash \Delta_1 \leqslant_s \Delta_2[s'\ x/x] \rightsquigarrow \xi}{\Gamma \vdash (x{:}S)\Delta_1 \leqslant_s (x{:}T)\Delta_2 \rightsquigarrow (s_1)\xi}$$

- By the IH, $\#\xi = \#\Delta_1 = \#\Delta_2$, so $\#(s_1)\xi = \#(x : S)\Delta_1 = \#(x : T)\Delta_2$. Now, suffices to check $|s_1| =_{\beta\eta} \lambda\, x.\, x$, which we have by Theorem 4.3.
- Since $\Gamma \vdash \tau : (x : S)\Delta_1$, by inversion τ is of the form $(t)\tau'$ and $\Gamma \vdash t : S$. By Theorem 4.3, $\Gamma \vdash s_1 \in S \to T$, so it follows that $\Gamma \vdash s_1\ t \in T$. By the IH, $\Gamma, x{:}S \vdash \xi \odot \tau : \Delta_2$, and so it follows that $\Gamma \vdash \xi \odot \tau : \Delta_2[s_1\ t/x]$ (sequences do not have as free variables the bound variables of the telescopes) by Lemma 20 of [83]. Conclude with the rule

$$\frac{\Gamma \vdash T \ni s_1\ t \quad \Gamma \vdash \xi \odot \tau : \Delta_2[s_1\ x/t]}{\Gamma \vdash (s_1\ t)(\xi \odot \tau) : (x{:}T)\Delta_2}$$

Case:

$$\frac{\Gamma \vdash S \leqslant_s T \rightsquigarrow s_1 \quad \Gamma, x{:}S \vdash \Delta_1 \leqslant_s \Delta_2[s_1'\ x/x] \rightsquigarrow \xi}{\Gamma \vdash (\text{-}\ x{:}S)\Delta_1 \leqslant_s (\text{-}\ x{:}T)\Delta_2 \rightsquigarrow (\text{-}\ s_1)\xi}$$

Similar to the above case.

Case:

$$\frac{\kappa_1 \cong \kappa_2 \quad \Gamma, X{:}\kappa_1 \vdash \Delta_1 \leqslant_s \Delta_2 \rightsquigarrow \xi}{\Gamma \vdash (X{:}\kappa_1)\Delta_1 \leqslant_s (X{:}\kappa_2)\Delta_2 \searrow (\lambda\, X{:}\kappa_1.\, X)\xi}$$

1. From the IH, we can deduce $\#(\cdot\ (\lambda\, X : \kappa_1.\ ))\xi_1 = \#(X : \kappa_1)\Delta_1 = \#(X : \kappa_2)\Delta_2$. The implication that $(X : \kappa_1)$ is of the form $(x{:}T)$ or $(\text{-}\ x{:}T)$ is vacuous.
2. Assuming $\Gamma \vdash \tau : (X : \kappa_1)\Delta_1$, then by inversion τ is of the form $(\cdot\ T)\tau'$. We see that $\Gamma \vdash (\lambda\, X{:}\kappa_1.\, X) \cdot T : \kappa_2$ since $\kappa_1 \cong \kappa_2$, and we conclude as we have in earlier cases.

□

APPENDIX B
SIGNATURE ELABORATION PROOFS

Proposition B.1. *For all* Γ *and* Δ *with* $\vdash \Gamma$ *and* $\Gamma \vdash \Delta$*, the following rules are admissable.*

1.
$$\frac{\Gamma, \Delta \vdash T : \kappa}{\Gamma \vdash \lambda\, \Delta.\, T : \Pi\, \Delta.\, \kappa}$$

2.
$$\frac{\Gamma, \Delta \vdash T : \star}{\Gamma \vdash \Pi\, \Delta.\, T : \star}$$

Proof. By a straightforward induction on $\Gamma \vdash \Delta$. □

Definition B.2 (Erased term variables of a telescope)**.** We calculate the set of erased term variables declared in a telescope Δ whose variables are distinct from each other inductively as follows:

$$\begin{array}{lcl} EDV(()) & = & \emptyset \\ EDV((x{:}T)\Delta) & = & EDV(\Delta) \\ EDV((\text{-}\ x{:}T)\Delta) & = & \{x\} \cup EDV(\Delta) \\ EDV((\cdot\ X{:}\kappa)\Delta) & = & EDV(\Delta) \end{array}$$

Proposition B.3. *The following inference rule is admissable in CDLE.*

$$\frac{T \dashrightarrow^* \Pi\, \Delta.\, T' \quad \Gamma, \Delta \vdash T' \ni t \quad EDV(\Delta) \cap FV(|t|) = \emptyset}{\Gamma \vdash T \ni \lambda\, \Delta.\, t}$$

Proof. By induction on Δ.

Case: $\Delta = ()$

We have as premises $T \dashrightarrow^* T'$ and $\Gamma \vdash T' \ni t$. By Lemma 16 of [83], we have $\Gamma \vdash T \ni t$.

Case: $\Delta = (x{:}T'')\Delta'$

Note that $EDV((x{:}T'')\Delta') = EDV(\Delta')$. Appeal to the IH and derive

$$\frac{T \dashrightarrow^* \Pi\, x{:}T''.\, \Pi\, \Delta'.\, T' \qquad \dfrac{\Gamma, x{:}T'' \vdash \Pi\, \Delta'.\, T \ni \lambda\, \Delta'.\, t \quad EDV(\Delta') \cap FV(|t|) = \emptyset}{\Gamma, x{:}T'' \vdash \Pi\, \Delta'.\, T' \ni \lambda\, \Delta'.\, t}}{\Gamma \vdash T \ni \lambda\, x.\, \lambda\, \Delta'.\, t}$$

Case: $\Delta = (\cdot\ X{:}\kappa\Delta')$. Similar to the above case.

Case: $\Delta = (\text{-}\ x{:}T'')\Delta'$

Note that $EDV((\text{-}\ x:T'')\Delta') = \{x\} \cup EDV(\Delta')$. So, $EDV(((\text{-}\ x:T'')\Delta')) \cap FV(t) = \emptyset$ and distinctness of the variables of Δ implies $x \notin FV(|\lambda\,\Delta'.\,t|)$. Appeal to the IH and derive

$$\dfrac{T \dashrightarrow^* \forall\, x:T''.\,\Pi\,\Delta'.\,T' \qquad \dfrac{\Gamma, x:T'' \vdash \Pi\,\Delta'.\,T' \ni \lambda\,\Delta'.\,t \quad EDV(\Delta') \cap FV(|t|) = \emptyset}{\Gamma, x:T'' \vdash \Pi\,\Delta'.\,T' \ni \lambda\,\Delta'.\,t} \quad x \notin FV(|\lambda\,\Delta'.\,t|)}{\Gamma \vdash T \ni \Lambda\, x.\,\lambda\,\Delta'.\,t}$$

□

Proposition B.4. *The following rule is admissable.*

$$\dfrac{\Gamma \vdash t \in \Pi\,\Delta.\,T \quad \Gamma \vdash \tau : \Delta}{\Gamma \vdash t\ \tau \in T[\tau/\Delta]}$$

Proof. By induction on Δ.

Case: $\Delta = ()$.

By inversion of $\Gamma \vdash \tau : \Delta$, $\tau = ()$, so $t\ \tau = t$ and $\Pi\,\Delta.\,T = T$. So, conclude with assumption $\Gamma \vdash t \in T$.

Case: $\Delta = (x:T')\Delta'$

- By inversion of $\Gamma \vdash \tau : \Delta$, τ is of the form $(t')\tau'$.
- By inversion of $\Gamma \vdash (t')\tau' : (x:T')\Delta'$, obtain $\Gamma \vdash \tau[t'/x] : \Delta'[t'/x]$.

Conclude by appealing to the IH: we obtain $\Gamma \vdash (t\ t')\ \tau' : T[t'/x][\tau'/\Delta']$, and observe $(t\ t')\ \tau' = t\ \tau'$ and $T[t'/x][\tau'/\Delta'] = T[\tau/\Delta]$.

Case: $\Delta = (\text{-}\ x:T')\Delta'$

Similar to the above case.

Case: $\Delta = (\cdot\ X:\kappa)\Delta'$.

Similar to the above case.

□

Proposition B.5. *For all Γ_1, Δ, Γ_2 with $\vdash \Gamma_1, \Delta, \Gamma_2$, we have $\Gamma_1, \Delta, \Gamma_2 \vdash \Delta : \Delta$.*

Proof. By a straightforward induction on $\vdash \Gamma_1, \Delta, \Gamma_2$. □

Proposition B.6. *If $\Gamma \vdash \Delta \searrow \Delta'$ then*

- *For all t, $\lambda\,\Delta.\,t = \lambda\,\Delta'.\,t$.*

Proof. By a straightforward induction on the derivation of $\Gamma \vdash \Delta \searrow \Delta'$. □

Fact B.7. *If $\Gamma \vdash Decl[D, \Delta] \searrow \mathcal{S}$ then the following properties hold.*

- *$Seq(\Delta) = Seq(\mathcal{S}.\Xi)$*
- *if $(c_i : \Pi\,\Delta_i.\,D) \in \Delta$ then $\mathcal{S}.\Xi(i)$ is of the form $(c_i : \Pi\,\Delta_i'[\mathcal{S}.R/D].\,D)$ where $\Gamma, D:\star \vdash \Delta_i \searrow \Delta_i'$*

Fact B.8. *If $\Gamma \vdash Decl[D, \Delta] \searrow (\Gamma', \mathcal{C})$ valid then the following properties hold.*

- $Seq(\Delta) = Seq(\mathcal{C}.\Delta)$
- *if* $(c_i : \Delta_i D) \in \Delta$ *then*
  - $\mathcal{C}.\Delta(i)$ *is of the form* $(c_i : \forall\, \mathcal{C}.R : \star.\, \Pi\, \Delta_i'[\mathcal{C}.R/D].\, sig(\mathcal{C}) \cdot \mathcal{C}.R)$ *where* $\Gamma \vdash \Delta_i \searrow \Delta_i'$
  - $|\mathcal{C}.\tau(i)\ \Delta_i| =_{\beta\eta} \lambda\, \Delta.\, c_i\ |\Delta_i|$

B.1 Datatype Signature and Signature Constructors

Proof (of Proposition 4.28).

1. $\Gamma', \mathcal{S}.R : \star, \mathcal{S}.D : \star \vdash \mathcal{S}.\Xi$

   By induction over $\#\mathcal{S}.\Xi$, using the assumption and weakening at each step.

2. $\Gamma' \vdash sig(\mathcal{S}) : \star \to \star$

   Derive

$$\dfrac{\dfrac{\dfrac{\dfrac{(\mathcal{S}.D : \star) \in \Gamma', \mathcal{S}.R : \star, \mathcal{S}.D : \star, \mathcal{S}.\Xi}{\Gamma', \mathcal{S}.R : \star, \mathcal{S}.D : \star, \mathcal{S}.\Xi \vdash \mathcal{S}.D : \star}}{\Gamma', \mathcal{S}.R : \star, \mathcal{S}.D : \star \vdash \Pi\, \mathcal{S}.\Xi.\, \mathcal{S}.D : \star}\ (a)}{\Gamma', \mathcal{S}.R : \star \vdash \forall\, \mathcal{S}.D : \star.\, \Pi\, \mathcal{S}.\Xi.\, \mathcal{S}.D : \star}}{\Gamma' \vdash \lambda\, \mathcal{S}.R : \star.\, \forall\, \mathcal{S}.D : \star.\, \Pi\, \mathcal{S}.\Xi.\, \mathcal{S}.D : \star \to \star}$$

   (a) By the above and Proposition B.1.

□

Proof (of Theorem 4.30). By inversion of the assumed derivation of $\vdash_{\mathbf{sig}}$,

$$\dfrac{R \notin \{D\} \cup DV(\Gamma) \cup \bigcup_{(c_i : \Pi\, \Delta_i.\, D) \in \Delta} DV(\Delta_i) \qquad \left(\dfrac{(c_i : \Pi\, \Delta_i.\, D) \in \Delta \quad \Gamma, D : \star \vdash \Delta_i \searrow \Delta_i'}{\Gamma \vdash_{\mathbf{sig}} (D, R, \Delta, i) \searrow (c_i, \Delta_i'[R/D])} \right)_{i \in \{1 \ldots \#\Delta\}}}{\Gamma \vdash_{\mathbf{sig}} Decl[D, \Delta] \searrow \mathsf{record}\ Sig\ \{D = D; R = R; \Xi_{\#\Delta}(i) = (c_i : \Pi\, \Delta_i'[R/D].\, D)\}}$$

forcing $\mathsf{record}\ Sig\ \{D = D; R = R; \Xi_{\#\Delta}(i) = (c_i : \Pi\, \Delta_i'[R/D].\, D)\}$ Now, let us check the premises of the single rule for the judgement we are trying to affirm.

Premise: $(\Gamma', R : \star, D : \star \vdash \mathcal{S}.\Xi(i))_{i \in 1 \ldots \#\Delta}$.

Pick $i \in \{1 \ldots \#\Delta\}$, and we have $\mathcal{S}.\Xi(i) = (c_i : \Pi\, \Delta_i'[R/D].\, D)$. Now, by assumption $\Gamma', D : \star \vdash \Delta_i'$, so by renaming and weakening $\Gamma, R : \star, D : \star \vdash \Delta_i'[R/D]$. By Proposition B.1, $\Gamma', R : \star, D : \star \vdash \Pi\, \Delta_i'.\, D : \star$

□

Proof (of Theorem 4.38). First, invert the assumed derivation of $\vdash_{\mathbf{con}}$ and the derivations of its premises.

$$\dfrac{\Gamma \vdash_{\mathbf{sig}} Decl[D, \Delta] \searrow \mathcal{S} \quad \mathcal{J}_1 :: \Gamma \vdash_{\mathbf{cty}} \mathcal{S} \searrow \Delta' \quad \mathcal{J}_2 :: \Gamma \vdash_{\mathbf{ctm}} (\mathcal{S}, \Delta') \searrow \tau}{\Gamma \vdash_{\mathbf{con}} Decl[D, \Delta] \searrow \mathsf{record}\ Con\ \{\mathcal{S} = \mathcal{S}; \Delta = \Delta'; \tau = \tau\}}$$

where $\mathcal{J}_1$, $\mathcal{J}_2$ are the derivations (with some inlining to save horizontal space)

$$\mathcal{J}_1 :: \dfrac{\left(\dfrac{(c_i : \Pi\, \Delta_i'.\, D) \in \mathcal{S}.\Xi)}{\Gamma \vdash_{\mathbf{cty}} (\mathcal{S}, i) \searrow (c_i, \forall\, \mathcal{S}.R : \star.\, \Pi\, \Delta_i'.\, sig(\mathcal{S}) \cdot \mathcal{S}.R)}\right)_{i \in \{1 \ldots \#\mathcal{S}.\Xi\}}}{\Gamma \vdash_{\mathbf{cty}} \mathcal{S} \searrow (\Delta'_{\#\mathcal{S}.\Xi}(i) =_{\mathbf{df}} (c_i : \forall\, \mathcal{S}.R : \star.\, \Pi\, \Delta_i'.\, sig(\mathcal{S}) \cdot \mathcal{S}.R))}$$

$$\mathcal{J}_2 :: \dfrac{\left(\dfrac{(c_i : \Pi\, \Delta_i'.\, D) \in \mathcal{S}.\Xi \quad T_i =_{\mathbf{df}} \forall\, \mathcal{S}.R : \star.\, \Pi\, \Delta_i'.\, sig(\mathcal{S}) \cdot \mathcal{S}.R}{\Gamma \vdash_{\mathbf{ctm}} (\mathcal{S}, \Delta, i) \searrow \chi\ T_i\ \text{-}\ \Lambda\, \mathcal{S}.R.\, \lambda\, \Delta_i'.\, \Lambda\, D.\, \lambda\, \mathcal{S}.\Xi.\, c\ \Delta_i'}\right)_{i \in \{1 \ldots \#\mathcal{S}.\Xi\}}}{\Gamma \vdash_{\mathbf{ctm}} (\mathcal{S}, \Delta') \searrow (\tau_{\#\mathcal{S}.\Xi}(i) = \chi\ T_i\ \text{-}\ \Lambda\, \mathcal{S}.R.\, \lambda\, \Delta_i'.\, \Lambda\, D.\, \lambda\, \mathcal{S}.\Xi.\, c\ \Delta_i')}$$

Now, let us derive each of the premises for the judgement we are trying to affirm.

Premise: $\Gamma \vdash_{\mathbf{con}} Decl[D, \Delta] \searrow \mathcal{C}$

By assumption.

Premise: $\Gamma \vdash Decl[D, \Delta] \searrow (\Gamma', \mathcal{C}.\mathcal{S})$ *valid*

From the premise, $\Gamma \vdash Decl[D, \Delta] \searrow \mathcal{C}.\mathcal{S}$. Apply Theorem 4.30.

Premise: $(\Gamma' \vdash \mathcal{C}.\Delta(i))_{i \in \{1 \ldots \#\mathcal{C}.\Delta\}}$

Pick arbitrary $i \in \{1 \ldots \#\mathcal{C}.\Delta\}$, and we see that $\mathcal{C}.\Delta(i) = (c_i : \forall\, \mathcal{S}.R : \star.\, \Pi\, \Delta_i'.\, sig(\mathcal{S}) \cdot \mathcal{S}.R))$ where $\Gamma \vdash_{\mathbf{cty}} (\mathcal{S}, i) \searrow (c_i, \forall\, \mathcal{S}.R : \star.\, \Pi\, \Delta_i'.\, sig(\mathcal{S}) \cdot \mathcal{S}.R)$. Derive

$$\dfrac{\dfrac{\dfrac{}{\Gamma', \mathcal{S}.R : \star, \Delta_i' \vdash sig(\mathcal{S}) : \star \to \star}\ 2 \quad \dfrac{(\mathcal{S}.R : \star) \in \Gamma', \mathcal{S}.R : \star, \Delta_i'}{\Gamma', \mathcal{S}.R : \star, \Delta_i' \vdash \mathcal{S}.R : \star}}{\Gamma', \mathcal{S}.R : \star, \Delta_i' \vdash sig(\mathcal{S}) \cdot \mathcal{S}.R}}{\dfrac{\Gamma', \mathcal{S}.R : \star \vdash \Pi\, \Delta_i'.\, sig(\mathcal{S}) \cdot \mathcal{S}.R}{\Gamma' \vdash \forall\, \mathcal{S}.R : \star.\, \Pi\, \Delta_i'.\, sig(\mathcal{S}) \cdot \mathcal{S}.R}\ 1}$$

1. By Proposition B.1.

   Note that we have $\Gamma', \mathcal{S}.R : \star \vdash \Delta_i'$ by assumption (see Definition 4.26) and renaming.

2. By Proposition 4.28 and weakening.

Premise: $\#\mathcal{C}.\Delta = \#\mathcal{C}.\tau$

Immediate, as both are equal to $\#\mathcal{C}.\mathcal{S}.\Xi$ (and thus equal to $\#\Delta$).

Premise: $(\Gamma' \vdash \mathcal{C}.\tau(i) : \mathcal{C}.\Delta(i))_{i \in \{1 \ldots \#\mathcal{C}.\Delta\}}$

Pick arbitrary $i \in \{1 \dots \#\mathcal{C}.\Delta\}$. Derive

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(c : \Pi\, \Delta'_i.\, D) \in \Gamma', \mathcal{S}.R : \star, \Delta'_i, D : \star, \mathcal{S}.\Xi}{\Gamma', \mathcal{S}.R : \star, \Delta'_i, D : \star, \mathcal{S}.\Xi \vdash c \in \Pi\, \Delta'_i.\, D} \quad \dfrac{}{\Gamma', \mathcal{S}.R : \star, \Delta'_i, D : \star, \mathcal{S}.\Xi \vdash \Delta'_i : \Delta'_i}\,3}{\Gamma', \mathcal{S}.R : \star, \Delta'_i, D : \star, \mathcal{S}.\Xi \vdash c\ \Delta'_i \in D}}{\Gamma', \mathcal{S}.R : \star, \Delta'_i, D : \star, \mathcal{S}.\Xi \vdash D \ni c\ \Delta'_i}}{\Gamma', \mathcal{S}.R : \star, \Delta'_i, D : \star \vdash \Pi\, \mathcal{S}.\Xi.\, D \ni \lambda\, \mathcal{S}.\Xi.\, c\ \Delta'_i}\,2}{\Gamma', \mathcal{S}.R : \star, \Delta'_i \vdash sig(\mathcal{S}) \cdot \mathcal{S}.R \ni \Lambda\, D.\, \lambda\, \mathcal{S}.\Xi.\, c\ \Delta'_i}}{\Gamma', \mathcal{S}.R : \star \vdash \Pi\, \Delta'_i.\, sig(\mathcal{S}) \cdot \mathcal{S}.R \ni \lambda\, \Delta'_i.\, \Lambda\, D.\, \lambda\, \mathcal{S}.\Xi.\, c\ \Delta'_i}\,1}{\Gamma' \vdash \forall\, \mathcal{S}.R : \star.\, \Pi\, \Delta'_i.\, sig(\mathcal{S}) \cdot \mathcal{S}.R \ni \Lambda\, \mathcal{S}.R.\, \lambda\, \Delta'_i.\, \Lambda\, D.\, \lambda\, \mathcal{S}.\Xi.\, c\ \Delta'}}{\Gamma' \vdash \tau(i) \in \forall\, \mathcal{S}.R : \star.\, \Pi\, \Delta'_i.\, sig(\mathcal{S}) \cdot \mathcal{S}.R}}{\Gamma' \vdash \forall\, \mathcal{S}.R : \star.\, \Pi\, \Delta'_i.\, sig(\mathcal{S}) \cdot \mathcal{S}.R \ni \tau(i)}$$

1. By Proposition B.1 and assumption
2. By Proposition B.1.
3. By Proposition B.5

□

Proof (of Proposition 4.44).

1. By induction on $\#\mathcal{I}.\Sigma$, using weakening.
2. Derive

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma', \mathcal{I}.R : \star \vdash sig(\mathcal{I}) : \star \to \star}\,(a) \quad \dfrac{(\mathcal{I}.R : \star) \in \Gamma', \mathcal{I}.R : \star}{\Gamma', \mathcal{I}.R : \star \vdash \mathcal{I}.R : \star}}{\Gamma', \mathcal{I}.R : \star \vdash sig(\mathcal{I}) \cdot \mathcal{I}.R : \star} \quad \mathcal{J}}{\Gamma', \mathcal{I}.R : \star \vdash \iota\, \mathcal{I}.z : sig(\mathcal{I}) \cdot \mathcal{I}.R.\, \forall\, \mathcal{I}.D : sig(\mathcal{I}) \cdot \mathcal{I}.R \to \star.\, \Pi\, \mathcal{I}.\Sigma.\, \mathcal{I}.D\ \mathcal{I}.z : \star}}{\Gamma' \vdash \lambda\, \mathcal{I}.R : \star.\, \iota\, \mathcal{I}.z : sig(\mathcal{I}) \cdot \mathcal{I}.R.\, \forall\, \mathcal{I}.D : sig(\mathcal{I}) \cdot \mathcal{I}.R \to \star.\, \Pi\, \mathcal{I}.\Sigma.\, \mathcal{I}.D\ \mathcal{I}.z : \star \to \star}$$

where $\mathcal{J}$ is the derivation (let $\Gamma'' =_{\mathbf{df}} \Gamma', \mathcal{I}.R : \star, \mathcal{I}.z : sig(\mathcal{I}) \cdot \mathcal{I}.R, \mathcal{I}.D : sig(\mathcal{I}) \cdot \mathcal{I}.R \to \star$ below)

$$\dfrac{\dfrac{\dfrac{(\mathcal{I}.D : sig(\mathcal{I}) \cdot \mathcal{I}.R \to \star) \in \Gamma'', \mathcal{I}.\Sigma}{\Gamma'', \mathcal{I}.\Sigma \vdash \mathcal{I}.D : sig(\mathcal{I}) \cdot \mathcal{I}.R \to \star} \quad \dfrac{(\mathcal{I}.z : sig(\mathcal{I}) \cdot \mathcal{I}.R) \in \Gamma'', \mathcal{I}.\Sigma}{\Gamma'', \mathcal{I}.\Sigma \vdash \mathcal{I}.z \in sig(\mathcal{I}) \cdot \mathcal{I}.R}}{\Gamma'', \mathcal{I}.\Sigma \vdash sig(\mathcal{I}) \cdot \mathcal{I}.R \ni \mathcal{I}.z}}{\Gamma'', \mathcal{I}.\Sigma \vdash \mathcal{I}.D\ \mathcal{I}.z : \star}}{\Gamma'' \vdash \Pi\, \mathcal{I}.\Sigma.\, \mathcal{I}.D\ \mathcal{I}.z : \star}\,(b)$$

$$\overline{\Gamma', \mathcal{I}.R : \star, \mathcal{I}.z : sig(\mathcal{I}) \cdot \mathcal{I}.R \vdash \forall\, \mathcal{I}.D : sig(\mathcal{I}) \cdot \mathcal{I}.R \to \star.\, \Pi\, \mathcal{I}.\Sigma.\, \mathcal{I}.D\ \mathcal{I}.z}$$

   (a) By Proposition 4.28 and weakening.
   (b) By Proposition 4.44(1.) and weakening.

3. Let $\Gamma'' = \Gamma', \mathcal{I}.R : \star, \mathcal{I}.D : isig(\mathcal{I}) \cdot \mathcal{I}.R \to \star, \mathcal{I}.z : sig(\mathcal{I}) \cdot \mathcal{I}.R, \mathcal{I}.v : View \cdot (isig(\mathcal{I}) \cdot \mathcal{I}.R)\ \beta\{\mathcal{I}.z\}$ and derive

$$\dfrac{\dfrac{\dfrac{(v : \mathit{View} \cdot (\mathit{isig}(\mathcal{I}) \cdot \mathcal{I}.R)\ \beta\{\mathcal{I}.z\}) \in \Gamma''}{\Gamma'' \vdash v \in \mathit{View} \cdot (\mathit{isig}(\mathcal{I}) \cdot \mathcal{I}.R)\ \beta\{\mathcal{I}.z\}}}{\Gamma'' \vdash \mathit{View} \cdot (\mathit{isig}(\mathcal{I}) \cdot \mathcal{I}.R)\ \beta\{\mathcal{I}.z\} \ni v} \quad \dfrac{(\mathcal{I}.D : \mathit{isig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star) \in \Gamma''}{\Gamma'' \vdash \mathcal{I}.D : \mathit{isig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star} \quad \dfrac{\Gamma'' \vdash \mathit{elimView}\ \beta\{\mathcal{I}.z\}\ \text{-}\mathcal{I}.v \in \mathit{isig}(\mathcal{I}) \cdot \mathcal{I}.R}{\Gamma'' \vdash \mathit{isig}(\mathcal{I}) \cdot \mathcal{I}.R \ni \mathit{elimView}\ \beta\{\mathcal{I}.z\}\ \text{-}\mathcal{I}.v}}{\Gamma'' \vdash\ \mathcal{I}.D\ (\mathit{elimView}\ \beta\{\mathcal{I}.z\}\ \text{-}\mathcal{I}.v)\ :\ \star}}{\begin{array}{l}\Gamma', \mathcal{I}.R : \star, \\ \mathcal{I}.D : \mathit{isig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star, \\ \mathcal{I}.z : \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R\end{array} \quad \vdash \quad \begin{array}{l}\forall \mathcal{I}.v : \mathit{View} \cdot (\mathit{isig}(\mathcal{I}) \cdot \mathcal{I}.R)\ \beta\{\mathcal{I}.z\}. \\ \mathcal{I}.D\ (\mathit{elimView}\ \beta\{\mathcal{I}.z\}\ \text{-}\mathcal{I}.v)\end{array} \quad : \quad \star}}{\begin{array}{l}\lambda \mathcal{I}.z : \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R. \\ \Gamma', \mathcal{I}.R : \star, \qquad\quad \forall \mathcal{I}.v : \mathit{View} \cdot (\mathit{isig}(\mathcal{I}) \cdot \mathcal{I}.R)\ \beta\{\mathcal{I}.z\}. \\ \mathcal{I}.D : \mathit{isig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star \ \vdash \quad \mathcal{I}.D\ (\mathit{elimView}\ \beta\{\mathcal{I}.z\}\ \text{-}\mathcal{I}.v) \quad : \quad \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star\end{array}}{\begin{array}{l}\lambda \mathcal{I}.D : \mathit{isig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star. \\ \lambda \mathcal{I}.z : \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R. \\ \qquad \forall \mathcal{I}.v : \mathit{View} \cdot (\mathit{isig}(\mathcal{I}) \cdot \mathcal{I}.R)\ \beta\{\mathcal{I}.z\}. \qquad (\mathit{isig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star) \\ \Gamma', \mathcal{I}.R : \star \vdash \quad \mathcal{I}.D\ (\mathit{elimView}\ \beta\{\mathcal{I}.z\}\ \text{-}\mathcal{I}.v) \quad : \quad \to \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star\end{array}}{\Gamma' \vdash \mathit{lift}(\mathcal{I}) : \Pi\, \mathcal{I}.R : \star.\, (\mathit{isig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star) \to \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star}$$

□

Proof (of Theorem 4.46). First, invert the derivation of $\vdash_{\mathbf{isig}}$.

$$\dfrac{\begin{array}{c}\Gamma \vdash_{\mathbf{con}} \mathit{Decl}[D, \Delta] \searrow \mathcal{C} \quad \{z, v\} \cap (DV(\Gamma) \cup DV(\Delta)) = \emptyset \wedge z \neq v \\ ((c_i : \Pi\, \Delta'_i.\, D) \in \mathcal{C}.\Xi)_{i \in \{1 \ldots \#\mathcal{S}.\Xi\}} \quad \Sigma_{\#\mathcal{C}.\Xi}(i) =_{\mathbf{df}} (c_i : \Pi\, \Delta'_i.\, D\ (\mathcal{C}.\tau(i) \cdot \mathcal{C}.R\ \Delta'_i))\end{array}}{\Gamma \vdash_{\mathbf{isig}} \mathit{Decl}[D, \Delta] \searrow \mathit{record}\ \mathit{ISig}\ \{\mathcal{C} = \mathcal{C}; z = z; \Sigma = \Sigma\}}$$

Now, we derive each of the premises for the judgement we are trying to affirm.

Premise: $\Gamma \vdash_{\mathbf{isig}} \mathit{Decl}[D, \Delta] \searrow \mathcal{I}$

By assumption.

Premise: $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{I}.\mathcal{C})\ \mathit{valid}$

By Theorem 4.38.

Premise: $(\Gamma', \mathcal{I}.R : \star, \mathcal{I}.D : \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star \vdash \mathcal{I}.\Sigma(i))_{i \in \{1 \ldots \#\mathcal{I}.\Sigma\}}$

Pick arbitrary $i \in \{1 \ldots \#\mathcal{I}.\Sigma\}$ (note the upper bound is equal to $\#\mathcal{I}.\Xi$), let $\Gamma'' =_{\mathbf{df}} \Gamma', \mathcal{I}.R : \star, \mathcal{I}.D : \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star, \Delta'_i$, and derive

$$\dfrac{\dfrac{(D : \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star) \in \Gamma''}{\Gamma'' \vdash D : \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star} \quad \mathcal{J} :: \Gamma'' \vdash \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R \ni \mathcal{I}.\tau(i) \cdot \mathcal{I}.R\ \Delta'_i}{\Gamma'' \vdash D\ (\mathcal{I}.\tau(i) \cdot \mathcal{I}.R\ \Delta'_i) : \star}}{\Gamma', \mathcal{I}.R : \star, \mathcal{I}.D : \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R \to \star \vdash \Pi\, \Delta'_i.\, D\ (\mathcal{I}.\tau(i) \cdot \mathcal{I}.R\ \Delta'_i) : \star}\ 1$$

where $\mathcal{J}$ is the derivation

$$\dfrac{\dfrac{\dfrac{}{\Gamma'' \vdash \mathcal{I}.\tau(i) \in \forall\, \mathcal{I}.R : \star.\, \Pi\, \Delta'_i.\, \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R}\ 3 \quad \dfrac{(\mathcal{I}.R : \star) \in \Gamma''}{\Gamma'' \vdash \mathcal{I}.R : \star}}{\Gamma'' \vdash \mathcal{I}.\tau(i) \cdot \mathcal{I}.R \in \Pi\, \Delta'_i.\, \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R} \quad \dfrac{}{\Gamma'' \vdash \Delta'_i : \Delta'_i}\ 2}{\Gamma'' \vdash \mathcal{I}.\tau(i) \cdot \mathcal{I}.R\ \Delta'_i \in \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R}}{\Gamma'' \vdash \mathit{sig}(\mathcal{I}) \cdot \mathcal{I}.R \ni \mathcal{I}.\tau(i) \cdot \mathcal{I}.R\ \Delta'_i}$$

1. By Proposition B.1.
   Note that by assumption $\Gamma' \vdash \Delta'_i$.
2. By Proposition B.5.
3. By the fact that $\mathcal{I}.\mathcal{C}$ is a valid collection of computational constructors for $Decl[D, \Delta]$.

□

Proof (of Theorem 4.52). First, invert the assumed derivation of $\vdash_{\mathbf{isig}}$ and some of its premises.

$$\frac{\Gamma \vdash_{\mathbf{isig}} Decl[D, \Delta] \searrow \mathcal{I} \quad \mathcal{J}_1 :: \Gamma \vdash_{\mathbf{icty}} \mathcal{I} \searrow \Theta \quad \mathcal{J}_2 :: \Gamma \vdash_{\mathbf{ictm}} (\mathcal{I}, \Theta) \searrow \gamma}{\Gamma \vdash_{\mathbf{icon}} Decl[D, \Delta] \searrow \mathsf{record}\ ICon\ \{\mathcal{I} = \mathcal{I}, \Theta = \Theta, \gamma = \gamma\}}$$

where

$$\mathcal{J}_1 :: \frac{\left(\dfrac{(c_i : \Pi\, \Delta'_i.\ D) \in \mathcal{I}.\Xi}{\Gamma \vdash_{\mathbf{icty}} (\mathcal{I}, i) \searrow (c_i, \forall\, \mathcal{I}.R : \star.\ \Pi\, \Delta'_i.\ isig(\mathcal{I}) \cdot \mathcal{I}.R)}\right)_{i \in \{1 \ldots \#\mathcal{I}.\Xi\}}}{\Gamma \vdash_{\mathbf{icty}} \mathcal{I} \searrow \Theta_{\#\mathcal{I}.\Sigma}(i) =_{\mathbf{df}} (c_i : \forall\, \mathcal{I}.R : \star.\ \Pi\, \Delta'_i.\ isig(\mathcal{I}) \cdot \mathcal{I}.R)}$$

and where

$$\mathcal{J}_2 :: \frac{\left(\dfrac{(c_i : T_i) \in \Theta \quad (c_i : \Pi\, \Delta'_i.\ D) \in \mathcal{I}.\Xi}{\Gamma \vdash_{\mathbf{ictm}} (\mathcal{I}, \Theta, i) \searrow \chi\ T_i\ \text{-}\ \Lambda\, \mathcal{I}.R.\ \lambda\, \Delta'_i.\ [\mathcal{I}.\tau(i) \cdot \mathcal{I}.R\ \Delta'_i, \Lambda\, \mathcal{I}.D.\ \lambda\, \mathcal{I}.\Sigma.\ c_i\ \Delta'_i]}\right)_{i \in \{1 \ldots \#\Theta\}}}{\Gamma \vdash_{\mathbf{ictm}} (\mathcal{I}, \Theta) \searrow \gamma_{\#\mathcal{I}.\Theta}(i) =_{\mathbf{df}} \chi\ T_i\ \text{-}\ \Lambda\, R.\ \lambda\, \Delta'_i.\ [\mathcal{I}.\tau(i) \cdot \mathcal{I}.R\ \Delta'_i, \Lambda\, D.\ \lambda\, \mathcal{I}.\Sigma.\ c_i\ \Delta'_i]}$$

Now, we consider each of the premises of the single inference rule for the judgement we are trying to affirm.

Premise: $\Gamma \vdash Decl[D, \Delta] \searrow \mathcal{R}$
By assumption.

Premise: $\Gamma \vdash Decl[D, \Delta] \searrow (\Gamma', \mathcal{R}.\mathcal{I})\ valid$
By Theorem 4.46.

Premise: $(\Gamma' \vdash \mathcal{R}.\Theta(i))_{i \in \{1 \ldots \#\mathcal{R}.\Theta\}}$
Pick arbitrary $i \in \{1 \ldots \#\mathcal{R}.\Theta\}$ and derive

$$\dfrac{\dfrac{\dfrac{}{\Gamma', \mathcal{I}.R : \star, \Delta'_i \vdash isig(\mathcal{I}) : \star \to \star}\ 2 \quad \dfrac{(\mathcal{I}.R : \star) \in \Gamma', \mathcal{I}.R : \star, \Delta'_i}{\Gamma', \mathcal{I}.R : \star, \Delta'_i \vdash \mathcal{I}.R : \star}}{\Gamma', \mathcal{I}.R : \star, \Delta'_i \vdash isig(\mathcal{I}) \cdot \mathcal{I}.R : \star}}{\dfrac{\Gamma', \mathcal{I}.R : \star \vdash \Pi\, \Delta'_i.\ isig(\mathcal{I}) \cdot \mathcal{I}.R : \star}{\Gamma' \vdash \forall\, \mathcal{I}.R : \star.\ \Pi\, \Delta'_i.\ isig(\mathcal{I}) \cdot \mathcal{I}.R : \star}}\ 1$$

1. By Proposition B.1.
2. By Proposition 4.44 and weakening.

Premise: $\#\mathcal{R}.\Theta = \#\mathcal{R}.\gamma$

Immediate (both are equal to $\#\mathcal{R}.\Xi$ and thus $\#\Delta$).

Premise: $(\Gamma' \vdash \mathcal{R}.\gamma(i) : \mathcal{R}.\Theta(i))_{i \in \{1 \ldots \#\mathcal{R}.\Theta\}}$

Pick arbitrary $i \in \{1 \ldots \#\mathcal{R}.\Theta\}$ and derive (where $T_i = \forall\, \mathcal{I}.R : \star.\, \Pi\, \Delta'_i.\, isig(\mathcal{I}) \cdot \mathcal{I}.R$)

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{J}_1 \quad \mathcal{J}_2 \quad \overline{|\mathcal{I}.\tau \cdot \mathcal{I}.R\ \Delta'_i| =_{\beta\eta} |\Lambda\, \mathcal{I}.D.\, \lambda\, \mathcal{I}.\Sigma.\, c_i\ \Delta'_i|}^{\,1}}{\Gamma', \mathcal{I}.R : \star, \Delta'_i \vdash isig(\mathcal{I}) \cdot \mathcal{I}.R \ni [\mathcal{I}.\tau(i) \cdot \mathcal{I}.R\ \Delta'_i, \Lambda\, \mathcal{I}.D.\, \lambda\, \mathcal{I}.\Sigma.\, c_i\ \Delta'_i]}}{\Gamma', \mathcal{I}.R : \star \vdash \Pi\, \Delta'_i.\, isig(\mathcal{I}) \cdot \mathcal{I}.R \ni \lambda\, \Delta'_i.\, [\mathcal{I}.\tau(i) \cdot \mathcal{I}.R\ \Delta'_i, \Lambda\, \mathcal{I}.D.\, \lambda\, \mathcal{I}.\Sigma.\, c_i\ \Delta'_i]}}{\Gamma' \vdash T_i \ni \Lambda\, \mathcal{I}.R.\, \lambda\, \Delta'_i.\, [\mathcal{I}.\tau(i) \cdot \mathcal{I}.R\ \Delta'_i, \Lambda\, \mathcal{I}.D.\, \lambda\, \mathcal{I}.\Sigma.\, c_i\ \Delta'_i]}}{\Gamma' \vdash \gamma(i) \in T_i}}{\Gamma' \vdash T_i \ni \gamma(i)}$$

where let $\Gamma'' =_{\mathbf{df}} \Gamma', \mathcal{I}.R : \star, \Delta'_i$ in

$$\mathcal{J}_1 :: \dfrac{\dfrac{\overline{\Gamma'' \vdash \mathcal{I}.\tau(i) \in \forall\, \mathcal{I}.R : \star.\, \Pi\, \Delta'_I.\, sig(\mathcal{I}) \cdot \mathcal{I}.R}^{\,3}}{\Gamma'' \vdash \mathcal{I}.\tau(i) \cdot \mathcal{I}.R \in \Pi\, \Delta'_I.\, sig(\mathcal{I}) \cdot \mathcal{I}.R} \quad \overline{\Gamma'' \vdash \Delta'_i : \Delta'_i}^{\,2}}{\Gamma'' \vdash \mathcal{I}.\tau(i) \cdot \mathcal{I}.R\ \Delta'_i \in sig(\mathcal{I}) \cdot \mathcal{I}.R}$$

and let $\Gamma''' =_{\mathbf{df}} \Gamma'', \mathcal{I}.D : sig(\mathcal{I}) \cdot \mathcal{I}.R \to \star$ in

$$\mathcal{J}_2 :: \dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma''', \mathcal{I}.\Sigma \vdash c_i \in \Pi\, \Delta'_i.\, D\ (\mathcal{I}.\tau(i) \cdot \mathcal{I}.R\ \Delta'_i)}^{\,4} \quad \overline{\Gamma''', \mathcal{I}.\Sigma \vdash \Delta'_i : \Delta'_I}^{\,5}}{\Gamma''', \mathcal{I}.\Sigma \vdash c_i\ \Delta_i \in \mathcal{I}.D\ (\mathcal{I}.\tau(i) \cdot \mathcal{I}.R\ \Delta'_i)}}{\Gamma''', \mathcal{I}.\Sigma \vdash \mathcal{I}.D\ (\mathcal{I}.\tau(i) \cdot \mathcal{I}.R\ \Delta'_i) \ni c_i\ \Delta'_i}}{\Gamma''' \vdash \Pi\, \mathcal{I}.\Sigma.\, \mathcal{I}.D\ (\mathcal{I}.\tau(i) \cdot \mathcal{I}.R\ \Delta'_i) \ni \lambda\, \mathcal{I}.\Sigma.\, c_i\ \Delta'_i}}{\Gamma'' \vdash \forall\, \mathcal{I}.D : sig(\mathcal{I}) \cdot \mathcal{I}.R \to \star.\, \Pi\, \mathcal{I}.\Sigma.\, \mathcal{I}.D\ (\mathcal{I}.\tau(i) \cdot \mathcal{I}.R\ \Delta'_i) \ni \Lambda\, \mathcal{I}.D.\, \lambda\, \mathcal{I}.\Sigma.\, c_i\ \Delta'_i}$$

1. By validity of $\mathcal{C}$ as the computational constructors for $Decl[D, \Delta]$, $|\mathcal{C}.\tau(i)| = \lambda\, \Delta.\, |c_i\ \Delta'_i|$. By Fact B.8, $Seq(\Delta) = Seq(\mathcal{I}.\Delta)$, so by Proposition B.6 $\lambda\, \Delta.\, |c_i\ \Delta'_i| = \lambda\, \mathcal{I}.\Sigma.\, |c_i\ \Delta_i|$.
2. By Proposition B.5
3. By validity of $\mathcal{C}$ as the computational constructors for $Decl[D, \Delta]$ and Fact B.8.
4. By Theorem 4.46 and inversion of the derivation of $\vdash_{\mathbf{isig}}$.
5. By Proposition B.5.

□

B.2 Datatype Signature Dependent Eliminator

Proof (of Theorem 4.56).

- **Typing law**

  Because this is a large derivation, we make some abbreviations for terms and derivations.

  Below, let $\Gamma'' =_{\mathbf{df}} \Gamma', \mathcal{R}.R : \star, \mathcal{R}.D : isig(\mathcal{R}) \cdot \mathcal{R}.R \to \star, Case(\mathcal{R}), \mathcal{R}.z : isig(\mathcal{R}) \cdot \mathcal{R}.R$.

– *View evidence*

Let $t_{\mathrm{v}} =_{\mathbf{df}} \mathit{intrView} \cdot (\mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R)\ \beta\{z\}\ \text{-}z\ \text{-}\beta$, and the derivation $\mathcal{J}_{\mathrm{v}}$ be

$$\mathcal{J}_{\mathrm{v}} :: \dfrac{\dfrac{\dfrac{}{\Gamma'' \vdash \mathit{isig}(\mathcal{R}) : \star \to \star}\,1 \quad \Gamma'' \vdash \mathcal{R}.R : \star}{\Gamma'' \vdash \mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R : \star} \quad \begin{array}{c}\Gamma'' \vdash \mathit{Top} \ni \beta\{z\} \\ \Gamma'' \vdash \mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R \ni z\end{array}}{\Gamma'' \vdash \{z \simeq z.1\} \ni \beta} \quad \Gamma'' \vdash t_{\mathrm{v}} \in \mathit{View} \cdot (\mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R)\ \beta\{z.1\}$$

– *Case branch types*

Let $i \in \{1 \ldots \#\Sigma\}$, let $t_{c_i} =_{\mathbf{df}} \mathcal{R}.\tau(i) \cdot \mathcal{R}.R\ \Delta'_i$, let $t'_{c_i} =_{\mathbf{df}} \mathcal{R}.\gamma(i) \cdot \mathcal{R}.R\ \Delta'_i$, let $\Gamma''' =_{\mathbf{df}} \Gamma'', \Delta'_i, \mathcal{R}.v : \mathit{View} \cdot (\mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R)\ \beta\{t_{c_i}\}$, and let $\mathcal{J}_{\mathrm{b},i}$ be

$$\mathcal{J}_{\mathrm{b},i} :: \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma''' \vdash c_i \in \Pi\,\Delta'_i.\,\mathcal{R}.D\ t'_{c_i}}\,2 \quad \dfrac{}{\Gamma''' \vdash \Delta_i : \Delta_i}\,3}{\Gamma''' \vdash c_i\ \Delta'_i \in \mathcal{R}.D\ t'_{c_i}} \quad \dfrac{|t'_{c_i}| =_{\beta\eta} |t_{c_i}|}{\mathcal{R}.D\ t'_{c_i} \cong \mathcal{R}.D\ t_{c_i}}\,4}{\Gamma''' \vdash \mathcal{R}.D\ (\mathit{elimView}\ \beta\{t_{c_i}\}\ \text{-}v) \ni c_i\ \Delta_i}}{\Gamma'', \Delta'_i \vdash \mathit{lift}(\mathcal{R}) \cdot \mathcal{R}.R \cdot \mathcal{R}.D\ t_{c_i} \ni \Lambda\,\mathcal{R}.v.\,c_i\ \Delta'_i}}{\Gamma'' \vdash \Pi\,\Delta'_i.\,\mathit{lift}(\mathcal{R}) \cdot \mathcal{R}.R \cdot \mathcal{R}.D\ t_{c_i} \ni \lambda\,\Delta'_i.\,\Lambda\,\mathcal{R}.v.\,c_i\ \Delta'_i}$$

We then obtain the derivation

$$\mathcal{J}_{\mathbf{b}} :: \Gamma'' \vdash \mathit{branches}(\mathcal{R}) : \mathcal{R}.\Sigma[\mathit{lift}(\mathcal{R}) \cdot \mathcal{R}.R \cdot \mathcal{R}.D/\mathcal{R}.D]$$

by weakening and the telescope formation rules.

For the above derivations to work, we need that Γ'' is a well-formed context. We already know from Proposition 4.44 and type constructor application that $\Gamma', \mathcal{R}.R : \star \vdash \mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R : \star$, so we just check well-formedness of the entries of $\mathit{Case}(\mathcal{R})$. Pick $i \in \{1 \ldots \#\mathit{Case}(\mathcal{R})\}$, redefine $\Gamma''' =_{\mathbf{df}} \Gamma', \mathcal{R}.R : \star, \mathcal{R}.D : \mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R \to \star$, and derive

*

$$\mathcal{J}_{t'_{c_i}} :: \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma'', \Delta'_i \vdash \mathcal{R}.\gamma(i) \in \forall\,\mathcal{R}.R : \star.\,\Pi\,\Delta'_i.\,\mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R}\,5}{\Gamma'', \Delta'_i \vdash \mathcal{R}.\gamma(i) \cdot \mathcal{R}.R \in \Pi\,\Delta'_i.\,\mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R} \quad \dfrac{}{\Gamma'', \Delta'_i \vdash \Delta'_i : \Delta'_i}}{\Gamma'', \Delta'_i \vdash t'_{c_i} \in \mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R}}{\Gamma'', \Delta'_i \vdash \mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R \ni t'_{c_i}}$$

*

$$\mathcal{J}_{\mathrm{C}_i} :: \dfrac{\Gamma'', \Delta'_i \vdash \mathcal{R}.D : \mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R \to \star \quad \mathcal{J}_{t_{c_i}} :: \Gamma'', \Delta'_i \vdash \mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R \ni t'_{c_i}}{\dfrac{\Gamma'', \Delta'_i \vdash \mathcal{R}.D\ (t'_{c_i}) : \star}{\Gamma'' \vdash \mathit{Case}(\mathcal{R})(i) : \star}}$$

Now, to conclude it suffices to give the following typing derivation for $\mathit{case}(\mathcal{R})$.

$$
\begin{array}{c}
\begin{array}{c}
\Gamma'' \vdash z \in \mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R \\
\hline
\Gamma \vdash z.2 \in \forall\, \mathcal{R}.D : \mathit{sig}(\mathcal{R}) \cdot \mathcal{R}.R \to \star. \\
\quad \Pi\, \mathcal{R}.\Sigma.\, \mathcal{R}.D \ z.1
\end{array}
\qquad
\begin{array}{c}
\\
\hline
\Gamma'' \vdash \mathit{lift}(\mathcal{R}) \cdot \mathcal{R}.R \cdot \mathcal{R}.D \\
\quad : \mathit{sig}(\mathcal{R}) \cdot \mathcal{R}.R \to \star
\end{array}
\ 7 \\
\hline
\Gamma'' \vdash z.2 \cdot (\mathit{lift}(\mathcal{R}) \cdot \mathcal{R}.R \cdot \mathcal{R}.D) \\
\quad \in \Pi\, \mathcal{R}.\Sigma[\mathit{lift}(\mathcal{R}) \cdot \mathcal{R}.R \cdot \mathcal{R}.\Delta / D].\, \mathit{lift}(\mathcal{R}) \cdot \mathcal{R}.R \cdot \mathcal{R}.D \ z.1 \qquad \mathcal{J}_{\mathbf{b}} \\
\hline
\Gamma'' \vdash \ z.2 \cdot (\mathit{lift}(\mathcal{R}) \cdot \mathcal{R}.R \cdot \mathcal{R}.D) \ \mathit{branches}(\mathcal{R}) \\
\quad \in \mathit{lift}(\mathcal{R}) \cdot \mathcal{R}.R \cdot \mathcal{R}.D \ z.1 \qquad \mathcal{J}_{\mathrm{v}} \\
\hline
\Gamma'' \vdash \ z.2 \cdot (\mathit{lift}(\mathcal{R}) \cdot \mathcal{R}.R \cdot \mathcal{R}.D) \ \mathit{branches}(\mathcal{R}) \text{ -}t_{\mathrm{v}} \\
\quad \in \mathcal{R}.D \ (\mathit{elimView} \ \beta\{z.1\} \text{ -}z \text{ -}\ldots) \\
\hline
\Gamma'' \vdash \ \mathcal{R}.D \ z \\
\quad \ni z.2 \cdot (\mathit{lift}(\mathcal{R}) \cdot \mathcal{R}.R \cdot \mathcal{R}.D) \ \mathit{branches}(\mathcal{R}) \text{ -}t_{\mathrm{v}} \\
\hline
\Gamma', \mathcal{R}.R : \star, \mathcal{R}.D : \mathit{isig}(\mathcal{R} \cdot \mathcal{R}.R \to \star), \mathit{Case}(\mathcal{R}) \vdash \\
\quad \Pi\, \mathcal{R}.z : \mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R.\, \mathcal{R}.D \ z \\
\quad \ni \lambda\, \mathcal{R}.z. \\
\qquad z.2 \cdot (\mathit{lift}(\mathcal{R}) \cdot \mathcal{R}.R \cdot \mathcal{R}.D) \ \mathit{branches}(\mathcal{R}) \text{ -}t_{\mathrm{v}} \\
\hline
\Gamma', \mathcal{R}.R : \star, \mathcal{R}.D : \mathit{isig}(\mathcal{R} \cdot \mathcal{R}.R \to \star) \vdash \qquad 6 \\
\quad \Pi\, \mathit{Case}(\mathcal{R}).\, \Pi\, \mathcal{R}.z : \mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R.\, \mathcal{R}.D \ z \\
\quad \ni \lambda\, \mathcal{R}.\Sigma.\, \lambda\, \mathcal{R}.z. \\
\qquad z.2 \cdot (\mathit{lift}(\mathcal{R}) \cdot \mathcal{R}.R \cdot \mathcal{R}.D) \ \mathit{branches}(\mathcal{R}) \text{ -}t_{\mathrm{v}} \\
\hline
\Gamma', \mathcal{R}.R : \star \vdash \\
\quad \forall\, \mathcal{R}.D : \mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R \to \star.\, \Pi\, \mathit{Case}(\mathcal{R}).\, \Pi\, \mathcal{R}.z : \mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R.\, \mathcal{R}.D \ z \\
\quad \ni \mathcal{R}.D\, \lambda\, \mathcal{R}.\Sigma.\, \lambda\, \mathcal{R}.z. \\
\qquad z.2 \cdot (\mathit{lift}(\mathcal{R}) \cdot \mathcal{R}.R \cdot \mathcal{R}.D) \ \mathit{branches}(\mathcal{R}) \text{ -}t_{\mathrm{v}}. \\
\hline
\Gamma' \vdash \forall\, \mathcal{R}.R : \star.\, \forall\, \mathcal{R}.D : \mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R \to \star.\, \Pi\, \mathit{Case}(\mathcal{R}).\, \Pi\, \mathcal{R}.z : \mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R.\, \mathcal{R}.D \ z \\
\quad \ni \Lambda\, \mathcal{R}.R.\, \mathcal{R}.D\, \lambda\, \mathcal{R}.\Sigma.\, \lambda\, \mathcal{R}.z. \\
\qquad z.2 \cdot (\mathit{lift}(\mathcal{R}) \cdot \mathcal{R}.R \cdot \mathcal{R}.D) \ \mathit{branches}(\mathcal{R}) \text{ -}t_{\mathrm{v}}.
\end{array}
$$

1. By Proposition 4.44 and weakening.
2. Recall the definition of $\mathit{Case}(\mathcal{R})$, Definition 4.55.
3. By Proposition B.5.
4. By Fact 4.51.
5. By assumption (see Definition 4.51)
6. By derivations $\mathcal{J}_{\mathrm{C}_i}$ and $\mathcal{J}_{t_{c_i}}$ above ($1 \leq i \leq \#\Delta$) and Proposition B.1.
7. We have $\Gamma'' \vdash \mathit{lift}(\mathcal{R}) : \Pi\, \mathcal{R}.R : \star.\, (\mathit{isig}(\mathcal{R}) \cdot \mathcal{R}.R \to \star) \to \mathit{sig}(\mathcal{R}) \cdot \mathcal{R}.R \to \star$ by Proposition 4.44, the rest is by type constructor application.

- **Computation law**

  Pick $i \in \{1 \ldots \#\Delta\}$, and let $(c_i : \Pi\, \Delta'_i.\, D) \in \mathcal{R}.\Xi$. Calculate

$$
\begin{array}{rl}
 & |case(\mathcal{R}) \cdot S \cdot T\ \tau\ (\mathcal{R}.\gamma(i)) \cdot S\ \Delta_i'| \\
= & (\lambda\, \mathcal{R}.\Sigma.\, \lambda\, \mathcal{R}.z.\, z\ |branches(\mathcal{R})|)\ |\tau|\ ((\lambda\, |\Delta_i'|.\, \lambda\, \mathcal{R}.\Sigma.\, c_i\ |\Delta_i'|)\ |\Delta_i'|) \\
\overset{\#\mathcal{R}.\Sigma}{\dashrightarrow} & (\lambda\, \mathcal{R}.z.\, z\ |branches(\mathcal{R})[|\tau|/\mathcal{R}.\Sigma]|)\ ((\lambda\, |\Delta_i'|.\, \lambda\, \mathcal{R}.\Sigma.\, c_i\ |\Delta_i'|)\ |\Delta_i'|) \\
\dashrightarrow & (\lambda\, |\Delta_i'|.\, \lambda\, \mathcal{R}.\Sigma.\, c_i\ |\Delta_i'|)\ |\Delta_i'|\ |branches(\mathcal{R})[|\tau|/\mathcal{R}.\Sigma]| \\
\overset{\#|\Delta_i|}{\dashrightarrow} & (\lambda\, \mathcal{R}.\Sigma.\, c_i\ |\Delta_i'|)\ |branches(\mathcal{R})[|\tau|/\mathcal{R}.\Sigma]| \\
\overset{\#\mathcal{R}.\Sigma}{\dashrightarrow} & |branches(\mathcal{R})(i)|[|\tau|/\mathcal{R}.\Sigma]\ |\Delta_i'| \\
= & (\lambda\, |\Delta_i'|.\, c_i\ |\Delta_i'|)[|\tau|/\mathcal{R}.\Sigma]\ |\Delta_i'| \\
= & (\lambda\, |\Delta_i'|.\, |\tau(i)|\ |\Delta_i'|)\ |\Delta_i'| \\
\overset{\#|\Delta_i|}{\dashrightarrow} & |\tau(i)|\ |\Delta_i'|
\end{array}
$$

Observe that $\#\mathcal{R}.\Sigma = \#\Delta$ by the assumption that $\mathcal{R}$ is valid for datatype declaration $Decl[D, \Delta]$.

□

APPENDIX C
DATATYPE ELABORATION PROOFS

Proof (of Theorem 5.10). The judgment we are trying to derive must be formed from a use of the following inference rule.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathbf{data}} \mathit{Decl}[D,\Delta] \searrow \mathcal{D} \quad \Gamma \vdash \mathit{Decl}[D,\Delta] \searrow (\Gamma', \mathcal{D}.R)\ \mathit{valid} \\ \Gamma' \vdash \mathcal{D}.D : \star \quad \#\mathcal{D}.\Delta = \#\mathcal{D}.\theta \quad (\Gamma' \vdash \mathcal{D}.\theta(i) : \mathcal{D}.\Delta(i))_{i \in \{1 \ldots \#\Delta\}}\end{array}}{\Gamma \vdash \mathit{Decl}[D,\Delta] \searrow (\Gamma', \mathcal{D})\ \mathit{valid}}$$

By assumption, we have a derivation of $\Gamma \vdash \mathit{Decl}[D,\Delta] \searrow \mathcal{D}$, so use inversion to obtain that it must be formed from the following rules.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathbf{icon}} \mathit{Decl}[D,\Delta] \searrow \mathcal{R} \quad \Gamma \vdash +\mathit{isig}(\mathcal{R}) \searrow t \\ (\Gamma \vdash_{\mathbf{dcon}} (\mathcal{R}, i) \searrow (c_i, T_i))_{i \in \{1 \ldots n\}} \quad (\Gamma \vdash_{\mathbf{dcon}} (\mathcal{R}, t, i) \searrow t_i)_{i \in 1 \ldots \#\Delta}\end{array}}{\Gamma \vdash_{\mathbf{data}} \mathit{Decl}[D,\Delta] \searrow \mathsf{record}\ \mathit{Data}\ \begin{array}{l}\{\mathcal{R} = \mathcal{R}; \mathit{mono} = t; D = \mu(\mathit{isig}(\mathcal{R})) \\ ; \Delta_{\#\Delta}(i) = (c_i : T_i); \theta_{\#\Delta}(i) = (t_i)\}\end{array}}$$

$$\frac{(c_i : \Pi\, \Delta_i.\ D) \in \mathcal{R}.\Xi}{\Gamma \vdash_{\mathbf{dcon}} (\mathcal{R}, i) \searrow (c_i, \Pi\, \Delta_i[\mu(\mathit{isig}(\mathcal{R}))/\mathcal{R}.D].\ \mu(\mathit{isig}(\mathcal{R})))}$$

$$\frac{(c_i : \Pi\, \Delta_i.\ D) \in \mathcal{R}.\Xi}{\Gamma \vdash_{\mathbf{dcon}} (\mathcal{R}, t, i) \searrow \lambda\, \Delta_i.\ \mathit{in} \cdot \mathit{isig}(\mathcal{R})\ \text{-}t\ (\mathcal{R}.\gamma(i) \cdot \mu(\mathit{isig}(\mathcal{R}))\ \Delta_i)}$$

Now, let us consider the premises of the rule we are trying to apply.

- $\Gamma \vdash_{\mathbf{data}} \mathit{Decl}[D,\Delta] \searrow \mathcal{D}$
  Given by assumption.
- $\Gamma \vdash \mathit{Decl}[D,\Delta] \searrow (\Gamma', \mathcal{D}.\mathcal{R})\ \mathit{valid}$
  We note that $\mathcal{D}.\mathcal{R} = \mathcal{R}$, and we have a derivation of $\Gamma \vdash_{\mathbf{icon}} \mathit{Decl}[D,\Delta] \searrow \mathcal{R}$. We thus obtain a derivation of the desired judgment by Theorem 4.52.
- $\Gamma' \vdash \mathcal{D}.D : \star/$
  We note that $\mathcal{D}.D = \mu(\mathit{isig}(\mathcal{R}))$. By Proposition 4.44, $\Gamma' \vdash \mathit{isig}(\mathcal{R}) : \star \to \star$, so by the formation rule of μ (Figure 5.6), $\Gamma' \vdash \mu(\mathit{isig}(\mathcal{R})) : \star$.
- $\#\mathcal{D}.\Delta = \#\mathcal{D}.\theta$
  Immediate, both are equal to $\#\Delta$.
- $(\Gamma' \vdash \mathcal{D}.\theta(i) : \mathcal{D}.\Delta(i))_{i \in \{1 \ldots \#\Delta\}}$
  Pick arbitrary $i \in \{1 \ldots \#\mathcal{D}.\Delta\}$, and this reduces to showing the following typing derivation (where $(c_i : \Pi\, \Delta_i.\ D \in \mathcal{R}.\Xi)$). Below, let $\Gamma'' =_{\mathbf{df}} \Gamma', \Delta_i[\mu(\mathit{isig}(\mathcal{R}))/\mathcal{R}.D]$.

$$\dfrac{\dfrac{\dfrac{\Gamma'' \vdash \mathit{isig}(\mathcal{R}) : \star \to \star \quad \Gamma \vdash \mathit{Mono} \cdot \mathit{isig}(\mathcal{R}) \ni t \quad \mathcal{J}_1}{\Gamma'' \vdash \mathit{in} \cdot \mathit{isig}(\mathcal{R})\ \text{-}t\ (\mathcal{R}.\gamma(i) \cdot \mu(\mathit{isig}(\mathcal{R}))\ \Delta_i) \in \mu(\mathit{isig}(\mathcal{R}))}}{\Gamma'' \vdash \mu(\mathit{isig}(\mathcal{R})) \ni \mathit{in} \cdot \mathit{isig}(\mathcal{R})\ \text{-}t\ (\mathcal{R}.\gamma(i) \cdot \mu(\mathit{isig}(\mathcal{R}))\ \Delta_i)}}{\Gamma' \vdash \Pi\, \Delta_i[\mu(\mathit{isig}(\mathcal{R}))/\mathcal{R}.D].\ \mu(\mathit{isig}(\mathcal{R})) \ni \lambda\, \Delta_i.\ \mathit{in} \cdot \mathit{isig}(\mathcal{R})\ \text{-}t\ (\mathcal{R}.\gamma(i) \cdot \mu(\mathit{isig}(\mathcal{R}))\ \Delta_i)}\ 1$$

where $\mathcal{J}_1$ is the derivation

$$\dfrac{\dfrac{\dfrac{}{\Gamma'' \vdash \mathcal{R}.\gamma(i) \in \forall\, \mathcal{R}.R{:}\star.\, \Pi\, \Delta_i.\, isig(\mathcal{R}) \cdot \mathcal{R}.R}\,{\scriptstyle 2} \quad \dfrac{}{\Gamma'' \vdash \mu(isig(\mathcal{R})) : \star}\,{\scriptstyle 3}}{\Gamma'' \vdash \mathcal{R}.\gamma(i) \cdot \mu(isig(\mathcal{R})) \in \Pi\, \Delta_i[\mu(isig(\mathcal{R}))/\mathcal{R}.R].\, isig(\mathcal{R}) \cdot \mu(isig(\mathcal{R}))} \quad \dfrac{}{\Gamma'' \vdash \Delta_i : \Delta_i}\,{\scriptstyle 4}}{\dfrac{\Gamma'' \vdash \mathcal{R}.\gamma(i) \cdot \mu(isig(\mathcal{R}))\ \Delta_i \in isig(\mathcal{R}) \cdot \mu(isig(\mathcal{R}))}{\Gamma'' \vdash isig(\mathcal{R}) \cdot \mu(isig(\mathcal{R})) \ni \mathcal{R}.\gamma(i) \cdot \mu(isig(\mathcal{R}))\ \Delta_i}}$$

1. By Proposition B.1.
   From $\Gamma \vdash Decl[D, \Delta] \searrow (\Gamma', \mathcal{R})$ *valid*, we readily obtain $\Gamma' \vdash \Delta_i$
2. By Fact 4.51.
3. By Proposition 4.44, the μ-formation rule, and weakening.
4. By Lemma B.5.

□

Proof (of Theorem 5.12). First, we invoke Proposition 4.54 to obtain a decision for the judgment $\Gamma \vdash_{\mathbf{icon}} Decl[D, \Delta] \searrow \mathcal{R}$ for some $\mathcal{R}$; we affirm the goal only if this decision is an affirmation.

For the monotonicity witness, observe that by inversion we have that for every $(c_i : \Pi\, \Delta_i.\, D) \in \Delta$, $\Gamma, \mathcal{R}.R : \star \vdash \Delta_i[\mathcal{R}.R/D] \searrow \Delta_i'$. By assumption, each Δ_i' has the property that its classifiers are terminating. It suffices to show that this implies $isig(\mathcal{R})$ is terminating as well, as the remaining premises are constructable.

$$isig(\mathcal{R}) = \lambda\, \mathcal{R}.R{:}\star.\, \iota\, \mathcal{R}.z : sig(\mathcal{R}) \cdot \mathcal{R}.R.\, \forall\, \mathcal{R}.D : sig(\mathcal{R}) \cdot \mathcal{R}.R \to \star.\, \Pi\, \mathcal{R}.\Sigma.\, \mathcal{R}.D\ \mathcal{R}.z$$

- Our first step is to show $sig(\mathcal{R}) \cdot \mathcal{R}.R$ is terminating. By one reduction (we substitute a variable, so it does not matter whether we reduce this first or first reduce the body of $isig(\mathcal{R})$).

  $$sig(\mathcal{R}) \cdot \mathcal{R}.R \dashrightarrow \forall\, \mathcal{R}.D{:}\star.\, \Pi\, \mathcal{R}.\Xi.\, \mathcal{R}.D$$

  and it suffices to show that each of the classifiers of $\mathcal{R}.\Xi$ are terminating. Pick arbitrary $i \in \{1 \ldots \#\mathcal{R}.\Xi\}$, and by inversion

  $$\mathcal{R}.\Xi(i) = (c_i : \Pi\, \Delta_i'.\, D)$$

  which is terminating because Δ_i' is.
- Next, we need that each classifier of $\mathcal{R}.\Sigma$ is terminating. Pick arbitrary $i \in \{1 \ldots \#\mathcal{R}.\Sigma\}$, and by inversion

  $$\mathcal{R}.\Sigma(i) = (c_i : \Pi\, \Delta_i'.\, D\ (\mathcal{R}.\tau(i) \cdot \mathcal{R}.R\ \Delta_i'))$$

  Since Δ_i' is terminating, it remains to show that $\mathcal{R}.\tau(i) \cdot \mathcal{R}.R\ \Delta_i'$ is terminating. By inversion,

  $$|\mathcal{R}.\tau(i)| = \lambda\, |\Delta_i'|.\, \lambda\, \mathcal{R}.\Xi.\, c_i\ |\Delta_i'|$$

  and it is clear then that the following term is terminating.

  $$(\lambda\, |\Delta_i'|.\, \lambda\, \mathcal{R}.\Xi.\, c_i\ |\Delta_i'|)\ |\Delta_i'|$$

□

APPENDIX D
DERIVATION OF COURSE-OF-VALUES INDUCTION IN CDLE

D.1 Implicity Restricted Existentials

```
RExtC : (⋆ → ⋆) → (⋆ → ⋆) → ⋆
= λ H: ⋆ → ⋆. λ F: ⋆ → ⋆.
  ∀ X: ⋆. (∀ R: ⋆. H ·R ⇒ F ·R → X) → X .

packC : ∀ H: ⋆ → ⋆. ∀ F: ⋆ → ⋆. ∀ R: ⋆. H ·R ⇒ F ·R → RExtC ·H ·F
= Λ H. Λ F. Λ R. Λ r. λ xs. Λ X. λ f. f -r xs .

unpackC
: ∀ H: ⋆ → ⋆. ∀ F: ⋆ → ⋆.
  ∀ X: ⋆. (∀ R: ⋆. H ·R ⇒ F ·R → X) → RExtC ·H ·F → X
= Λ H. Λ F. Λ X. λ f. λ x. x f .

RExtI : Π H: ⋆ → ⋆. Π F: ⋆ → ⋆. RExtC ·H ·F → ⋆
= λ H: ⋆ → ⋆. λ F: ⋆ → ⋆. λ x: RExtC ·H ·F.
  ∀ P: RExtC ·H ·F → ⋆.
  (∀ R: ⋆. ∀ r: H ·R. Π xs: F ·R. P (packC -r xs)) → P x .

RExt : (⋆ → ⋆) → (⋆ → ⋆) → ⋆
= λ H: ⋆ → ⋆. λ F: ⋆ → ⋆. ι x: RExtC ·H ·F. RExtI ·H ·F x .

pack : ∀ H: ⋆ → ⋆. ∀ F: ⋆ → ⋆. ∀ R: ⋆. H ·R ⇒ F ·R → RExt ·H ·F
= Λ H. Λ F. Λ R. Λ r. λ xs.
  [ packC -r xs , Λ P. λ f. f -r xs ] .

_ : { pack ≃ λ xs. λ f. f xs } = β .

unpack : ∀ H: ⋆ → ⋆. ∀ F: ⋆ → ⋆. ∀ P: RExt ·H ·F → ⋆.
  (∀ R: ⋆. ∀ r: H ·R. Π xs: F ·R. P (pack -r xs)) → Π x: RExt ·H ·F. P x
= Λ H. Λ F. Λ P. λ f. λ x.
  x.2 ·(λ x: RExtC ·H ·F. ∀ y: RExt ·H ·F. ∀ eq: { y ≃ x }. P (φ eq - y {x}))
    (Λ R. Λ r. λ xs. Λ y. Λ eq. f -r xs)
    -x -β .

_ : { unpack ≃ λ f. λ x. x f } = β .
```

Figure D.1: CDLE derivation of *RExt*

APPENDIX E
BASIC META-THEORY

E.1 Variables

Lemma E.1. *If* $\vdash \Gamma \searrow \Gamma'$ *then*

- *if* $(X:\kappa) \in \Gamma$ *then there exists* κ' *such that* $\Gamma \vdash \kappa \searrow \kappa'$ *and* $(X:\kappa') \in \Gamma'$*; and*
- *if* $(x:T) \in \Gamma$ *then there exists* T' *such that* $\Gamma \vdash T : \star \searrow T'$ *and* $(X:T') \in \Gamma'$.

Proof. By a straightforward inductive argument on the derivation of $\vdash \Gamma \searrow \Gamma'$. □

Definition E.2. For Γ such that $\vdash \Gamma \searrow \Gamma'$ for some Γ', define $Seq(\Gamma)$ as follows.

$$\begin{array}{rcl} Seq(\emptyset) & = & () \\ Seq(\Gamma, x:T) & = & Seq(\Gamma)(x) \\ Seq(\Gamma, X:\kappa) & = & Seq(\Gamma)(\cdot\ X) \\ Seq(\Gamma, Decl[D, \Delta]) & = & Seq(\Gamma) \end{array}$$

Fact E.3. *If* $\vdash \Gamma \searrow \Gamma'$ *then* $Seq(\Gamma) = Seq(\Gamma')$.

Lemma E.4 (Variable erasure checks)**.** *If* $\vdash \Gamma \searrow \Gamma'$ *and* $\Gamma \vdash t \overset{\leftrightarrow}{\in} T \searrow t'$*, then for all* x *such that* $x \notin FV(|t|)$ *we have* $x \notin FV(|t'|)$.

Proof. By a straightforward induction on the assumed derivation that t elaborates to t'. □

E.2 Unicity

The lemmas in this section are all proven simultaneously by mutual induction. We have chosen to split them up into separate headings for easier reading (and writing).

Lemma E.5 (Unicity (positivity checker and subtyping))**.** *If* $\vdash \Gamma$ *then:*

1. *if* $\Gamma \vdash +F \rightsquigarrow t_1$ *and* $\Gamma \vdash +F \rightsquigarrow t_2$ *then* $t_1 = t_2$*; and*
2. *if* $\Gamma \vdash S \leqslant_s T \rightsquigarrow t_1$ *and* $\Gamma \vdash S \leqslant_s T \rightsquigarrow t_2$ *then* $t_1 = t_2$.

Proof. Part 1. follows as a corrolary to Part 2. Part 2. proceeds by mutual induction, proving a stronger statement that guarantees unicity of subtyping for types in weak head normal form. □

Lemma E.6 (Unicity (signature elaboration))**.** *If* $\vdash \Gamma \searrow \Gamma'$ *then:*

1. *if* $\Gamma \vdash_{\textbf{\textit{sig}}} Decl[D, \Delta] \searrow \mathcal{S}_1$ *and* $\Gamma \vdash_{\textbf{\textit{sig}}} Decl[D, \Delta] \searrow \mathcal{S}_2$ *then* $\mathcal{S}_1 = \mathcal{S}_2$*;*
2. *if* $\Gamma \vdash_{\textbf{\textit{con}}} Decl[D, \Delta] \searrow \mathcal{C}_1$ *and* $\Gamma \vdash_{\textbf{\textit{con}}} Decl[D, \Delta] \searrow \mathcal{C}_2$ *then* $\mathcal{C}_1 = \mathcal{C}_2$*;*
3. *if* $\Gamma \vdash_{\textbf{\textit{isig}}} Decl[D, \Delta] \searrow \mathcal{I}_1$ *and* $\Gamma \vdash_{\textbf{\textit{isig}}} Decl[D, \Delta] \searrow \mathcal{I}_2$ *then* $\mathcal{I}_1 = \mathcal{I}_2$*; and*

4. if $\Gamma \vdash_{\textit{icon}} \mathit{Decl}[D, \Delta] \searrow \mathcal{R}_1$ *and* $\Gamma \vdash_{\textit{icon}} \mathit{Decl}[D, \Delta] \searrow \mathcal{R}_2$ *then* $\mathcal{R}_1 = \mathcal{R}_2$.

Proof. 1.

$$\dfrac{\dfrac{(c_i : \Pi\, \Delta_i.\, D) \in \Delta \quad \Gamma, D : \star \vdash \Delta_i \searrow \Delta_i'}{\Gamma \vdash_{\mathbf{sig}} (D, R, \Delta, i) \searrow (c_i, \Delta_i'[R/D])} \quad R \notin \{D\} \cup DV(\Gamma) \cup \bigcup_{(c_i : \Pi\, \Delta_i.\, D) \in \Delta} DV(\Delta_i) \quad (\Gamma \vdash_{\mathbf{sig}} (D, R, \Delta, i) \searrow (c_i, \Delta_i'))_{i \in \{1 \ldots \#\Delta\}} \quad \underset{i \le \#\Delta}{\Xi(i)} =_{\mathbf{df}} (c_i : \Pi\, \Delta_i'.\, D)}{\Gamma \vdash_{\mathbf{sig}} \mathit{Decl}[D, \Delta] \searrow \mathsf{record}\ \mathit{Sig}\ \{D = D; R = R; \Xi = \Xi\}}$$

$$\dfrac{\dfrac{(c_i : \Pi\, \Delta_i.\, D) \in \Delta \quad \Gamma, D : \star \vdash \Delta_i \searrow \Delta_i''}{\Gamma \vdash_{\mathbf{sig}} (D, R, \Delta, i) \searrow (c_i, \Delta_i'[R/D])} \quad R \notin \{D\} \cup DV(\Gamma) \cup \bigcup_{(c_i : \Pi\, \Delta_i.\, D) \in \Delta} DV(\Delta_i) \quad (\Gamma \vdash_{\mathbf{sig}} (D, R, \Delta, i) \searrow (c_i, \Delta_i''))_{i \in \{1 \ldots \#\Delta\}} \quad \underset{i \le \#\Delta}{\Xi(i)} =_{\mathbf{df}} (c_i : \Pi\, \Delta_i''.\, D)}{\Gamma \vdash_{\mathbf{sig}} \mathit{Decl}[D, \Delta] \searrow \mathsf{record}\ \mathit{Sig}\ \{D = D; R = R; \Xi = \Xi\}}$$

We make the simplifying assumption that the choice of fresh type variable R is deterministic. It therefore becomes clear that the only question is whether $\Delta_i' = \Delta_i''$ for all $i \in \{1 \ldots \#\Delta\}$. We obtain this from Lemma E.14.

2.

$$\dfrac{\Gamma \vdash_{\mathbf{sig}} \mathit{Decl}[D, \Delta] \searrow \mathcal{S}_1 \quad \Gamma \vdash_{\mathbf{cty}} \mathcal{S}_1 \searrow \Delta' \quad \Gamma \vdash_{\mathbf{ctm}} (\mathcal{S}_1, \Delta') \searrow \tau}{\Gamma \vdash_{\mathbf{con}} \mathit{Decl}[D, \Delta] \searrow \mathsf{record}\ \mathit{Con}\ \{\mathcal{S} = \mathcal{S}_1; \Delta = \Delta'; \tau = \tau\}}$$

$$\dfrac{\Gamma \vdash_{\mathbf{sig}} \mathit{Decl}[D, \Delta] \searrow \mathcal{S}_2 \quad \Gamma \vdash_{\mathbf{cty}} \mathcal{S}_2 \searrow \Delta'' \quad \Gamma \vdash_{\mathbf{ctm}} (\mathcal{S}_2, \Delta'') \searrow \tau}{\Gamma \vdash_{\mathbf{con}} \mathit{Decl}[D, \Delta] \searrow \mathsf{record}\ \mathit{Con}\ \{\mathcal{S} = \mathcal{S}_1; \Delta = \Delta''; \tau = \tau\}}$$

By Part 1., $\mathcal{S}_1 = \mathcal{S}_2$. We will call this record $\mathcal{S}$ from now on. Now consider the auxiliary judgments.

- $$\dfrac{\dfrac{(c_i : \Pi\, \Delta_i'.\, D) \in \mathcal{S}.\Xi}{\Gamma \vdash_{\mathbf{cty}} (\mathcal{S}, i) \searrow (c, \forall\, \mathcal{S}.R : \star.\, \Pi\, \Delta_i'.\, \mathit{sig}(\mathcal{S}) \cdot \mathcal{S}.R)} \quad (\Gamma \vdash_{\mathbf{cty}} (\mathcal{S}, i) \searrow (c_i, T_i))_{i \in \{1 \ldots \#\mathcal{S}.\Xi\}} \quad \underset{i \le \#\mathcal{S}.\Xi}{\Delta'(i)} =_{\mathbf{df}} (c_i : T_i)}{\Gamma \vdash_{\mathbf{cty}} \mathcal{S} \searrow \Delta'}$$

  $$\dfrac{\dfrac{(c_i : \Pi\, \Delta_i'.\, D) \in \mathcal{S}.\Xi}{\Gamma \vdash_{\mathbf{cty}} (\mathcal{S}, i) \searrow (c, \forall\, \mathcal{S}.R : \star.\, \Pi\, \Delta_i'.\, \mathit{sig}(\mathcal{S}) \cdot \mathcal{S}.R)} \quad (\Gamma \vdash_{\mathbf{cty}} (\mathcal{S}, i) \searrow (c_i, T_i'))_{i \in \{1 \ldots \#\mathcal{S}.\Xi\}} \quad \underset{i \le \#\mathcal{S}.\Xi}{\Delta''(i)} =_{\mathbf{df}} (c_i : T_i')}{\Gamma \vdash_{\mathbf{cty}} \mathcal{S} \searrow \Delta''}$$

  It is immediate that $T_i = T_i'$ for $i \in \{1 \ldots \#\mathcal{S}.\Xi\}$. Therefore, $\Delta' = \Delta''$. Call this Δ' from now on.

- $$\frac{(c_i : \Pi\, \Delta_i'.\ D) \in \mathcal{S}.\Xi \quad (c_i : T_i) \in \Delta'}{\Gamma \vdash_{\mathbf{ctm}} (\mathcal{S}, \Delta', i) \searrow \chi\ T_i \text{ - } \Lambda\, \mathcal{S}.R.\, \lambda\, \Delta_i'.\, \Lambda\, \mathcal{S}.D.\, \lambda\, \mathcal{S}.\Xi.\, c_i\ \Delta_i'}$$
$$\frac{(\Gamma \vdash (\mathcal{S}, \Delta', i) \searrow t_i)_{i \in \{1 \ldots \#\mathcal{S}.\Xi\}} \qquad \underset{i \leq \#\mathcal{S}.\Xi}{\tau(i)} =_{\mathbf{df}} (t_i)}{\Gamma \vdash_{\mathbf{ctm}} (\mathcal{S}, \Delta') \searrow \tau}$$
$$\frac{(c_i : \Pi\, \Delta_i'.\ D) \in \mathcal{S}.\Xi \quad (c_i : T_i) \in \Delta'}{\Gamma \vdash_{\mathbf{ctm}} (\mathcal{S}, \Delta', i) \searrow \chi\ T_i \text{ - } \Lambda\, \mathcal{S}.R.\, \lambda\, \Delta_i'.\, \Lambda\, \mathcal{S}.D.\, \lambda\, \mathcal{S}.\Xi.\, c_i\ \Delta_i'}$$
$$\frac{(\Gamma \vdash (\mathcal{S}, \Delta', i) \searrow t_i')_{i \in \{1 \ldots \#\mathcal{S}.\Xi\}} \qquad \underset{i \leq \#\mathcal{S}.\Xi}{\tau'(i)} =_{\mathbf{df}} (t_i)}{\Gamma \vdash_{\mathbf{ctm}} (\mathcal{S}, \Delta') \searrow \tau'}$$

  It is clear $t_i = t_i'$ for $i \in \{1 \ldots \#\mathcal{S}.\Xi\}$, therefore $\tau = \tau'$.

3. $$\frac{\Gamma \vdash_{\mathbf{con}} \mathit{Decl}[D, \Delta] \searrow \mathcal{C}_1 \quad \{z, v\} \cap (DV(\Gamma) \cup DV(\Delta)) = \emptyset \wedge z \neq v \quad ((c_i : \Pi\, \Delta_i'.\ D) \in \mathcal{C}_1.\Xi)_{i \in \{1 \ldots \#\mathcal{S}.\Xi\}} \quad \underset{i \leq \#\mathcal{C}_1.\Xi}{\Sigma(i)} =_{\mathbf{df}} (c_i : \Pi\, \Delta_i'.\ D\ (\mathcal{C}_1.\tau(i) \cdot \mathcal{C}_1.R\ \Delta_i'))}{\Gamma \vdash_{\mathbf{isig}} \mathit{Decl}[D, \Delta] \searrow \mathsf{record}\ \mathit{ISig}\ \{\mathcal{C} = \mathcal{C}_1; z = z; v = v; \Sigma = \Sigma\}}$$
$$\frac{\Gamma \vdash_{\mathbf{con}} \mathit{Decl}[D, \Delta] \searrow \mathcal{C}_2 \quad \{z, v\} \cap (DV(\Gamma) \cup DV(\Delta)) = \emptyset \wedge z \neq v \quad ((c_i' : \Pi\, \Delta_i''.\ D) \in \mathcal{C}_2.\Xi)_{i \in \{1 \ldots \#\mathcal{S}.\Xi\}} \quad \underset{i \leq \#\mathcal{C}_2.\Xi}{\Sigma'(i)} =_{\mathbf{df}} (c_i' : \Pi\, \Delta_i''.\ D\ (\mathcal{C}_2.\tau(i) \cdot \mathcal{C}_2.R\ \Delta_i''))}{\Gamma \vdash_{\mathbf{isig}} \mathit{Decl}[D, \Delta] \searrow \mathsf{record}\ \mathit{ISig}\ \{\mathcal{C} = \mathcal{C}_2; z = z; v = v; \Sigma = \Sigma'\}}$$

   From Part 2. above, $\mathcal{C}_1 = \mathcal{C}_2$. We assume that the selection of fresh variables z, v is deterministic. We therefore see that $c_i = c_i'$ and $\Delta_i' = \Delta_i''$ for all $i \in \{1 \ldots \#\mathcal{S}.\Xi\}$, and therefore $\Sigma = \Sigma'$.

4. $$\frac{\Gamma \vdash_{\mathbf{isig}} \mathit{Decl}[D, \Delta] \searrow \mathcal{I}_1 \quad \Gamma \vdash_{\mathbf{icty}} \mathcal{I}_1 \searrow \Theta_1 \quad \Gamma \vdash_{\mathbf{ictm}} (\mathcal{I}_1, \Theta_1) \searrow \gamma_1}{\Gamma \vdash_{\mathbf{icon}} \mathit{Decl}[D, \Delta] \searrow \mathsf{record}\ \mathit{ICon}\ \{\mathcal{I} = \mathcal{I}_1, \Theta = \Theta_1, \gamma = \gamma_1\}}$$
$$\frac{\Gamma \vdash_{\mathbf{isig}} \mathit{Decl}[D, \Delta] \searrow \mathcal{I}_2 \quad \Gamma \vdash_{\mathbf{icty}} \mathcal{I} \searrow \Theta_2 \quad \Gamma \vdash_{\mathbf{ictm}} (\mathcal{I}, \Theta) \searrow \gamma_2}{\Gamma \vdash_{\mathbf{icon}} \mathit{Decl}[D, \Delta] \searrow \mathsf{record}\ \mathit{ICon}\ \{\mathcal{I} = \mathcal{I}_2, \Theta = \Theta_2, \gamma = \gamma_2\}}$$

   By Part 3. above, $\mathcal{I}_1 = \mathcal{I}_2$. Call this record $\mathcal{I}$ henceforth. We now turn to the auxiliary judgments.

- $$\frac{(c_i : \Pi\, \Delta_i'.\, D) \in \mathcal{I}.\Xi}{\Gamma \vdash_{\mathbf{icty}} (\mathcal{I}, i) \searrow (c_i, \forall\, \mathcal{I}.R : \star.\, \Pi\, \Delta_i'.\, isig(\mathcal{I}) \cdot \mathcal{I}.R)} \quad \frac{(\Gamma \vdash_{\mathbf{icty}} (\mathcal{I}, i) \searrow (c_i, T_i))_{i \in \{1 \ldots \#\mathcal{I}.\Xi\}} \quad \underset{i \leq \#\mathcal{I}.\Sigma}{\Theta(i)} =_{\mathbf{df}} (c_i : T_i)}{\Gamma \vdash_{\mathbf{icty}} \mathcal{I} \searrow \Theta}$$
$$\frac{(c_i : \Pi\, \Delta_i'.\, D) \in \mathcal{I}.\Xi}{\Gamma \vdash_{\mathbf{icty}} (\mathcal{I}, i) \searrow (c_i, \forall\, \mathcal{I}.R : \star.\, \Pi\, \Delta_i'.\, isig(\mathcal{I}) \cdot \mathcal{I}.R)} \quad \frac{(\Gamma \vdash_{\mathbf{icty}} (\mathcal{I}, i) \searrow (c_i', T_i'))_{i \in \{1 \ldots \#\mathcal{I}.\Xi\}} \quad \underset{i \leq \#\mathcal{I}.\Sigma}{\Theta'(i)} =_{\mathbf{df}} (c_i' : T_i')}{\Gamma \vdash_{\mathbf{icty}} \mathcal{I} \searrow \Theta'}$$

  It is clear $c_i = c_i'$ and $T_i = T_i'$ for all $i \in \{1 \ldots \#\mathcal{I}.\Xi\}$, and therefore $\Theta = \Theta'$. Call this telescope Θ henceforth.

- $$\frac{(c_i : T_i) \in \Theta \quad (c_i : \Pi\, \Delta_i'.\, D) \in \mathcal{I}.\Xi}{\Gamma \vdash_{\mathbf{ictm}} (\mathcal{I}, \Theta, i) \searrow \chi\ T_i \text{ - } \Lambda\, \mathcal{I}.R.\, \lambda\, \Delta_i'.\, [\mathcal{I}.\tau(i) \cdot \mathcal{I}.R\ \Delta_i', \Lambda\, \mathcal{I}.D.\, \lambda\, \mathcal{I}.\Sigma.\, c_i\ \Delta_i']} \quad \frac{(\Gamma \vdash_{\mathbf{ictm}} (\mathcal{I}, \Theta, i) \searrow t_i)_{i \in \{1 \ldots \#\Theta\}} \quad \underset{i \leq \#\mathcal{I}.\Theta}{\gamma(i)} =_{\mathbf{df}} t_i}{\Gamma \vdash_{\mathbf{ictm}} (\mathcal{I}, \Theta) \searrow \gamma}$$
$$\frac{(c_i : T_i) \in \Theta \quad (c_i : \Pi\, \Delta_i'.\, D) \in \mathcal{I}.\Xi}{\Gamma \vdash_{\mathbf{ictm}} (\mathcal{I}, \Theta, i) \searrow \chi\ T_i \text{ - } \Lambda\, \mathcal{I}.R.\, \lambda\, \Delta_i'.\, [\mathcal{I}.\tau(i) \cdot \mathcal{I}.R\ \Delta_i', \Lambda\, \mathcal{I}.D.\, \lambda\, \mathcal{I}.\Sigma.\, c_i\ \Delta_i']} \quad \frac{(\Gamma \vdash_{\mathbf{ictm}} (\mathcal{I}, \Theta, i) \searrow t_i')_{i \in \{1 \ldots \#\Theta\}} \quad \underset{i \leq \#\mathcal{I}.\Theta}{\gamma'(i)} =_{\mathbf{df}} t_i'}{\Gamma \vdash_{\mathbf{ictm}} (\mathcal{I}, \Theta) \searrow \gamma'}$$

  It is clear $t_i = t_i'$ for all $i \in \{1 \ldots \#\Theta\}$, and therefore $\gamma = \gamma'$.

□

Lemma E.7 (Unicity (datatype elaboration))**.** *If* $\vdash \Gamma \searrow \Gamma'$ *then*

1. *if* $\Gamma \vdash_{\textbf{\textit{data}}} Decl[D, \Delta] \searrow \mathcal{D}_1$ *and* $\Gamma \vdash_{\textbf{\textit{data}}} Decl[D, \Delta] \searrow \mathcal{D}_2$ *then* $\mathcal{D}_1 = \mathcal{D}_2$*;*
2. *if* $\Gamma \vdash_{\textbf{\textit{dcon}}} (\mathcal{R}, i) \searrow (c, T)$ *and* $\Gamma \vdash_{\textbf{\textit{dcon}}} (\mathcal{R}, i) \searrow (c', T')$ *then* $(c, T) = (c', T')$*; and*
3. *if* $\Gamma \vdash_{\textbf{\textit{dcon}}} (\mathcal{R}, t, i) \searrow t'$ *and* $\Gamma \vdash_{\textbf{\textit{dcon}}} (\mathcal{R}, t, i) \searrow t''$ *then* $t' = t''$*.*

Proof. Parts 2. and 3. follow from the assumption that constructors identifiers are distinct.

For part 1., invert the premises

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathbf{icon}} Decl[D, \Delta] \searrow \mathcal{R}_1 \quad \Gamma \vdash +isig(\mathcal{R}_1) \rightsquigarrow t \\ (\Gamma \vdash_{\mathbf{dcon}} (\mathcal{R}_1, i) \searrow (c_i, T_i))_{i \in \{1 \ldots \#\Delta\}} \quad (\Gamma \vdash_{\mathbf{dcon}} (\mathcal{R}_1, t', i) \searrow t_i)_{i \in 1 \ldots \#\Delta} \\ \underset{i \leq \#\Delta}{\Delta'(i)} =_{\mathbf{df}} (c_i : T_i) \quad \underset{i \leq \#\Delta}{\theta(i)} =_{\mathbf{df}} (t_i)\end{array}}{\Gamma \vdash_{\mathbf{data}} Decl[D, \Delta] \searrow \mathsf{record}\ Data \quad \begin{array}{l}\{\mathcal{R} = \mathcal{R}_1; mono = t_1; D = \mu(isig(\mathcal{R}_1)) \\ ; \Delta = \Delta_1'; \theta = \theta\}\end{array}}$$

$$\frac{\begin{array}{cc}\Gamma \vdash_{\mathbf{icon}} \mathit{Decl}[D,\Delta] \searrow \mathcal{R}_{\in} & \Gamma \vdash +\mathit{isig}(\mathcal{R}_2) \rightsquigarrow t' \\ (\Gamma \vdash_{\mathbf{dcon}} (\mathcal{R}_2, i) \searrow (c_i', T_i'))_{i\in\{1\ldots\#\Delta\}} & (\Gamma \vdash_{\mathbf{dcon}} (\mathcal{R}_2, t', i) \searrow t_i')_{i\in 1\ldots\#\Delta} \\ \Delta''(i) =_{\mathbf{df}} (c_i' : T_i) & \theta'(i) =_{\mathbf{df}} (t_i') \\ i\leq\#\Delta & i\leq\#\Delta\end{array}}{\Gamma \vdash_{\mathbf{data}} \mathit{Decl}[D,\Delta] \searrow \mathsf{record}\ \mathit{Data}\ \begin{array}{l}\{\mathcal{R} = \mathcal{R}_2; \mathit{mono} = t''; D = \mu(\mathit{isig}(\mathcal{R}_2)) \\ ; \Delta = \Delta''; \theta = \theta'\}\end{array}}$$

- By Lemma E.6, $\mathcal{R}_1 = \mathcal{R}_2$.
- By Lemma E.5, $t = t'$.
- By parts 2. and 3., $(c_i, T_i)_{i\in\{1\ldots\#\Delta\}} = (c_i', T_i')_{i\in\{1\ldots\#\Delta\}}$ and $(t_i)_{i\in\{1\ldots\#\Delta\}} = (t_i')_{i\in\{1\ldots\#\Delta\}}$. This makes $\Delta' = \Delta''$ and $\theta = \theta'$

□

Lemma E.8 (Unicity (datatype lookup))**.** *If* $\vdash \Gamma \searrow \Gamma'$ *then:*

1. *if* $\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta_1] \searrow \mathcal{D}_1$ *and* $\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta_2] \searrow \mathcal{D}_2$ *then* $\Delta_1 = \Delta_2$ *and* $\mathcal{D}_1 = \mathcal{D}_2$*;*
2. *if* $\Gamma \vdash \mathit{Con}[c] : \mathit{Decl}[D_1, \Delta_1] \searrow \mathcal{D}_1$ *and* $\Gamma \vdash \mathit{Con}[c] : \mathit{Decl}[D_2, \Delta_2] \searrow \mathcal{D}_2$ *then* $D_1 = D_2$*,* $\Delta_1 = \Delta_2$*, and* $\mathcal{D}_1 = \mathcal{D}_2$*.*
3. *if* $\Gamma \vdash (c, \mathit{Decl}[D, \Delta], j) : \mathit{Decl}[D_1, \Delta_1] \searrow (\mathcal{D}_1, i_1)$ *and* $\Gamma \vdash (c, \mathit{Decl}[D, \Delta], j) : \mathit{Decl}[D_2, \Delta_2] \searrow (\mathcal{D}_2, i_2)$ *then* $D_1 = D_2$*,* $\Delta_1 = \Delta_2$*,* $\mathcal{D}_1 = \mathcal{D}_2$*, and* $i_1 = i_2$*.*

Proof. Part 1. is by a straightforward induction on the assumed derivations and an appeal to Lemma E.7. Parts 2. and 3. are by mutual induction and an appeal to Lemma E.7. □

Lemma E.9 (Unicity (datatype lookup, untyped))**.** *If* $\vdash \Gamma \searrow \Gamma'$ *then if* $\Gamma \vdash \mathit{ConU}[\Delta'] : \mathit{Decl}[D_1, \Delta_1] \searrow \mathcal{D}_1$ *and* $\Gamma \vdash \mathit{ConU}[\Delta'] : \mathit{Decl}[D_2, \Delta_2] \searrow \mathcal{D}_2$ *then* $D_1 = D_2$*,* $\Delta_1 = \Delta_2$*, and* $\mathcal{D}_1 = \mathcal{D}_2$*.*

Proof. By a straightforward induction on the assumed derivations of untyped lookup, appealing to Lemma E.7. □

Lemma E.10 (Unicity (elaboration of untyped terms))**.**
If $\vdash \Gamma_1, \Gamma_2 \searrow \Gamma'$ *then if* $\Gamma_1; \Gamma_2 \vdash t \searrow t'$ *and* $\Gamma_1; \Gamma_2 \vdash t \searrow t''$ *then* $t' = t''$*.*

Proof. By induction on the assumed derivations. We show only the interesting cases.
Case:

$$\frac{\Gamma_1 \vdash \mathit{Con}[c] : \mathit{Decl}[D_1, \Delta_1] \searrow (\mathcal{D}_1, i_1)}{\Gamma_1; \Gamma_2 \vdash c \searrow |\mathcal{D}_1.\theta(i_1)|} \quad \frac{\Gamma_1 \vdash \mathit{Con}[c] : \mathit{Decl}[D_2, \Delta_2] \searrow (\mathcal{D}_2, i_2)}{\Gamma_1; \Gamma_2 \vdash c \searrow |\mathcal{D}_2.\theta(i_2)|}$$

By Lemma E.8, $D_1 = D_2$, $\Delta_1 = \Delta_2$, $\mathcal{D}_1 = \mathcal{D}_2$, and $i_1 = i_2$. This suffices.

Case:

$$\frac{\begin{array}{c}\Gamma_1;\Gamma_2 \vdash t \searrow t' \quad \Gamma_1 \vdash \mathit{ConU}\left[\underset{i\leq n}{\Delta'(i) =_{\mathbf{df}} (c_i : \mathit{Top})}\right] : \mathit{Decl}[D_1, \Delta_1] \searrow \mathcal{D}_1 \\ (\#\varrho_i = \#|\Delta_i| \text{ where } (c_i : \Pi\, \Delta_i.\, D_1) \in \Delta_1)_{i\in\{1\ldots n\}} \\ (\Gamma_1;\Gamma_2 \vdash \lambda\, \varrho_i.\, t_i \searrow t'_i)_{i\in\{1\ldots n\}} \quad \underset{i\leq n}{\tau(i) =_{\mathbf{df}} (t'_i)}\end{array}}{\Gamma_1;\Gamma_2 \vdash \sigma\ t\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \searrow |\mathit{sigma}\ t'\ (\mathit{case}(\mathcal{D}_1)\ \tau')|}$$

$$\frac{\begin{array}{c}\Gamma_1;\Gamma_2 \vdash t \searrow t'' \quad \Gamma_1 \vdash \mathit{ConU}\left[\underset{i\leq n}{\Delta'(i) =_{\mathbf{df}} (c_i : \mathit{Top})}\right] : \mathit{Decl}[D_2, \Delta_2] \searrow \mathcal{D}_2 \\ (\#\varrho_i = \#|\Delta'_i| \text{ where } (c'_i : \Pi\, \Delta'_i.\, D_2) \in \Delta_2)_{i\in\{1\ldots n\}} \\ (\Gamma_1;\Gamma_2 \vdash \lambda\, \varrho_i.\, t_i \searrow t''_i)_{i\in\{1\ldots n\}} \quad \underset{i\leq n}{\tau'(i) =_{\mathbf{df}} (t'_i)}\end{array}}{\Gamma_1;\Gamma_2 \vdash \sigma\ t\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \searrow |\mathit{sigma}\ t''\ (\mathit{case}(\mathcal{D}_2)\ \tau)|}$$

- By the IH, $t' = t''$.
- By Lemma E.9, $D_1 = D_2$, $\Delta_1 = \Delta_2$, and $\mathcal{D}_1 = \mathcal{D}_2$.
- By the IH, $t'_i = t''_i$ for all $i \in \{1 \ldots n\}$.

This suffices. □

Lemma E.11 (Unicity (kinds))**.** *If* $\vdash \Gamma \searrow \Gamma'$ *then if* $\Gamma \vdash \kappa \searrow \kappa'_1$ *and* $\Gamma \vdash \kappa \searrow \kappa'_2$ *then* $\kappa'_1 = \kappa'_2$

Proof. By a straightforward induction on the assumed derivations of kind elaboration, appealing to Lemma E.12 and inversion for quantification over terms. □

Lemma E.12 (Unicity (type constructors))**.** *If* $\vdash \Gamma \searrow \Gamma'$ *then if* $\Gamma \vdash T : \kappa_1 \searrow T'_1$ *and* $\Gamma \vdash T : \kappa_2 \searrow T'_2$ *then* $\kappa_1 = \kappa_2$ *and* $T'_1 = T'_2$.

Proof. By induction on the assumed derivations of type constructor elaboration. All cases except two, which involve datatypes, are congruence rules. We therefore only show a handful of cases.

Case:

$$\frac{\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta_1] \searrow \mathcal{D}_1}{\Gamma \vdash D : \star \searrow \mathcal{D}_1.D} \quad \frac{\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta_2] \searrow \mathcal{D}_2}{\Gamma \vdash D : \star \searrow \mathcal{D}.D}$$

By Lemma E.8, $\Delta_1 = \Delta_2$ and $\mathcal{D}_1 = \mathcal{D}_2$. Therefore, $\mathcal{D}_1.D = \mathcal{D}_2.D$.

Case:

$$\frac{\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta_1] \searrow \mathcal{D}_1}{\Gamma \vdash \mathtt{Is}/D : \star \to \star \searrow \mathit{IsD} \cdot \mathit{isig}(\mathcal{D}_1)} \quad \frac{\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta_2] \searrow \mathcal{D}_2}{\Gamma \vdash \mathtt{Is}/D : \star \to \star \searrow \mathit{IsD} \cdot \mathit{isig}(\mathcal{D}_2)}$$

Similar to the above case.

Case:

$$\frac{\Gamma \vdash T_1 : \Pi\, X : \kappa_2.\, \kappa_1 \searrow T_1' \quad \Gamma \vdash T_2 : \kappa_3 \searrow T_2' \quad \kappa_2 \cong \kappa_3}{\Gamma \vdash T_1 \cdot T_2 : \kappa_1[T_2/X] \searrow T_1' \cdot T_2'} \qquad \frac{\Gamma \vdash T_1 : \Pi\, X : \kappa_5.\, \kappa_4 \searrow T_1'' \quad \Gamma \vdash T_2 : \kappa_6 \searrow T_2'' \quad \kappa_5 \cong \kappa_6}{\Gamma \vdash T_1 \cdot T_2 : \kappa_4[T_2/X] \searrow T_1'' \cdot T_2''}$$

By the IH and inversion, $\kappa_2 = \kappa_5$, $\kappa_1 = \kappa_4$, $T_1' = T_1''$, $\kappa_6 = \kappa_3$, and $T_2' = T_2''$. We therefore have that $\kappa_1[T_2/X] = \kappa_4[T_2/X]$ and $T_1' \cdot T_2' = T_1'' \cdot T_2''$.

Case:

$$\frac{\Gamma; \emptyset \vdash |t_1| \searrow t_1' \quad \Gamma; \emptyset \vdash |t_2| \searrow t_2'}{\Gamma \vdash \{t_1 \simeq t_2\} : \star \searrow \{t_1' \simeq t_2'\}} \qquad \frac{\Gamma; \emptyset \vdash |t_1| \searrow t_1'' \quad \Gamma; \emptyset \vdash |t_2| \searrow t_2''}{\Gamma \vdash \{t_1 \simeq t_2\} : \star \searrow \{t_1'' \simeq t_2''\}}$$

By Lemma E.10, $t_1' = t_1''$ and $t_2' = t_2''$. Therefore, $\{t_1' \simeq t_2'\} = \{t_1'' \simeq t_2''\}$. □

Lemma E.13. *If* $\vdash \Gamma \searrow \Gamma'$ *then*

1. *if* $\Gamma \vdash t \in T_1 \searrow t_1'$ *and* $\Gamma \vdash t \in T_2 \searrow t_2'$ *then* $T_1 = T_2$ *and* $t_1' = t_2'$*; and*
2. *if* $\Gamma \vdash T \ni t \searrow t_1'$ *and* $\Gamma \vdash T \ni t \searrow t_2'$ *then* $t_1' = t_2'$.

Proof. By mutual induction on the assumed derivations of the elaboration of terms. Most cases are congruence rules, so we therefore only show the interesting cases.

Case:

$$\frac{\Gamma \vdash \mathit{Con}[c] : \mathit{Decl}[D_1, \Delta_1] \searrow (\mathcal{D}_1, i_1) \quad (c : \Pi\, \Delta'.\, D_1) \in \Delta_1}{\Gamma \vdash c \in \Pi\, \Delta'.\, D_1 \searrow \mathcal{D}_1.\theta(i_1)}$$

$$\frac{\Gamma \vdash \mathit{Con}[c] : \mathit{Decl}[D_2, \Delta_2] \searrow (\mathcal{D}_2, i_2) \quad (c : \Pi\, \Delta''.\, D_2) \in \Delta_2}{\Gamma \vdash c \in \Pi\, \Delta''.\, D_2 \searrow \mathcal{D}_2.\theta(i_2)}$$

By Lemma E.8, $D_1 = D_2$, $\Delta_1 = \Delta_2$, and $i_1 = i_2$. By the assumption that constructor telescopes list disjoint identifiers, $\Delta' = \Delta''$. We therefore conclude that $\Pi\, \Delta'.\, D_1 = \Pi\, \Delta''.\, D_2$ and $\mathcal{D}_1.\theta(i_1) = \mathcal{D}_2.\theta(i_2)$.

Case:

$$\frac{\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta_1] \searrow \mathcal{D}_1}{\Gamma \vdash \texttt{is/}D \in \texttt{Is/}D \cdot D \searrow \mathit{isD} \cdot \mathit{isig}(\mathcal{D}_1)\ \text{-}\mathcal{D}_1.\mathit{mono}}$$

$$\frac{\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta_2] \searrow \mathcal{D}_2}{\Gamma \vdash \texttt{is/}D \in \texttt{Is/}D \cdot D \searrow \mathit{isD} \cdot \mathit{isig}(\mathcal{D}_2)\ \text{-}\mathcal{D}_2.\mathit{mono}}$$

Similar to the above case.

Case:

$$\frac{\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta_1] \searrow \mathcal{D}_1}{\begin{array}{l}\Gamma \vdash \texttt{to/}D \in \forall\, X : \star.\, \texttt{Is/}D \cdot X \Rightarrow X \to D \\ \quad \searrow \chi\ (\forall\, X : \star.\, \mathit{IsD} \cdot \mathit{isig}(\mathcal{D}_1) \cdot X \Rightarrow X \to \mu(\mathit{isig}(\mathcal{D}_1))) \\ \quad\quad \text{-}\ \Lambda\, X.\, \Lambda\, x.\, \mathit{elimCast}\ \text{-}(\mathit{toD} \cdot \mathit{isig}(\mathcal{D}_2) \cdot X\ \text{-}x)\end{array}}$$

$$\frac{\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta_2] \searrow \mathcal{D}_2}{\begin{array}{l}\Gamma \vdash \texttt{to/}D \in \forall\, X : \star.\, \texttt{Is/}D \cdot X \Rightarrow X \to D \\ \quad \searrow \chi\ (\forall\, X : \star.\, \mathit{IsD} \cdot \mathit{isig}(\mathcal{D}_2) \cdot X \Rightarrow X \to \mu(\mathit{isig}(\mathcal{D}_2))) \\ \quad\quad \text{-}\ \Lambda\, X.\, \Lambda\, x.\, \mathit{elimCast}\ \text{-}(\mathit{toD} \cdot \mathit{isig}(\mathcal{D}_2) \cdot X\ \text{-}x)\end{array}}$$

Similar to the above case.

Case:

$$\frac{\begin{array}{c}\Gamma \vdash t \in S_1 \searrow t' \\ \mathfrak{M}_1 =_{\mathbf{df}} \mathsf{record}\ \mathit{MotiveSrc}\ \{S = S_1; s = s; T = T\} \\ \mathfrak{P}_1 =_{\mathbf{df}} \mathsf{record}\ \mathit{PattSrc}\ \{S = S_1; s = s\} \\ \mathfrak{B} =_{\mathbf{df}} \mathsf{record}\ \mathit{BranchSrc}\ \{\gamma_n(i) = (c_i); (\varrho_i = \varrho_i)_{i\in\{1\ldots n\}}; (t_i = t_i)_{i\in\{1\ldots n\}}\} \\ \Gamma \vdash \mathfrak{M}_1 \in \mathit{Decl}[D, \Delta] \searrow \mathcal{M}_1 \quad \Gamma \vdash (\mathit{Decl}[D, \Delta], \mathfrak{M}_1, \mathfrak{P}_1) \ni \mathfrak{B} \searrow \mathcal{B}_1\end{array}}{\begin{array}{l}\Gamma \vdash \sigma\langle s\rangle\ t\ @T\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \in T\ (\mathtt{to}/D\ \text{-}s\ t) \\ \quad \searrow \mathit{sigma} \cdot \mathit{isig}(\mathcal{M}_1)\ \text{-}\mathcal{M}_1.\mathit{mono} \cdot \mathcal{M}_1.S\ \text{-}\mathcal{M}_1.s\ t' \cdot \mathcal{M}_1.T\ \mathit{ctree}(\mathcal{M}_1, \mathcal{B}_1)\end{array}}$$

$$\frac{\begin{array}{c}\Gamma \vdash t \in S_2 \searrow t'' \\ \mathfrak{M}_2 =_{\mathbf{df}} \mathsf{record}\ \mathit{MotiveSrc}\ \{S = S_2; s = s; T = T\} \\ \mathfrak{P}_2 =_{\mathbf{df}} \mathsf{record}\ \mathit{PattSrc}\ \{S = S_2; s = s\} \\ \mathfrak{B} =_{\mathbf{df}} \mathsf{record}\ \mathit{BranchSrc}\ \{\gamma_n(i) = (c_i); (\varrho_i = \varrho_i)_{i\in\{1\ldots n\}}; (t_i = t_i)_{i\in\{1\ldots n\}}\} \\ \Gamma \vdash \mathfrak{M}_2 \in \mathit{Decl}[D', \Delta'] \searrow \mathcal{M}_2 \quad \Gamma \vdash (\mathit{Decl}[D', \Delta'], \mathfrak{M}_2, \mathfrak{P}_2) \ni \mathfrak{B} \searrow \mathcal{B}_2\end{array}}{\begin{array}{l}\Gamma \vdash \sigma\langle s\rangle\ t\ @T\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \in T\ (\mathtt{to}/D\ \text{-}s\ t) \\ \quad \searrow \mathit{sigma} \cdot \mathit{isig}(\mathcal{M}_2)\ \text{-}\mathcal{M}_2.\mathit{mono} \cdot \mathcal{M}_2.S\ \text{-}\mathcal{M}_2.s\ t'' \cdot \mathcal{M}_2.T\ \mathit{ctree}(\mathcal{M}_2, \mathcal{B}_2)\end{array}}$$

- By the IH, $S_1 = S_2$ and $t' = t''$. This means that $\mathfrak{M}_1 = \mathfrak{M}_2$ and $\mathfrak{P}_1 = \mathfrak{P}_2$. We henceforth refer to $\mathfrak{M}_1$, $\mathfrak{M}_2$ as $\mathfrak{M}$, and refer to $\mathfrak{P}_1$, $\mathfrak{P}_2$ as $\mathfrak{P}$.
- Invert the motive information derivations.

$$\frac{\begin{array}{c}y \notin \mathit{DV}(\Gamma) \quad \Gamma \vdash \mathfrak{M}.S : \star \searrow S_1' \quad \Gamma \vdash \mathfrak{M}.s \overset{\dashrightarrow^*}{\in} \mathtt{Is}/D \cdot S_1 \searrow s_1' \quad S_1 \cong \mathfrak{M}.S \\ \Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D} \quad \Gamma \vdash \mathfrak{M}.T : \kappa_1 \searrow T_1' \quad \kappa_1 \cong D \to \star\end{array}}{\Gamma \vdash \mathfrak{M} \in \mathit{Decl}[D, \Delta] \searrow \mathsf{record}\ \mathit{Motive}\ \{\mathcal{D} = \mathcal{D}; S = S_1'; s = s_1'; T = T_1'; y = y\}}$$

$$\frac{\begin{array}{c}y \notin \mathit{DV}(\Gamma) \quad \Gamma \vdash \mathfrak{M}.S : \star \searrow S_2' \quad \Gamma \vdash \mathfrak{M}.s \overset{\dashrightarrow^*}{\in} \mathtt{Is}/D' \cdot S_2 \searrow s_2' \quad S_2 \cong \mathfrak{M}.S \\ \Gamma \vdash \mathit{Data}[D'] : \mathit{Decl}[D', \Delta'] \searrow \mathcal{D}' \quad \Gamma \vdash \mathfrak{M}.T : \kappa_2 \searrow T_2' \quad \kappa_2 \cong D' \to \star\end{array}}{\Gamma \vdash \mathfrak{M} \in \mathit{Decl}[D', \Delta'] \searrow \mathsf{record}\ \mathit{Motive}\ \{\mathcal{D} = \mathcal{D}; S = S_2'; s = s_2'; T = T_2'; y = y\}}$$

  - We assume fresh variable y is picked deterministically.
  - By Lemma E.12, $S_1' = S_2'$ and $D = D'$.
  - By the IH (and an easy inductive proof on the reduction relation), $S_1 = S_2$ and $s_1' = s_2'$.
  - By Lemma E.8, $\Delta = \Delta'$ and $\mathcal{D} = \mathcal{D}'$.
  - By Lemma E.12, $\kappa_1 = \kappa_2$ and $T_1' = T_2'$

  This establishes that $\mathcal{M}_1 = \mathcal{M}_2$. We henceforth refer to $\mathcal{M}_1$, $\mathcal{M}_2$ as $\mathcal{M}$.

- Invert the branch information derivations.

$$\frac{\begin{array}{c}\mathfrak{B}.\gamma = \mathit{Seq}(\Delta) \quad (\vdash \mathfrak{B}.\varrho_i := \Delta_i)_{(c_i:\Pi\,\Delta_i.\,D)\in\Delta} \\ \Gamma \vdash \mathfrak{P}.S : \star \searrow S_1' \quad \Gamma \vdash \mathfrak{P}.s \overset{\dashrightarrow^*}{\in} \mathtt{Is}/D \cdot S_1 \searrow s_1' \quad S_1 \cong \mathfrak{P}.S \\ (\Gamma \vdash \Delta_i[\mathfrak{P}.S/D] \leqslant_{\mathtt{to}/D\ \text{-}\mathfrak{P}.s} \Delta_i \rightsquigarrow \xi_i)_{(c_i:\Pi\,\Delta_i.\,D)\in\Delta} \\ (\Gamma \vdash \Pi\,\Delta_i[\mathfrak{P}.S/D].\,\mathfrak{M}.T\ (c_i\ (\xi_i \odot \Delta_i)) \ni \lambda\,\mathfrak{B}.\varrho_i.\,\mathfrak{B}.t_i \searrow t_i')_{c_i:\Pi\,\Delta_i.\,D} \\ \underset{i\le\#\Delta}{\tau(i)} =_{\mathbf{df}} (t_i')\end{array}}{\Gamma \vdash (\mathit{Decl}[D,\Delta],\mathfrak{M},\mathfrak{P}) \ni \mathfrak{B} \searrow \mathsf{record}\ \mathit{Branch}\ \{S = S_1'; s = s_1'; \tau = \tau\}}$$

$$\frac{\begin{array}{c}\mathfrak{B}.\gamma = \mathit{Seq}(\Delta) \quad (\vdash \mathfrak{B}.\varrho_i := \Delta_i)_{(c_i:\Pi\,\Delta_i.\,D)\in\Delta} \\ \Gamma \vdash \mathfrak{P}.S : \star \searrow S_2' \quad \Gamma \vdash \mathfrak{P}.s \overset{\dashrightarrow^*}{\in} \mathtt{Is}/D \cdot S_2 \searrow s_2' \quad S_2 \cong \mathfrak{P}.S \\ (\Gamma \vdash \Delta_i[\mathfrak{P}.S/D] \leqslant_{\mathtt{to}/D\ \text{-}\mathfrak{P}.s} \Delta_i \rightsquigarrow \xi_i')_{(c_i:\Pi\,\Delta_i.\,D)\in\Delta} \\ (\Gamma \vdash \Pi\,\Delta_i[\mathfrak{P}.S/D].\,\mathfrak{M}.T\ (c_i\ (\xi_i' \odot \Delta_i)) \ni \lambda\,\mathfrak{B}.\varrho_i.\,\mathfrak{B}.t_i \searrow t_i'')_{c_i:\Pi\,\Delta_i.\,D} \\ \underset{i\le\#\Delta}{\tau'(i)} =_{\mathbf{df}} (t_i'')\end{array}}{\Gamma \vdash (\mathit{Decl}[D,\Delta],\mathfrak{M},\mathfrak{P}) \ni \mathfrak{B} \searrow \mathsf{record}\ \mathit{Branch}\ \{S = S_2'; s = s_2'; \tau = \tau'\}}$$

  - By Lemma E.12, $S_1' = S_2'$.
  - By the IH, $S_1 = S_2$ and $s_1' = s_2'$.
  - By Lemma E.5 (and an easy induction on the length of Δ_i for $i \in \{1 \dots \#\Delta\}$), $\xi_i = \xi_i'$ for $i \in \{1 \dots \#\Delta\}$.
  - By the IH, $t_i' = t_i''$ for $i \in \{1 \dots \#\Delta\}$. This means $\tau = \tau'$.

  This establishes that $\mathcal{B}_1 = \mathcal{B}_2$.

We therefore have that the elaborations of the two σ-expressions are equal (it is already apparent that the types the synthesize are equal).

Case:

$$\frac{\begin{array}{c}\Gamma \vdash t \overset{\dashrightarrow^*}{\in} D \searrow t' \\ \mathfrak{M} =_{\mathbf{df}} \mathsf{record}\ \mathit{MotiveSrc}\ \{T = T; S = D; s = \mathtt{is}/D\} \\ \mathfrak{P} =_{\mathbf{df}} \mathsf{record}\ \mathit{PattSrc}\ \{S = \mathtt{Type}/x; s = \mathtt{isType}/x\} \\ \mathfrak{B} =_{\mathbf{df}} \mathsf{record}\ \mathit{BranchSrc}\ \{\gamma_n(i) = (c_i); (\varrho_i = \varrho_i)_{i\in\{1\dots n\}}; (t_i = t_i)_{i\in\{1\dots n\}}\} \\ \Gamma \vdash \mathfrak{M} \in \mathit{Data}[D,\Delta] \searrow \mathcal{M} \quad y \notin \mathit{DV}(\Gamma) \cup \{\mathtt{isType}/x, x\} \\ \Gamma, \mathit{MuLoc}(D,\mathfrak{M},x,y) \vdash (\mathit{Data}[D,\Delta],\mathfrak{M},\mathfrak{P}) \ni \mathfrak{B} \searrow \mathcal{B}\end{array}}{\begin{array}{l}\Gamma \vdash \mu\ x.\ t\ @T\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\dots n\}} \in T\ t \\ \quad \searrow \mathit{mu} \cdot \mathit{isig}(\mathcal{M})\ \text{-}\mathit{mono}(\mathcal{M})\ t' \cdot \mathcal{M}.T\ (\lambda\,\mathit{MuLoc}(D,\mathfrak{M},x,y).\ \mathit{ctree}(\mathcal{M},\mathcal{B}))\end{array}}$$

$$\frac{\begin{array}{c}\Gamma \vdash t \overset{\dashrightarrow^*}{\in} D' \searrow t'' \\ \mathfrak{M}' =_{\mathbf{df}} \mathsf{record}\ \mathit{MotiveSrc}\ \{T = T; S = D'; s = \mathtt{is}/D'\} \\ \mathfrak{P} =_{\mathbf{df}} \mathsf{record}\ \mathit{PattSrc}\ \{S = \mathtt{Type}/x; s = \mathtt{isType}/x\} \\ \mathfrak{B} =_{\mathbf{df}} \mathsf{record}\ \mathit{BranchSrc}\ \{\gamma_n(i) = (c_i); (\varrho_i = \varrho_i)_{i\in\{1\dots n\}}; (t_i = t_i)_{i\in\{1\dots n\}}\} \\ \Gamma \vdash \mathfrak{M}' \in \mathit{Data}[D',\Delta'] \searrow \mathcal{M}' \quad y \notin \mathit{DV}(\Gamma) \cup \{\mathtt{isType}/x, x\} \\ \Gamma, \mathit{MuLoc}(D',\mathfrak{M}',x,y) \vdash (\mathit{Data}[D',\Delta'],\mathfrak{M}',\mathfrak{P}) \ni \mathfrak{B}' \searrow \mathcal{B}'\end{array}}{\begin{array}{l}\Gamma \vdash \mu\ x.\ t\ @T\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\dots n\}} \in T\ t \\ \quad \searrow \mathit{mu} \cdot \mathit{isig}(\mathcal{M}')\ \text{-}\mathit{mono}(\mathcal{M}')\ t'' \cdot \mathcal{M}'.T\ (\lambda\,\mathit{MuLoc}(D',\mathfrak{M}',x,y).\ \mathit{ctree}(\mathcal{M}',\mathcal{B}'))\end{array}}$$

Similar to the above case. □

Lemma E.14 (Unicity (telescope elaboration))**.** *If* $\vdash \Gamma \searrow \Gamma'$ *then if* $\Gamma \vdash \Delta \searrow \Delta'_1$ *and* $\Gamma \vdash \Delta \searrow \Delta'_2$ *then* $\Delta'_1 = \Delta'_2$.

Proof. By a straightforward induction on the assumed derivation of telescope elaboration, invoking Lemma E.12 and Lemma E.11 as needed. □

APPENDIX F
STATIC AND DYNAMIC SOUNDNESS

F.1 Datatype and Constructor Lookup

Lemma F.1.

- *If* $\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$ *then* $\Gamma \vdash_{\textbf{\textit{data}}} \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$.
- *If* $\Gamma \vdash \mathit{Con}[c] : \mathit{Decl}[D, \Delta] \searrow (\mathcal{D}, i)$, *then* $\Gamma \vdash_{\textbf{\textit{data}}} \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$ *and* c *is the identifier of the* i*th entry of* Δ.

Proof. By induction on the assumed derivation, using weakening. □

Lemma F.2. *If* $\Gamma \vdash \mathit{ConU}[\Delta'] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$ *and* c *is the* i*th entry of* Δ' *then* $\Gamma \vdash \mathit{Con}[c] : \mathit{Decl}[D, \Delta] \searrow (\mathcal{D}, i)$.

Proof. By induction on the assumed derivation. □

F.2 Dynamic Soundness

Proposition F.3. *If* $\Gamma_1; \Gamma_2 \vdash t \searrow t'$ *and* $\vdash \Gamma_1 \searrow \Gamma_1'$ *then* $FV(t') \subseteq DV(\Gamma_1') \cup DV(\Gamma_2)$

Proof. By a straightforward induction on the first assumed derivation. □

Lemma F.4. *If* $\Gamma_1; \Gamma_2 \vdash \lambda\, \varrho.\, t \searrow t'$ *then* $t' = \lambda\, \varrho.\, t''$ *where* $\Gamma_1; \Gamma_2, \left(\underset{i \leq \#\varrho}{\Delta(i) =_{\textbf{\textit{df}}} (\varrho(i) : \mathit{Top})} \right) \vdash t \searrow t''$

Proof. By a straightforward induction on the assumed derivation. □

Lemma F.5. *If* $\Gamma_1; \Gamma_2 \vdash c\ \tau \searrow t'$ *then there exists* $\mathit{Decl}[D, \Delta]$ *with* $\Gamma_1 \vdash \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$, i, *and* τ' *such that* $t' = \mathcal{D}.\theta(i)\ \tau'$ *and* $\#\tau = \#\tau'$

Proof. By induction on the assumed derivation

Case:

$$\frac{\Gamma_1; \Gamma_2 \vdash c\ \tau \searrow t' \quad \Gamma_1; \Gamma_2 \vdash t_1 \searrow t_1'}{\Gamma_1; \Gamma_2 \vdash (c\ \tau)\ t_1 \searrow t'}$$

Invoke the induction hypothesis to obtain $\Gamma_1 \vdash \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$, i, and τ'. Exhibit these and $\tau'(t_1')$

Case:

$$\frac{\Gamma_1 \vdash \mathit{Con}[c] : \mathit{Decl}[D, \Delta] \searrow (\mathcal{D}, i)}{\Gamma_1; \Gamma_2 \vdash c \searrow \mathcal{D}.\theta(i)}$$

Invoke Lemma F.1, and exhibit sequence (). □

Lemma F.6 (Elaboration respects substitution)**.** *If* $\Gamma_1; \Gamma_{21}, x : \mathit{Top}, \Gamma_{22} \vdash t \searrow t'$ *and* $\Gamma_1; \Gamma_{21} \vdash t_1 \searrow t_1'$ *then* $\Gamma_1; \Gamma_{21}, \Gamma_{22} \vdash t[t_1/x] \searrow t'[t_1'/x]$

Proof. By induction on the first assumed derivation.

Case:

$$\frac{y \in DV(\Gamma_1, \Gamma_{21}, x : Top, \Gamma_{22})}{\Gamma_1; \Gamma_{21}, x : Top, \Gamma_{22} \vdash y \searrow y}$$

If $y \neq x$ then we are done. Otherwise, by weakening obtain $\Gamma_1; \Gamma_{21}, \Gamma_{22} \vdash t_1 \searrow t_1'$

Case:

$$\frac{\Gamma_1; \Gamma_{21}, x : Top, \Gamma_{22} x' : Top \vdash t \searrow t'}{\Gamma_1; \Gamma_{21}, x : Top, \Gamma_{22} \vdash \lambda\, x'.\, t \searrow \lambda\, x'.\, t'}$$

By the IH and the definition of substitution.

Case:

$$\frac{\Gamma_1; \Gamma_{21}, x : Top, \Gamma_{22} \vdash t_1 \searrow t_1' \quad \Gamma_1; \Gamma_{21}, x : Top, \Gamma_{22} \vdash t_2 \searrow t_2'}{\Gamma_1; \Gamma_{21}, x : Top, \Gamma_{22} \vdash t_1\ t_2 \searrow t_1'\ t_2'}$$

By the IH (invoked twice) and the definition of substitution.

Case:

$$\frac{\Gamma_1 \vdash Con[c] : Decl[D, \Delta] \searrow (\mathcal{D}, i)}{\Gamma_1; \Gamma_{21}, x : Top, \Gamma_{22} \vdash c \searrow \mathcal{D}.\theta(i)}$$

Immediate, as $x \notin DV(\Gamma_1)$ and so $x \notin FV(\mathcal{D}.\theta(i))$.

Case:

$$\frac{\Gamma_1 \vdash Data[D] : Decl[D, \Delta] \searrow \mathcal{D}}{\Gamma_1; \Gamma_{21}, x : Top, \Gamma_{22} \vdash \texttt{is/}D \searrow |isD|}$$

Immediate, as $x \notin DV(\Gamma_1)$ and so $x \notin FV(isD)$.

Case:

$$\frac{\Gamma_1; \Gamma_{21}, x : Top, \Gamma_{22} \vdash s \searrow s' \quad \Gamma_1 \vdash ConU \left[\underset{i \leq n}{\Delta'(i) =_{\mathbf{df}} (c_i : Top)}\right] : Decl[D, \Delta] \searrow \mathcal{D} \quad (\#\varrho_i = \#|\Delta_i| \text{ where } (c_i : \Pi\, \Delta_i.\, D) \in \Delta)_{i \in \{1 \ldots n\}} \quad (\Gamma_1; \Gamma_{21}, x : Top, \Gamma_{22} \vdash \lambda\, \varrho_i.\, t_i \searrow t_i')_{i \in \{1 \ldots n\}} \quad \underset{i \leq n}{\tau(i) =_{\mathbf{df}} (t_i')}}{\Gamma_1; \Gamma_{21}, x : Top, \Gamma_{22} \vdash \sigma\ s\ \{\ c_i\ \varrho_i \to t_i\}_{i \in \{1 \ldots n\}} \searrow |sigma|\ s'\ (|case(\mathcal{D})|\ \tau)}$$

We note $x \notin FV(|sigma)|$ and $x \notin FV(case(\mathcal{D}))$. Apply the induction hypothesis to the derivations of elaboration of s to s' and $\lambda\, \varrho_i.\, t_i \searrow t_i'$ (for $i \in \{1 \ldots n\}$), then rebuild the derivation to conclude.

Case:

$$\frac{\Gamma_1; \Gamma_{21}, x : Top, \Gamma_{22} \vdash s \searrow s' \quad \Gamma_1 \vdash ConU \left[\underset{i \leq n}{\Delta'(i) =_{\mathbf{df}} (c_i : Top)}\right] : Decl[D, \Delta] \searrow \mathcal{D} \quad (\#\varrho_i = \#|\Delta_i| \text{ where } (c_i : \Pi\, \Delta_i.\, D) \in \Delta)_{i \in \{1 \ldots n\}} \quad (\Gamma_1; \Gamma_{21}, x : Top, \Gamma_{22}, x' : Top \vdash \lambda\, \varrho_i.\, t_i \searrow t_i')_{i \in \{1 \ldots n\}} \quad \underset{i \leq n}{\tau(i) =_{\mathbf{df}} (t_i')}}{\Gamma_1; \Gamma_{21}, x : Top, \Gamma_{22} \vdash \mu\ x'.\ s\ \{c_i\ \varrho_i \to t_i\}_{i \in \{1 \ldots n\}} \searrow |mu|\ s'\ (\lambda\, x'.\, |case(\mathcal{D})|\ \tau)}$$

We note $x \notin FV(|mu)|$ and $x \notin FV(case(\mathcal{D}))$. Apply the induction hypothesis to the derivations of elaboration of s to s' and $\lambda\, \varrho_i.\, t_i \searrow t'_i$ (for $i \in \{1 \dots n\}$), then rebuild the derivation to conclude. □

Proof (of Theorem 5.38). We prove a stronger version by mutual induction: both $\dashrightarrow^*$ and $\dashrightarrow^*_{\mathbf{a}}$ preserve elaboratability.

Case:

$$\frac{\exists\, j \in \{1 \dots n\}.\, c = c_j \quad \#\gamma = \#\varrho_j}{\sigma\ (c\ \gamma)\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\dots n\}} \dashrightarrow_{\mathbf{a}} t_j[\gamma/\varrho_i]}\ \textsc{Sigma}$$

Our assumption is

$$\frac{\begin{array}{c}\Gamma_1; \Gamma_2 \vdash c\ \gamma \searrow |\mathcal{D}.\theta(j)|\ \gamma' \quad \Gamma_1 \vdash ConU \left[\underset{i\le n}{\Delta'(i) =_{\mathbf{df}} (c_i : Top)}\right] : Decl[D, \Delta] \searrow \mathcal{D} \\ (\#\varrho_i = \#|\Delta_i| \text{ where } (c_i : \Pi\, \Delta_i.\, D) \in \Delta)_{i\in\{1\dots n\}} \\ (\Gamma_1; \Gamma_2 \vdash \lambda\, \varrho_i.\, t_i \searrow t'_i)_{i\in\{1\dots n\}} \quad \underset{i\le n}{\tau(i) =_{\mathbf{df}} (t'_i)}\end{array}}{\Gamma_1; \Gamma_2 \vdash \sigma\ (c\ \gamma)\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\dots n\}} \searrow |sigma|\ (|\mathcal{D}.\theta(j)|\ \gamma')\ (|case(\mathcal{D})|\ \tau)}$$

- Our first premise is obtained by inversion, where $\Gamma_1; \Gamma_2 \vdash \gamma(i) \searrow \gamma'(i)$ for all $i \in \{1 \dots \#\gamma\}$ and where $\Gamma_1 \vdash Con[c] : Decl[D, \Delta] \searrow \mathcal{D}$. By Lemma F.1, $\Gamma_1 \vdash_{\mathbf{data}} Decl[D, \Delta] \searrow \mathcal{D}$.
- For the second premise, by Lemma F.2, $\Gamma_1 \vdash_{\mathbf{data}} Decl[D, \Delta] \searrow \mathcal{D}$. By Lemma E.7 we may refer to the same $Decl[D, \Delta]$ and $\mathcal{D}$ throughout (we were already doing this before unicity was invoked to avoid too much name clutter).
- By inversion, $t'_j = \lambda\, \varrho_j.\, t''_j$ where $\Gamma_1; \Gamma_2, \varrho_j \vdash t_j \searrow t''_j$.

To conclude, invoke Lemma F.6 to obtain $\Gamma_1; \Gamma_2 \vdash t_j[\gamma/\varrho_j] \searrow t'_j[\gamma'/\varrho_j]$.

Case:

$$\frac{\begin{array}{c}\exists\, j \in \{1 \dots n\}.\, c = c_j \quad \#\tau = \#\varrho_j \\ y \notin \bigcup_{i\in\{1\dots n\}} (DV(\varrho_i) \cup FV(t_i)) \quad t_{\mathbf{m}} =_{\mathbf{df}} \lambda\, y.\, \mu\ x.\ y\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\dots n\}}\end{array}}{\mu\ x.\ (c\ \tau)\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\dots n\}} \dashrightarrow_{\mathbf{a}} t_j[t_{\mathbf{m}}/x][\tau/\varrho_j]}\ \textsc{Mu}$$

Our assumption is

$$\frac{\begin{array}{c}\Gamma_1; \Gamma_2 \vdash c\ \tau \searrow |\mathcal{D}.\theta(j)|\ \tau' \quad \Gamma_1 \vdash ConU \left[\underset{i\le n}{\Delta'(i) =_{\mathbf{df}} (c_i : Top)}\right] : Decl[D, \Delta] \searrow \mathcal{D} \\ (\#\varrho_i = \#|\Delta_i| \text{ where } (c_i : \Pi\, \Delta_i.\, D) \in \Delta)_{i\in\{1\dots n\}} \\ (\Gamma_1; \Gamma_2, x : Top \vdash \lambda\, \varrho_i.\, t_i \searrow t'_i)_{i\in\{1\dots n\}} \quad \underset{i\le n}{\gamma(i) =_{\mathbf{df}} (t'_i)}\end{array}}{\Gamma_1; \Gamma_2 \vdash \mu\ (c\ \tau).\ t\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\dots n\}} \searrow |mu|\ (|\mathcal{D}.\theta(j)|\ \tau')\ (\lambda\, x.\, |case(\mathcal{D})|\ \gamma)}$$

- Our first premise is obtained by inversion, where $\Gamma_1; \Gamma_2 \vdash \gamma(i) \searrow \gamma'(i)$ for all $i \in \{1 \dots \#\gamma\}$ and where $\Gamma_1 \vdash Con[c] : Decl[D, \Delta] \searrow \mathcal{D}$. By Lemma F.1, $\Gamma_1 \vdash_{\mathbf{data}} Decl[D, \Delta] \searrow \mathcal{D}$.

- For the second premise, by Lemma F.2, $\Gamma_1 \vdash \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$. By Lemma E.7, we may refer to the same $\mathit{Decl}[D, \Delta]$ and $\mathcal{D}$ throughout.
- By inversion, $t'_j = \lambda\, \varrho_j.\, t''_j$ where $\Gamma_1; \Gamma_2, \varrho_j \vdash t_j \searrow t''_j$.
- It is clear we may derive $\Gamma_1; \Gamma_2 \vdash t_{\mathbf{m}} \searrow \lambda\, y.\, |mu|\ y\ (\lambda\, x.\, |\mathit{case}(\mathcal{D})|\ \gamma')$. Call this $t'_{\mathbf{m}}$.

To conclude, invoke Lemma F.6 to obtain $\Gamma_1; \Gamma_2 \vdash t_j[t_{\mathbf{m}}/x][\tau/\varrho_j] \searrow t'_j[t'_{\mathbf{m}}/x][\tau'/\varrho_j]$.

Case:

$$\frac{s_1 \dashrightarrow_{\mathbf{a}} s_2}{\sigma\ s_1\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{a}} \sigma\ s_2\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}}}\ *_1$$

Our assumption is

$$\frac{\begin{array}{c}\Gamma_1; \Gamma_2 \vdash s_1 \searrow s'_1 \quad \Gamma_1 \vdash \mathit{ConU}\left[\underset{i\le n}{\Delta'(i) =_{\mathbf{df}} (c_i : \mathit{Top})}\right] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D} \\ (\#\varrho_i = \#|\Delta_i| \text{ where } (c_i : \Pi\, \Delta_i.\, D) \in \Delta)_{i\in\{1\ldots n\}} \\ (\Gamma_1; \Gamma_2 \vdash \lambda\, \varrho_i.\, t_i \searrow t'_i)_{i\in\{1\ldots n\}} \quad \underset{i\le n}{\tau(i) =_{\mathbf{df}} (t'_i)}\end{array}}{\Gamma_1; \Gamma_2 \vdash \sigma\ s_1\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \searrow |\mathit{sigma}|\ s'_1\ (|\mathit{case}(\mathcal{D})|\ \tau)}$$

By the IH, $\Gamma_1; \Gamma_2 \vdash s_2 \searrow s'_2$ for some s'_2. Exhibit

$$\frac{\begin{array}{c}\Gamma_1; \Gamma_2 \vdash s_2 \searrow s'_2 \quad \Gamma_1 \vdash \mathit{ConU}\left[\underset{i\le n}{\Delta'(i) =_{\mathbf{df}} (c_i : \mathit{Top})}\right] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D} \\ (\#\varrho_i = \#|\Delta_i| \text{ where } (c_i : \Pi\, \Delta_i.\, D) \in \Delta)_{i\in\{1\ldots n\}} \\ (\Gamma_1; \Gamma_2 \vdash \lambda\, \varrho_i.\, t_i \searrow t'_i)_{i\in\{1\ldots n\}} \quad \underset{i\le n}{\tau(i) =_{\mathbf{df}} (t'_i)}\end{array}}{\Gamma_1; \Gamma_2 \vdash \sigma\ s_2\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \searrow |\mathit{sigma}|\ s'_2\ (|\mathit{case}(\mathcal{D})|\ \tau)}$$

Case:

$$\frac{t \dashrightarrow_{\mathbf{a}} t'}{\mu\ x.\ t\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{a}} \mu\ x.\ t'\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}}}\ *_2$$

Similar to the above case.

Case:

$$\frac{t \not\dashrightarrow \quad \underset{i\le n}{\tau(i) =_{\mathbf{df}} (t_i)} \quad \tau \dashrightarrow \tau'}{\sigma\ t\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{a}} \sigma\ t\ \{\ c_i\ \varrho_i \to \tau'(i)\}_{i\in\{1\ldots n\}}}\ *_1$$

Our assumption is

$$\frac{\begin{array}{c}\Gamma_1; \Gamma_2 \vdash t \searrow t' \quad \Gamma_1 \vdash \mathit{ConU}\left[\underset{i\le n}{\Delta'(i) =_{\mathbf{df}} (c_i : \mathit{Top})}\right] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D} \\ (\#\varrho_i = \#|\Delta_i| \text{ where } (c_i : \Pi\, \Delta_i.\, D) \in \Delta)_{i\in\{1\ldots n\}} \\ (\Gamma_1; \Gamma_2 \vdash \lambda\, \varrho_i.\, t_i \searrow t'_i)_{i\in\{1\ldots n\}} \quad \underset{i\le n}{\tau(i) =_{\mathbf{df}} (t'_i)}\end{array}}{\Gamma_1; \Gamma_2 \vdash \sigma\ t\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \searrow |\mathit{sigma}|\ t'\ (|\mathit{case}(\mathcal{D})|\ \tau)}$$

- By an easy inductive argument, exists $1 \leq j \leq n$ such that $t_j \dashrightarrow s$ for some s.
- By inversion of one of our premises, $\Gamma_1; \Gamma_2, \varrho_j \vdash t_j \searrow t'_j$
- By the IH, $\Gamma_1; \Gamma_2, \varrho_j \vdash s \searrow s'$ for some s'.
  Form family $(s_i)_{i \in \{1 \ldots n\}}$ by $s_i = t_i$ when $i \neq j$ and $s_j = s$. Form family $(s'_i)_{i \in \{1 \ldots n\}}$ by $s'_i = t'_i$ when $i \neq j$ and $s'_j = s'$.

Conclude by exhibiting

$$\frac{\begin{array}{c}\Gamma_1; \Gamma_2 \vdash t \searrow t' \quad \Gamma_1 \vdash \mathit{ConU} \left[\underset{i \leq n}{\Delta'(i) =_{\mathbf{df}} (c_i : \mathit{Top})}\right] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D} \\ (\#\varrho_i = \#|\Delta_i| \text{ where } (c_i : \Pi\, \Delta_i.\, D) \in \Delta)_{i \in \{1 \ldots n\}} \\ (\Gamma_1; \Gamma_2 \vdash \lambda\, \varrho_i.\, s_i \searrow s'_i)_{i \in \{1 \ldots n\}} \quad \underset{i \leq n}{\tau(i) =_{\mathbf{df}} (t'_i)}\end{array}}{\Gamma_1; \Gamma_2 \vdash \sigma\ t\ \{\ c_i\ \varrho_i \to s_i\}_{i \in \{1 \ldots n\}} \searrow |\mathit{sigma}|\ t'\ (|\mathit{case}(\mathcal{D})|\ \tau)}$$

Case:

$$\frac{t \not\dashrightarrow \quad \underset{i \leq n}{\tau(i) =_{\mathbf{df}} (t_i)} \quad \tau \dashrightarrow \tau'}{\mu\ x.\ t\ \{c_i\ \varrho_i \to t_i\}_{i \in \{1 \ldots n\}} \dashrightarrow_{\mathbf{a}} \mu\ x.\ t'\ \{c_i\ \varrho_i \to \tau'(i)\}_{i \in \{1 \ldots n\}}}\ *_2$$

Similar to the above case.

Case:

$$\frac{}{(\lambda\, x.\, t_1)\ t_2 \dashrightarrow_{\mathbf{a}} t_1[t_2/x]}\ \textsc{Lambda}$$

This follows by inversion, the IH, and Lemma F.6.

Case:

$$\frac{t_1 \dashrightarrow_{\mathbf{a}} t'_1}{t_1\ t_2 \dashrightarrow_{\mathbf{a}} t'_1\ t_2}\ *_3$$

This follows by inversion and the IH.

Case:

$$\frac{t_1 \not\dashrightarrow \quad t_2 \dashrightarrow t'_2}{t_1\ t_2 \dashrightarrow_{\mathbf{a}} t_1\ t'_2}\ *_3$$

This follows by inversion and the IH.

Case:

$$\frac{t \dashrightarrow t'}{\lambda\, x.\, t \dashrightarrow \lambda\, x.\, t'}$$

This follows by inversion and the IH.

Case:

$$\frac{\Gamma \vdash t \dashrightarrow_{\mathbf{a}} t'}{\Gamma \vdash t \dashrightarrow t'}\ *_4$$

This follows by the IH. □

Proof (of Theorem 5.39). By mutual induction of $\dashrightarrow$ and $\dashrightarrow_{\mathbf{a}}$.

Case:

$$\frac{\exists\, j \in \{1 \ldots n\}.\, c = c_j \quad \#\tau = \#\varrho_j}{\sigma\ (c\ \tau)\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{a}} t_j[\tau/\varrho_i]}\ \textsc{Sigma}$$

Our assumptions are

- By inversion,

$$\frac{\begin{array}{c}\Gamma_1;\Gamma_2 \vdash c\ \tau \searrow t' \quad \Gamma_1 \vdash \mathit{ConU}\left[\underset{i\le n}{\Delta'(i) =_{\mathbf{df}} (c_i : \mathit{Top})}\right] : \mathit{Decl}[D,\Delta] \searrow \mathcal{D} \\ (\#\varrho_i = \#|\Delta_i| \text{ where } (c_i : \Pi\,\Delta_i.\, D) \in \Delta)_{i\in\{1\ldots n\}} \\ (\Gamma_1;\Gamma_2 \vdash \lambda\,\varrho_i.\, t_i \searrow t'_i)_{i\in\{1\ldots n\}} \quad \underset{i\le n}{\gamma(i) =_{\mathbf{df}} (t'_i)}\end{array}}{\Gamma_1;\Gamma_2 \vdash \sigma\ (c\ \tau)\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \searrow |\mathit{sigma}|\ t'\ (|\mathit{case}(\mathcal{D})|\ \gamma)}$$

  - Applying Lemma F.5, $t' = \mathcal{D}'.\theta(j')\ \tau'$ for some $\mathit{Decl}[D,\Delta']$, j', and τ' with $\Gamma_1 \vdash \mathit{Decl}[D,\Delta'] \searrow \mathcal{D}'$. Applying Lemma F.2 with the second premise and then Lemma E.8, we have that these must be $\mathit{Decl}[D,\Delta]$, j, $\mathcal{D}$
  - From the premises $(\Gamma_1;\Gamma_2 \vdash \lambda\,\varrho_i.\, t_i \searrow t'_i)_{i\in\{1\ldots n\}}$, obtain for all i that $t_i = \lambda\,\varrho_i.\, t''_i$ using Lemma F.4.

- $\Gamma_1;\Gamma_2 \vdash t_j[\tau/\varrho_j] \searrow t''$
  We have as a premise $\Gamma_1;\Gamma_2 \vdash \lambda\,\varrho_j.\, t_j \searrow \lambda\,\varrho_j.\, t''_j$, and by an easy inductive proof this yields $\Gamma_1;\Gamma_2,\varrho_j \vdash t_j \searrow t''_j$. By a generalization of Lemma F.6, obtain $\Gamma_1;\Gamma_2 \vdash t_j[\tau/\varrho_j] \searrow t''_j[\tau'/\varrho_j]$. By Lemma E.10, $t'' = t''_j[\tau'/\varrho_j]$

We make observation that $\#\tau = \#\tau' = \#\varrho_j = \#|\Delta_j|$. Conclude with the equational reasoning below.

$$\begin{array}{ll}
 & |\mathit{sigma}\ (\mathcal{D}.\theta(j)\ \tau')\ (\mathit{case}(\mathcal{D})\ \gamma)| \\
= & (|\mathcal{D}.\theta(j)|\ \tau')\ (\lambda\, i.\, \lambda\, \mathit{xs}.\, \mathit{xs}\ (|\mathit{case}(\mathcal{D})|\ \gamma)) \\
\dashrightarrow^* & |\mathit{in}|\ (|\mathcal{D}.\gamma(j)|\ \tau')\ (\lambda\, i.\, \lambda\, \mathit{xs}.\, \mathit{xs}\ (|\mathit{case}(\mathcal{D})|\ \gamma)) \\
\dashrightarrow^* & (\lambda\, i.\, \lambda\, \mathit{xs}.\, \mathit{xs}\ (|\mathit{case}(\mathcal{D})|\ \gamma))\ (\lambda\, x.\, x\ (\lambda\, i.\, \lambda\, \mathit{xs}.\, \mathit{xs}\ (|\mathit{case}(\mathcal{D})|\ \gamma)))\ (\lambda\, f.\, f\ (|\mathcal{D}.\gamma(j)|\ \tau')) \\
\dashrightarrow^* & (\lambda\, f.\, f\ (|\mathcal{D}.\gamma(j)|\ \tau'))\ (|\mathit{case}(\mathcal{D})|\ \gamma) \\
\dashrightarrow & |\mathit{case}(\mathcal{D})|\ \gamma\ (|\mathcal{D}.\gamma|\ \tau') \\
\dashrightarrow^* & \gamma(j)\ \tau' \\
= & \lambda\,\varrho_j.\, t''_j\ \tau' \\
\dashrightarrow^* & t''_j[\tau'/\varrho_j]
\end{array}$$

Case:

$$\frac{\begin{array}{c}\exists\, j \in \{1 \ldots n\}.\, c = c_j \quad \#\tau = \#\varrho_j \\ y \notin \bigcup_{i\in\{1\ldots n\}} (DV(\varrho_i) \cup FV(t_i)) \quad t_{\mathbf{m}} =_{\mathbf{df}} \lambda\, y.\, \mu\ x.\ y\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}}\end{array}}{\mu\ x.\ (c\ \tau)\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{a}} t_j[t_{\mathbf{m}}/x][\tau/\varrho_j]}\ \textsc{Mu}$$

Our assumptions are

- By inversion,

$$\frac{\Gamma_1;\Gamma_2 \vdash c\ \tau \searrow t' \quad \Gamma_1 \vdash \mathit{ConU}\left[\underset{i\le n}{\Delta'(i) =_{\mathbf{df}} (c_i : \mathit{Top})}\right] : \mathit{Decl}[D,\Delta] \searrow \mathcal{D} \quad (\#\varrho_i = \#|\Delta_i| \text{ where } (c_i : \Pi\,\Delta_i.\,D) \in \Delta)_{i\in\{1\ldots n\}} \quad (\Gamma_1;\Gamma_2, x : \mathit{Top} \vdash \lambda\,\varrho_i.\,t_i \searrow t'_i)_{i\in\{1\ldots n\}} \quad \underset{i\le n}{\gamma(i) =_{\mathbf{df}} (t'_i)}}{\Gamma_1;\Gamma_2 \vdash \mu\ x.\ (c\ \tau)\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \searrow |mu|\ t'\ (\lambda\,x.\,|\mathit{case}(\mathcal{D})|\ \gamma)}$$

  - Applying Lemma F.5, $t' = \mathcal{D}'.\theta(j')\ \tau'$ for some $\mathit{Decl}[D,\Delta']$, j', and τ' with $\Gamma_1 \vdash \mathit{Decl}[D,\Delta'] \searrow \mathcal{D}'$. Applying Lemma F.2 with the second premise and then Lemma E.10, we have that these must be $\mathit{Decl}[D,\Delta]$, j, $\mathcal{D}$
  - From the premises $(\Gamma_1;\Gamma_2, x : \mathit{Top} \vdash \lambda\,\varrho_i.\,t_i \searrow t'_i)_{i\in\{1\ldots n\}}$, obtain for all i that $t_i = \lambda\,\varrho_i.\,t''_i$ using Lemma F.4.

- $\Gamma_1;\Gamma_2 \vdash t_j[t_{\mathbf{m}}/x][\tau/\varrho_j] \searrow t''$

  We have as a premise $\Gamma_1;\Gamma_2, x : \mathit{Top} \vdash \lambda\,\varrho_j.\,t_j \searrow \lambda\,\varrho_j.\,t''_j$, and by an easy inductive proof this yields $\Gamma_1;\Gamma_2, x : \mathit{Top}, \varrho_j \vdash t_j \searrow t''_j$. By a generalization of Lemma F.6 we obtain $\Gamma_1;\Gamma_2, x : \mathit{Top} \vdash t_j[\tau/\varrho_j] \searrow t''_j[\tau'/\varrho_j]$.

  We now construct the following derivation for $t_{\mathbf{m}}$.

$$\frac{\dots}{\dfrac{\Gamma_1;\Gamma_2, y : \mathit{Top} \vdash \mu\ x.\ y\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \searrow |mu|\ y\ (\lambda\,x.\,|\mathit{case}(\mathcal{D})|\ \gamma)}{\Gamma_1;\Gamma_2 \vdash \lambda\,y.\,\mu\ x.\ y\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \searrow \lambda\,y.\,|mu|\ y\ (\lambda\,x.\,|\mathit{case}(\mathcal{D})|\ \gamma)}}$$

  where the elided premises are readily available from our inverted elaboration derivation for the original μ-expression (or, in the case of the scrutinee, given by the rule for variables). Call the expression elaborated above $t'_{\mathbf{m}}$

  By Lemma E.10, $t'' = t''_j[\lambda\,y.\,|mu|\ y\ (\lambda\,x.\,|\mathit{case}(\mathcal{D})|\ \gamma)/x][\tau'/\varrho_j]$

We make observation that $\#\tau = \#\tau' = \#\varrho_j = \#|\Delta_j|$. Conclude with the equational reasoning below.

$$\begin{array}{rl}
 & |mu\ (\mathcal{D}.\theta(j)\ \tau')\ (\lambda\,x.\,\mathit{case}(\mathcal{D})\ \gamma)| \\
= & |(\mathcal{D}.\theta(j)\ \tau')\ \lambda\,x.\,\lambda\,xs.\,xs\ (\mathit{case}(\mathcal{D})\ \gamma)| \\
\dashrightarrow^* & |in\ (\mathcal{D}.\gamma(j)\ \tau')\ \lambda\,x.\,\lambda\,xs.\,xs\ (\mathit{case}(\mathcal{D})\ \gamma)| \\
\dashrightarrow^* & |(\lambda\,x.\,\lambda\,xs.\,xs\ (\mathit{case}(\mathcal{D})\ \gamma))\ (\lambda\,y.\,y\ (\lambda\,x.\,\lambda\,xs.\,xs\ (\mathit{case}(\mathcal{D})\ \gamma)))\ (\lambda\,f.\,f\ (\mathcal{D}.\gamma(j)\ \tau'))| \\
= & |(\lambda\,x.\,\lambda\,xs.\,xs\ (\mathit{case}(\mathcal{D})\ \gamma))\ t'_{\mathbf{m}}\ (\lambda\,f.\,f\ (\mathcal{D}.\gamma(j)\ \tau'))| \\
\dashrightarrow^* & (\lambda\,f.\,f\ (\mathcal{D}.\gamma(j)\ \tau'))\ (\mathit{case}(\mathcal{D})\ \gamma[t'_{\mathbf{m}}/x]) \\
\dashrightarrow & \mathit{case}(\mathcal{D})\ \gamma[t'_{\mathbf{m}}/x]\ (\mathcal{D}.\gamma(j)\ \tau') \\
\dashrightarrow^* & \gamma(j)[t'_{\mathbf{m}}/x]\ \tau' \\
= & (\lambda\,\varrho_j.\,t''_j[t'_{\mathbf{m}}/x])\ \tau' \\
\dashrightarrow^* & t''_j[t'_{\mathbf{m}}/x][\varrho_j/\tau']
\end{array}$$

Case:

$$\frac{t \dashrightarrow_{\mathbf{a}} t'}{\sigma\ t\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{a}} \sigma\ t'\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}}}\ *_1$$

Our assumptions are

- By inversion,

$$\frac{\Gamma_1;\Gamma_2 \vdash t \searrow t'' \quad \Gamma_1 \vdash \mathit{ConU}\left[\underset{i\leq n}{\Delta'(i) =_{\mathbf{df}} (c_i : \mathit{Top})}\right] : \mathit{Decl}[D,\Delta] \searrow \mathcal{D} \quad (\#\varrho_i = \#|\Delta_i| \text{ where } (c_i : \Pi\,\Delta_i.\, D) \in \Delta)_{i\in\{1\ldots n\}} \quad (\Gamma_1;\Gamma_2 \vdash \lambda\,\varrho_i.\, t_i \searrow t'_i)_{i\in\{1\ldots n\}} \quad \underset{i\leq n}{\tau(i) =_{\mathbf{df}} (t'_i)}}{\Gamma_1;\Gamma_2 \vdash \sigma\ t\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \searrow |\mathit{sigma}|\ t''\ (|\mathit{case}(\mathcal{D})|\ \tau)}$$

- By inversion,

$$\frac{\Gamma_1;\Gamma_2 \vdash t' \searrow t''' \quad \Gamma_1 \vdash \mathit{ConU}\left[\underset{i\leq n}{\Delta'(i) =_{\mathbf{df}} (c_i : \mathit{Top})}\right] : \mathit{Decl}[D,\Delta] \searrow \mathcal{D} \quad (\#\varrho_i = \#|\Delta_i| \text{ where } (c_i : \Pi\,\Delta_i.\, D) \in \Delta)_{i\in\{1\ldots n\}} \quad (\Gamma_1;\Gamma_2 \vdash \lambda\,\varrho_i.\, t_i \searrow t'_i)_{i\in\{1\ldots n\}} \quad \underset{i\leq n}{\tau(i) =_{\mathbf{df}} (t'_i)}}{\Gamma_1;\Gamma_2 \vdash \sigma\ t'\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \searrow |\mathit{sigma}|\ t''\ (|\mathit{case}(\mathcal{D})|\ \tau)}$$

We have $\Gamma_1;\Gamma_2 \vdash t \searrow t''$, $t \dashrightarrow_{\mathbf{a}} t'$, and $\Gamma_1;\Gamma_2 \vdash t' \searrow t'''$, so by the induction hypothesis, $t'' \cong t'''$. It is then easy to derive $|\mathit{sigma}|\ t''\ (|\mathit{case}(\mathcal{D})\ \gamma|) \cong |\mathit{sigma}|\ t'''$

Case:

$$\frac{t \dashrightarrow_{\mathbf{a}} t'}{\mu\ x.\ t\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{a}} \mu\ x.\ t'\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}}}\ *_2$$

Similar to the above case.

Case:

$$\frac{t \not\dashrightarrow \quad \underset{i\leq n}{\tau(i) =_{\mathbf{df}} (t_i)} \quad \tau \dashrightarrow \tau'}{\sigma\ t\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{a}} \sigma\ t\ \{\ c_i\ \varrho_i \to \tau'(i)\}_{i\in\{1\ldots n\}}}\ *_1$$

Our assumptions are

- By inversion,

$$\frac{\Gamma_1;\Gamma_2 \vdash t \searrow t' \quad \Gamma_1 \vdash \mathit{ConU}\left[\underset{i\leq n}{\Delta'(i) =_{\mathbf{df}} (c_i : \mathit{Top})}\right] : \mathit{Decl}[D,\Delta] \searrow \mathcal{D} \quad (\#\varrho_i = \#|\Delta_i| \text{ where } (c_i : \Pi\,\Delta_i.\, D) \in \Delta)_{i\in\{1\ldots n\}} \quad (\Gamma_1;\Gamma_2 \vdash \lambda\,\varrho_i.\, t_i \searrow t'_i)_{i\in\{1\ldots n\}} \quad \underset{i\leq n}{\gamma(i) =_{\mathbf{df}} (t'_i)}}{\Gamma_1;\Gamma_2 \vdash \sigma\ t\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \searrow |\mathit{sigma}|\ t'\ (|\mathit{case}(\mathcal{D})|\ \gamma)}$$

- By inversion,

$$\frac{\Gamma_1;\Gamma_2 \vdash t \searrow t' \quad \Gamma_1 \vdash \mathit{ConU}\left[\underset{i\leq n}{\Delta'(i) =_{\mathbf{df}} (c_i : \mathit{Top})}\right] : \mathit{Decl}[D,\Delta] \searrow \mathcal{D} \quad (\#\varrho_i = \#|\Delta_i| \text{ where } (c_i : \Pi\,\Delta_i.\, D) \in \Delta)_{i\in\{1\ldots n\}} \quad (\Gamma_1;\Gamma_2 \vdash \lambda\,\varrho_i.\, \tau'(i) \searrow t''_i)_{i\in\{1\ldots n\}} \quad \underset{i\leq n}{\gamma'(i) =_{\mathbf{df}} (t''_i)}}{\Gamma_1;\Gamma_2 \vdash \sigma\ t\ \{\ c_i\ \varrho_i \to \tau'(i)\}_{i\in\{1\ldots n\}} \searrow |\mathit{sigma}|\ t'\ (|\mathit{case}(\mathcal{D})|\ \gamma')}$$

We make a few observations.

- By Lemma F.4 and E.9, $t'_i = \lambda\, \varrho_i.\, s'_i$ and $t''_i = \lambda\, \varrho_i.\, s''_i$ for all i and for some families s'_i, s''_i.

We have $(\Gamma_1; \Gamma_2 \vdash \lambda\, \varrho_i.\, \tau(i) \searrow \lambda\, \varrho_i.\, s'_i)_{i \in \{1 \ldots n\}}$, which by an easy inductive proof yields $(\Gamma_1; \Gamma_2, \varrho_i \vdash \tau(i) \searrow s''_i)_{i \in \{1 \ldots n\}}$. Similarly, we have $(\Gamma_1; \Gamma_2, \varrho_i \vdash \tau'(i) \searrow s''_i)_{i \in \{1 \ldots n\}}$. Finally, with $\tau \dashrightarrow \tau'$ we invoke the induction hypothesis to obtain $(s'_i \cong s''_i)_{i \in \{1 \ldots n\}}$, from which we obtain $(\lambda\, \varrho_i.\, s'_i \cong \lambda\, \varrho_i.\, s''_i)_{i \in \{1 \ldots n\}}$. This means $\gamma \cong \gamma'$, and from this it is straightforward to obtain $|sigma|\ t'\ (|case(\mathcal{D})|\ \gamma) \cong |sigma|\ t'\ (|case(\mathcal{D})|\ \gamma')$.

Case:

$$\frac{t \not\searrow \quad \underset{i \le n}{\tau(i) =_{\mathbf{df}} (t_i)} \quad \tau \dashrightarrow \tau'}{\mu\ x.\ t\ \{c_i\ \varrho_i \to t_i\}_{i \in \{1 \ldots n\}} \dashrightarrow_{\mathbf{a}} \mu\ x.\ t'\ \{c_i\ \varrho_i \to \tau'(i)\}_{i \in \{1 \ldots n\}}}\ *_2$$

Similar to the above case.

Case:

$$\frac{t \dashrightarrow t'}{(t)\tau \dashrightarrow (t')\tau}$$

Our assumptions are

- $(\Gamma_1; \Gamma_{2i} \vdash ((t)\tau)(i) \searrow t'_i)_{i \in \{1 \ldots 1 + \#\tau\}}$
  So in particular $\Gamma_1; \Gamma_{21} \vdash t \searrow t'_1$
- $(\Gamma_1; \Gamma_{2i} \vdash ((t')\tau)(i) \searrow t''_i)_{i \in \{1 \ldots 1 + \#\tau\}}$
  So in particular $\Gamma_1; \Gamma_{21} \vdash t' \searrow t''_1$

We observe that by Lemma E.10, for $i \in \{2 \ldots 1 + \#\tau\}$ we have $t'_i = t''_i$., using the induction hypothesis, $t'_1 \cong t''_1$. We therefore conclude $(t'_i \cong t''_i)_{i \in \{1 \ldots 1 + \#\tau\}}$

Case:

$$\frac{t \not\searrow \quad \tau \dashrightarrow \tau'}{(t)\tau \dashrightarrow (t)\tau'}$$

Our assumptions are

- $(\Gamma_1; \Gamma_{2i} \vdash ((t)\tau)(i) \searrow t'_i)_{i \in \{1 \ldots 1 + \#\tau\}}$
  From this we obtain (by fudging our indexed family) $\left(\Gamma_1; \Gamma_{2(i+1)} \vdash \tau(i) \searrow t'_{i+1}\right)_{i \in \{1 \ldots \#\tau\}}$.
- $(\Gamma_1; \Gamma_{2i} \vdash ((t')\tau)(i) \searrow t''_i)_{i \in \{1 \ldots 1 + \#\tau\}}$
  From this we obtain $\left(\Gamma_1; \Gamma_{2(i+1)} \vdash \tau'(i) \searrow t''_{i+1} \vdash\right)_{i \in \{1 \ldots \#\tau\}}$
  By the induction hypothesis, $\tau \cong \tau'$, so we obtain $(t)\tau \cong (t)\tau'$.

Case:

$$\frac{}{(\lambda\, x.\, t_1)\ t_2 \dashrightarrow_{\mathbf{a}} t_1[t_2/x]}\ \textsc{Lambda}$$

Our assumptions are

- By inversion,

$$\frac{\dfrac{\Gamma_1; \Gamma_2, x : Top \vdash t_1 \searrow t'_1}{\Gamma_1; \Gamma_2 \vdash \lambda\, x.\, t_1 \searrow \lambda\, x.\, t'_1} \quad \Gamma_1; \Gamma_2 \vdash t_2 \searrow t'_2}{\Gamma_1; \Gamma_2 \vdash (\lambda\, x.\, t_1)\ t_2 \searrow (\lambda\, x.\, t'_1)\ t'_2}$$

By Lemma F.6, $\Gamma_1; \Gamma_2 \vdash t_1[t_2/x] \searrow t_1'[t_2'/x]$

- $\Gamma_1; \Gamma_2 \vdash t_1[t_2/x] \searrow t''$
  By Lemma E.10, $t'' = t_1'[t_2'/x]$

It is clear $(\lambda\, x.\, t_1')\ t_2' \cong t_1'[t_2'/x]$, so we conclude.

Case:

$$\frac{t_1 \dashrightarrow_{\mathbf{a}} t_1'}{t_1\ t_2 \dashrightarrow_{\mathbf{a}} t_1'\ t_2}\ *_3$$

By a straightforward application of inversion and the induction hypothesis.

Case:

$$\frac{t_1 \not\dashrightarrow \quad t_2 \dashrightarrow t_2'}{t_1\ t_2 \dashrightarrow_{\mathbf{a}} t_1\ t_2'}\ *_3$$

By a straightforward application of inversion and the induction hypothesis. □

Proof (of Corollary 5.41). By definition of $\cong$, $t_1 \dashrightarrow^* t_3$, $t_2 \dashrightarrow^* t_4$, and t_3 and t_4 are η-convertible. By induction on the derivations of reduction, repeatedly invoking Theorem 5.38 and Theorem 5.39, we have that there is some t_3' and t_4' such that $\Gamma_1; \Gamma_2 \vdash t_3 \searrow t_3'$, $\Gamma_1; \Gamma_2 \vdash t_4 \searrow t_4'$, and $t_1' =_{\beta\eta} t_3'$ and $t_2' =_{\beta\eta} t_4'$. By Proposition 5.40, t_3' and t_4' are η-convertible. It therefore follows that $t_1' =_{\beta\eta} t_2'$. □

Proof (of Theorem 5.42). By induction on the assumed derivation of reduction.

Case:

$$\frac{\exists\, j \in \{1 \ldots n\}.\, c = c_j \quad \#\tau = \#\varrho_j}{\sigma\ (c\ \tau)\ \{\ c_i\ \varrho_i \to t_i\}_{i \in \{1 \ldots n\}} \dashrightarrow_{\mathbf{cbn}} t_j[\tau/\varrho_i]}\ \textsc{Sigma}$$

Our assumptions are

- By inversion,

$$\frac{\begin{array}{c}\Gamma_1; \Gamma_2 \vdash c\ \tau \searrow t' \quad \Gamma_1 \vdash \mathit{ConU}\left[\underset{i \le n}{\Delta'(i) =_{\mathbf{df}} (c_i : \mathit{Top})}\right] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D} \\ (\#\varrho_i = \#|\Delta_i| \text{ where } (c_i : \Pi\, \Delta_i.\, D) \in \Delta)_{i \in \{1 \ldots n\}} \\ (\Gamma_1; \Gamma_2 \vdash \lambda\, \varrho_i.\, t_i \searrow t_i')_{i \in \{1 \ldots n\}} \quad \underset{i \le n}{\gamma(i) =_{\mathbf{df}} (t_i')}\end{array}}{\Gamma_1; \Gamma_2 \vdash \sigma\ (c\ \tau)\ \{\ c_i\ \varrho_i \to t_i\}_{i \in \{1 \ldots n\}} \searrow |\mathit{sigma}|\ t'\ (|\mathit{case}(\mathcal{D})|\ \gamma)}$$

  - Applying Lemma F.5, $t' = \mathcal{D}'.\theta(j')\ \tau'$ for some $\mathit{Decl}[D, \Delta']$, j', and τ' with $\Gamma_1 \vdash \mathit{Decl}[D, \Delta'] \searrow \mathcal{D}'$. Applying Lemma F.2 with the second premise and then Lemma E.8, we have that these must be $\mathit{Decl}[D, \Delta]$, j, $\mathcal{D}$
  - From the premises $(\Gamma_1; \Gamma_2 \vdash \lambda\, \varrho_i.\, t_i \searrow t_i')_{i \in \{1 \ldots n\}}$, obtain for all i that $t_i = \lambda\, \varrho_i.\, t_i''$ using Lemma F.4.

- $\Gamma_1; \Gamma_2 \vdash t_j[\tau/\varrho_j] \searrow t''$
  We have as a premise $\Gamma_1; \Gamma_2 \vdash \lambda\, \varrho_j.\, t_j \searrow \lambda\, \varrho_j.\, t_j''$, and by an easy inductive proof this yields $\Gamma_1; \Gamma_2, \varrho_j \vdash t_j \searrow t_j''$. By a generalization of Lemma F.6, obtain $\Gamma_1; \Gamma_2 \vdash t_j[\tau/\varrho_j] \searrow t_j''[\tau'/\varrho_j]$. By Lemma E.10, $t'' = t_j''[\tau'/\varrho_j]$

We make observation that $\#\tau = \#\tau' = \#\varrho_j = \#|\Delta_j|$. Conclude with the equational reasoning below.

$$
\begin{array}{ll}
 & |sigma\ (\mathcal{D}.\theta(j)\ \tau')\ (case(\mathcal{D})\ \gamma)| \\
= & (\mathcal{D}.\theta(j)\ \tau')\ (\lambda\, i.\, \lambda\, x.\, x\ (|case(\mathcal{D})|\ \gamma)) \\
\dashrightarrow^* & (in\ (\mathcal{D}.\gamma(j)\ \tau'))\ (\lambda\, i.\, \lambda\, x.\, x\ (|case(\mathcal{D})|\ \gamma)) \\
= & (\lambda\, x.\, \lambda\, a.\, a\ (\lambda\, y.\, y\ a)\ (\lambda\, f.\, f\ x))\ (\mathcal{D}.\gamma(j)\ \tau'))\ (\lambda\, i.\, \lambda\, x.\, x\ (|case(\mathcal{D})|\ \gamma)) \\
\dashrightarrow^* & (\lambda\, i.\, \lambda\, x.\, x\ (|case(\mathcal{D})|\ \gamma))\ (\lambda\, y.\, y\ (\lambda\, i.\, \lambda\, x.\, x\ (|case(\mathcal{D})|\ \gamma)))\ (\lambda\, f.\, f\ (\mathcal{D}.\gamma(j)\ \tau')) \\
\dashrightarrow & (\lambda\, x.\, x\ (|case(\mathcal{D})|\ \gamma))\ (\lambda\, f.\, f\ (\mathcal{D}.\gamma(j)\ \tau')) \\
\dashrightarrow & (\lambda\, f.\, f\ (\mathcal{D}.\gamma(j)\ \tau'))\ (|case(\mathcal{D})|\ \gamma) \\
\dashrightarrow & |case(\mathcal{D})|\ \gamma\ (\mathcal{D}.\gamma(j)\ \tau') \\
\dashrightarrow^* & \gamma(j)\ \tau' \\
\dashrightarrow^* & t''_j[\tau'/\varrho_j]
\end{array}
$$

Case:

$$\frac{\begin{array}{c}\exists\, j \in \{1 \ldots n\}.\, c = c_j \quad \#\tau = \#\varrho_j \\ y \notin \bigcup_{i\in\{1\ldots n\}} (DV(\varrho_i) \cup FV(t_i)) \quad t_{\mathbf{m}} =_{\mathbf{df}} \lambda\, y.\, \mu\ x.\ y\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}}\end{array}}{\mu\ x.\ (c\ \tau)\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{cbn}} t_j[t_{\mathbf{m}}/x][\tau/\varrho_j]}\ \textsc{Mu}$$

Our assumptions are

- By inversion,

$$\frac{\begin{array}{c}\Gamma_1; \Gamma_2 \vdash c\ \tau \searrow t' \quad \Gamma_1 \vdash ConU \left[\underset{i\le n}{\Delta'(i) =_{\mathbf{df}} (c_i : Top)}\right] : Decl[D, \Delta] \searrow \mathcal{D} \\ (\#\varrho_i = \#|\Delta_i| \text{ where } (c_i : \Pi\, \Delta_i.\, D) \in \Delta)_{i\in\{1\ldots n\}} \\ (\Gamma_1; \Gamma_2, x : Top \vdash \lambda\, \varrho_i.\, t_i \searrow t'_i)_{i\in\{1\ldots n\}} \quad \underset{i\le n}{\gamma(i) =_{\mathbf{df}} (t'_i)}\end{array}}{\Gamma_1; \Gamma_2 \vdash \mu\ x.\ (c\ \tau)\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \searrow |mu\ t'\ (\lambda\, x.\, case(\mathcal{D})\ \gamma)|}$$

  - Applying Lemma F.5, $t' = \mathcal{D}'.\theta(j')\ \tau'$ for some $Decl[D, \Delta']$, j', and τ' with $\Gamma_1 \vdash Decl[D, \Delta'] \searrow \mathcal{D}'$. Applying Lemma F.2 with the second premise and then Lemma E.8, we have that these must be $Decl[D, \Delta]$, j, $\mathcal{D}$
  - From the premises $(\Gamma_1; \Gamma_2 \vdash \lambda\, \varrho_i.\, t_i \searrow t'_i)_{i\in\{1\ldots n\}}$, obtain for all i that $t_i = \lambda\, \varrho_i.\, t''_i$ using Lemma F.4.

- $\Gamma_1; \Gamma_2 \vdash t_j[t_{\mathbf{m}}/x][\tau/\varrho_j] \searrow t''$.
  We have as a premise $\Gamma_1; \Gamma_2, x : Top \vdash \lambda\, \varrho_j.\, t_j \searrow t''_j$, and by an easy inductive proof this yields $\Gamma_1; \Gamma_2, x : Top, \varrho_j \vdash t_j \searrow t''_j$. We construct the derivation $\Gamma_1; \Gamma_2 \vdash t_{\mathbf{m}} \searrow t'_{\mathbf{m}}$, where $t'_{\mathbf{m}} = \lambda\, y.\, |mu\ y\ (\lambda\, x.\, case(\mathcal{D})\ \gamma)|$ By a generalization of Lemma F.6, obtain $\Gamma_1; \Gamma_2 \vdash t_j[t_{\mathbf{m}}/x][\tau/\varrho_j] \searrow t''_j[t'_{\mathbf{m}}/x][\tau'/\varrho_j]$. By Lemma E.10, $t'' = t''_j[t'_{\mathbf{m}}/x][\tau'/\varrho_j]$

We make observation that $\#\tau = \#\tau' = \#\varrho_j = \#|\Delta_j|$. Conclude with the equational reasoning below.

$$
\begin{array}{ll}
 & |mu\ (\mathcal{D}.\theta(j)\ \tau')\ (\lambda\, x.\, case(\mathcal{D})\ \gamma)| \\
= & |\mathcal{D}.\theta(j)\ \tau'\ \lambda\, x.\, \lambda\, xs.\, xs\ (case(\mathcal{D})\ \gamma) \\
\dashrightarrow^* & |in|\ (|\mathcal{D}.\gamma(j)|\ \tau')\ \lambda\, x.\, \lambda\, xs.\, xs\ (case(\mathcal{D})\ \gamma) \\
\dashrightarrow^* & (\lambda\, x.\, \lambda\, xs.\, xs\ (case(\mathcal{D})\ \gamma))\ t'_{\mathbf{m}}\ (\lambda\, f.\, f\ (|\mathcal{D}.\gamma(j)|\ \tau')) \\
\dashrightarrow^* & (\lambda\, f.\, f\ (|\mathcal{D}.\gamma(j)|\ \tau'))\ (case(\mathcal{D})\ \gamma[t'_{\mathbf{m}}/x]) \\
\dashrightarrow & case(\mathcal{D})\ \gamma[t'_{\mathbf{m}}/x]\ (|\mathcal{D}.\gamma(j)|\ \tau') \\
\dashrightarrow^* & \gamma(j)[t'_{\mathbf{m}}][\varrho_j/\tau'] \\
= & t''_j[t'_{\mathbf{m}}][\varrho_j/\tau']
\end{array}
$$

Case:

$$
\frac{s_1 \dashrightarrow_{\mathbf{cbn}} s_2}{\sigma\ s_1\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{cbn}} \sigma\ s_2\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}}}\ *_1
$$

Our assumptions are

- By inversion,

$$
\frac{\begin{array}{c}\Gamma_1;\Gamma_2 \vdash s_1 \searrow s'_1 \quad \Gamma_1 \vdash ConU\left[\underset{i\le n}{\Delta'(i) =_{\mathbf{df}} (c_i : Top)}\right] : Decl[D,\Delta] \searrow \mathcal{D} \\ (\#\varrho_i = \#|\Delta_i| \text{ where } (c_i : \Pi\, \Delta_i.\, D) \in \Delta)_{i\in\{1\ldots n\}} \\ (\Gamma_1;\Gamma_2 \vdash \lambda\, \varrho_i.\, t_i \searrow t'_i)_{i\in\{1\ldots n\}} \quad \underset{i\le n}{\tau(i) =_{\mathbf{df}} (t'_i)}\end{array}}{\Gamma_1;\Gamma_2 \vdash \sigma\ s_1\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \searrow |sigma\ s'_1\ (case(\mathcal{D})\ \tau)|}
$$

- By inversion,

$$
\frac{\begin{array}{c}\Gamma_1;\Gamma_2 \vdash s_2 \searrow s'_2 \quad \Gamma_1 \vdash ConU\left[\underset{i\le n}{\Delta'(i) =_{\mathbf{df}} (c_i : Top)}\right] : Decl[D,\Delta] \searrow \mathcal{D} \\ (\#\varrho_i = \#|\Delta_i| \text{ where } (c_i : \Pi\, \Delta_i.\, D) \in \Delta)_{i\in\{1\ldots n\}} \\ (\Gamma_1;\Gamma_2 \vdash \lambda\, \varrho_i.\, t_i \searrow t'_i)_{i\in\{1\ldots n\}} \quad \underset{i\le n}{\tau(i) =_{\mathbf{df}} (t'_i)}\end{array}}{\Gamma_1;\Gamma_2 \vdash \sigma\ s_2\ \{\ c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \searrow |sigma\ s'_2\ (case(\mathcal{D})\ \tau)|}
$$

So by the IH, $s'_1 \dashrightarrow^* s'_2$. Conclude with the following reduction reasoning.

$$
\begin{array}{ll}
 & |sigma\ s'_1\ (case(\mathcal{D})\ \gamma)| \\
= & s'_1\ (\lambda\, i.\, \lambda\, xs.\, xs\ (case(\mathcal{D})\ \gamma)) \\
\dashrightarrow^* & s'_2\ (\lambda\, i.\, \lambda\, xs.\, xs\ (case(\mathcal{D})\ \gamma)) \\
= & |sigma\ s'_2\ (case(\mathcal{D})\ \gamma)|
\end{array}
$$

Case:

$$
\frac{t \dashrightarrow_{\mathbf{cbn}} t'}{\mu\ x.\ t\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}} \dashrightarrow_{\mathbf{cbn}} \mu\ x.\ t'\ \{c_i\ \varrho_i \to t_i\}_{i\in\{1\ldots n\}}}\ *_2
$$

Similar to the above case. □

F.3 Static Soundness

F.3.1 Case Tree and Motive Soundness

Proof (of Proposition 5.27).

1.

$$\dfrac{\overline{\Gamma' \vdash isig(\mathcal{M}) \cdot \mathcal{B}.S : \star}\ \text{a} \qquad \dfrac{\dfrac{}{\Gamma', y : isig(\mathcal{M}) \cdot \mathcal{B}.S \vdash \mathcal{M}.T : \mathcal{D}.D \to \star}\ \text{b} \quad \mathcal{J}}{\begin{array}{l}\mathcal{M}.T\ (in \cdot isig(\mathcal{M})\ \text{-} \\ \quad (elimCast\ \text{-}(toFD \cdot isig(\mathcal{M}) \\ \quad\ \text{-}\mathcal{M}.mono \cdot \mathcal{B}.S\ \text{-}\mathcal{B}.s)\ \mathfrak{M}.y))\end{array} : \star}}{\Gamma' \vdash mot(\mathcal{M}, \mathcal{B}) : \Pi\, \mathcal{M}.y : isig(\mathcal{M}) \cdot \mathcal{B}.S.\, \star}$$

where $\mathcal{J}$ is the derivation (letting $\Gamma'' =_{\mathbf{df}} \Gamma', y : isig(\mathcal{M}) \cdot \mathcal{B}.S$)

$$\dfrac{}{\Gamma'' \vdash in \cdot isig(\mathcal{M})\ \text{-}(elimCast\ \text{-}(toFD \cdot isig(\mathcal{M})\ \text{-}\mathcal{M}.mono \cdot \mathcal{B}.S\ \text{-}\mathcal{B}.s)\ \mathfrak{M}.y)}\ \text{c}$$

(a) By Proposition 4.44, $\Gamma' \vdash isig(\mathcal{M}) : \star \to \star$, and by assumption $\Gamma' \vdash \mathcal{B}.S : \star$.

(b) By assumption. Actually, the kind of $\mathcal{M}.T$ is $T \to \star$ for some $T \cong \mathcal{D}.D$, but by Lemma 17 of Stump and Jenkins [83] we may exchange one type checking derivation with another when the types are convertible.

(c) By assumption and weakening, $\Gamma'' \vdash \mathcal{B}.S : \star$ and $\Gamma'' \vdash \mathcal{B}.s : IsD \cdot isig(\mathcal{M}) \cdot \mathcal{B}.S$. By assumption, $\Gamma \vdash Mono \cdot isig(\mathcal{M}) \ni \mathcal{M}.mono$. Therefore, $\Gamma'' \vdash toFD \cdot isig(\mathcal{M})\ \text{-}\mathcal{M}.mono \cdot \mathcal{B}.S\ \text{-}\mathcal{M}.s \in Cast \cdot (isig(\mathcal{M}) \cdot \mathcal{B}.S) \cdot isig(\mathcal{M}) \cdot \mathcal{D}.D$. The remainder of the derivation is straightforward.

2.

$$\dfrac{\Gamma' \vdash \mathcal{B}.S : \star \quad \Gamma' \vdash mot(\mathcal{M}, \mathcal{B}) : isig(\mathcal{M}) \cdot \mathcal{B}.S \to \star \quad \left(\begin{array}{c}\Pi\, \Delta_i'[\mathcal{B}.S/\mathcal{M}.R].\, mot(\mathcal{M}, \mathcal{B})\ (\mathcal{M}.\mathcal{R}.\gamma(i) \cdot \mathcal{B}.S\ \Delta_i') \ni \mathcal{B}.\tau(i) \\ \text{where } (c_i : \Pi\, \Delta_i'.\, \mathcal{M}.\mathcal{R}.D) \in \mathcal{M}.\Xi\end{array}\right)_{i \in \{1 \ldots \#\tau\}}}{\Gamma' \vdash case(\mathcal{M}) \cdot \mathcal{B}.S \cdot mot(\mathcal{M}, \mathcal{B})\ \mathcal{B}.\tau \in \Pi\, \mathcal{M}.y : isig(\mathcal{M}) \cdot \mathcal{B}.S.\, mot(\mathcal{M}, \mathcal{B})\ y}$$

For the last family of type checking premises, pick $i \in \{1 \ldots \Delta\}$, let $\Gamma'' = \Gamma', \Delta_i[\mathcal{B}.S/\mathcal{M}.R]$, and note that by inversion $\tau(i) = \lambda\, \varrho_i.\, t_i''$ where $\vdash \varrho_i := \Delta_i$ and $\Gamma, \Delta_i[\mathfrak{P}.S/D] \vdash \mathfrak{M}.T\ (c_i\ (\xi_i \odot \Delta_i)) \ni \mathfrak{B}.t_i \searrow t_i''[\Delta_i'/\varrho_i]$. Derive

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma'' \vdash \mathcal{M}.T\ (\mathcal{M}.\theta(i)\ \mathcal{B}.\xi_i) \ni t_i''[\Delta_i'/\varrho_i]}\ d}{\Gamma'' \vdash \mathcal{M}.T\ (in\ \text{-}\mathcal{M}.mono\ (\mathcal{M}.\mathcal{R}.\gamma(i) \cdot \mathcal{M}.D\ \mathcal{B}.\xi_i)) \ni t_i''[\Delta_i'/\varrho_i]}\ c}{\Gamma'' \vdash \mathcal{M}.T\ (in\ \text{-}\mathcal{M}.mono\ (elimCast\ \text{-}(\ldots)\ (\mathcal{M}.\mathcal{R}.\gamma(i) \cdot \mathcal{B}.S\ \Delta_i'))) \ni t_i''[\Delta_i'/\varrho_i]}\ \text{b}}{\Gamma'' \vdash mot(\mathcal{M}, \mathcal{B})\ (\mathcal{M}.\mathcal{R}.\gamma(i) \cdot \mathcal{B}.S\ \Delta_i') \ni t_i''[\Delta_i'/\varrho_i]}\ \text{a}}{\Gamma' \vdash \Pi\, \Delta_i'[\mathcal{B}.S/\mathcal{M}.R].\, mot(\mathcal{M}, \mathcal{B})\ (\mathcal{M}.\mathcal{R}.\gamma(i) \cdot \mathcal{B}.S\ \Delta_i') \ni \mathcal{B}.\tau(i)}$$

(a) Exchanging the checked type with a well-kinded convertible one (reduction).

(b) Exchaning the checked type with a well-kinded convertible one (erasure of retyping functions).

(c) Exchaning the checked type with a well-kinded convertible one.
Recall that $\mathcal{M}.\theta(i) = \lambda\,\Delta_i.\ in\ \text{-}\mathcal{M}.mono\ (\mathcal{M}.\mathcal{R}.\gamma(i) \cdot \mathcal{M}.D\ \Delta_i)$.

(d) By inversion of the assumption that $\Gamma' \vdash \Pi\,\Delta_i[\mathcal{B}.S/D].\,\mathcal{M}.T\ (\mathcal{M}.\theta(i)\ \mathcal{B}.\xi_i) \ni \tau(i)$

□

F.3.2 Substitution Lemma

Lemma F.7 (Elaboration respects term substitution)**.**

1. *If* $\vdash \Gamma_1, x : T, \Gamma_2 \searrow \Gamma_1', x : T', \Gamma_2'$ *and* $\Gamma_1 \vdash t \in T \searrow t'$ *then* $\vdash \Gamma_1, \Gamma_2[t/x] \searrow \Gamma_1', \Gamma_2'[t'/x]$
2. *If* $\Gamma_1, x : T, \Gamma_2 \vdash \kappa \searrow \kappa'$ *and* $\Gamma_1 \vdash t \in T \searrow t'$ *then* $\Gamma_1, \Gamma_2[t/x] \vdash \kappa[t/x] \searrow \kappa'[t/x]$
3. *If* $\Gamma_1, x : T, \Gamma_2 \vdash T_1 : \kappa \searrow T'$ *and* $\Gamma_1 \vdash t \in T \searrow t'$ *then* $\Gamma_1, \Gamma_2[t/x] \vdash T_1[t/x] : \kappa[t/x] \searrow T_1'[t'/x]$
4. *If* $\Gamma_1, x : T, \Gamma_2 \vdash t_1 \overset{\leftrightarrow}{\in} T_1 \searrow t_1'$ *and* $\Gamma_1 \vdash t \in T \searrow t'$ *then* $\Gamma_1, \Gamma_2[t/x] \vdash t_1[t/x] \overset{\leftrightarrow}{\in} T_1[t/x] \searrow T_1'[t'/x]$
5. *If* $\Gamma_1, x : T, \Gamma_2; \Gamma_3 \vdash t_1 \searrow t_1'$ *and* $\Gamma_1 \vdash t \in T \searrow t'$ *then* $\Gamma_1, \Gamma_2[t/x]; \Gamma_3 \vdash t_1[|t|/x] \searrow t_1'[|t'|/x]$

Proof. By mutual induction. 1. is by induction on Γ_2, 2-4 are by induction on the kinding and typing derivation. 5. proceeds similarly to Lemma F.6, so we omit its cases (in the variable case, we need IH (1.) to obtain well-formedness of the context in order to apply weakening).

Case: 1. $\Gamma_2 = \emptyset$
By inversion,

$$\frac{\Gamma_1 \vdash \Gamma_1' \quad \Gamma_1 \vdash T : \star \searrow T'}{\vdash \Gamma_1, x{:}T \searrow \Gamma_1, x{:}T'}$$

and the first premise gives us what we need.

Case: 1. $\Gamma_2 = \Gamma_2, y{:}T_{\mathbf{y}}$
By inversion and Proposition E.3,

$$\frac{\vdash \Gamma_1, x{:}T, \Gamma_2 \searrow \Gamma_1', x{:}T', \Gamma_2' \quad \Gamma_1, x{:}T, \Gamma_2 \vdash T_{\mathbf{y}} : \star \searrow T_{\mathbf{y}}'}{\vdash \Gamma_1, x{:}T, \Gamma_2, y{:}T_{\mathbf{y}} \searrow \Gamma_1', x{:}T', \Gamma_2', y{:}T_{\mathbf{y}}'}$$

By the IH(1.), $\vdash \Gamma_1, \Gamma_2[t/x] \searrow \Gamma_1', \Gamma_2'[t'/x]$. By the IH(3.), $\Gamma_1, \Gamma_2[t/x] \vdash T_{\mathbf{y}} : \star \searrow T_{\mathbf{y}}'$. Derive

$$\frac{\vdash \Gamma_1, \Gamma_2[t/x] \searrow \Gamma_1', \Gamma_2'[t'/x] \quad \Gamma_1, \Gamma_2[t/x] \vdash T_{\mathbf{y}}[t/x] \searrow T_{\mathbf{y}}'[t'/x]}{\vdash \Gamma_1, \Gamma_2[t/x], y{:}T_{\mathbf{y}}[t/x] \searrow \Gamma_1', \Gamma_2'[t'/x], y{:}T_{\mathbf{y}}'[t'/x]}$$

Case: 1. $\Gamma_2 = \Gamma_2, X{:}\kappa$
Similar to the above case.

Case: 1. $\Gamma_2 = \Gamma_2, \mathit{Decl}[D, \Delta]$

By assumption, $\emptyset \vdash \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$. The proof otherwise proceeds as above.

Case: 2.

$$\frac{}{\Gamma_1, x{:}T, \Gamma_2 \vdash \star \searrow \star}$$

Immediate.

Case: 2.

$$\frac{\Gamma_1, x{:}T, \Gamma_2 \vdash T_1 : \star \searrow T_1' \quad \Gamma_1, x{:}T, \Gamma_2, y{:}T_1 \vdash \kappa \searrow \kappa'}{\Gamma_1, x{:}T, \Gamma_2 \vdash \Pi\, y{:}T_1.\, \kappa \searrow \Pi\, y{:}T_1'.\, \kappa'}$$

By the IH, $\Gamma_1, \Gamma_2[t/x] \vdash T_1[t/x] : \star \searrow T_1'[t'/x]$. Derive

$$\frac{\vdash \Gamma_1, x{:}T, \Gamma_2 \searrow \Gamma_1', x{:}T', \Gamma_2' \quad \Gamma_1, x{:}T, \Gamma_2 \vdash T_1 : \star \searrow T_1'}{\vdash \Gamma_1, x{:}T, \Gamma_2, y{:}T_1 \searrow \Gamma_1, x{:}T', \Gamma_2', y{:}T_1'}$$

By the IH, $\Gamma_1, \Gamma_2[t/x], y{:}T_1[t/x] \vdash \kappa[t/x] \searrow \kappa'[t'/x]$.
Conclude with the derivation

$$\frac{\Gamma_1, \Gamma_2[t/x] \vdash T_1[t/x] : \star \searrow T_1'[t'/x] \quad \Gamma_1, \Gamma_2[t/x], y{:}T_1[t/x] \vdash \kappa[t/x] \searrow \kappa'[t'/x]}{\Gamma_1, \Gamma_2[t/x] \vdash \Pi\, y{:}T_1[t/x].\, \kappa[t/x] \searrow \Pi\, y{:}T_1'[t'/x].\, \kappa'[t'/x]}$$

Case: 2.

$$\frac{\Gamma_1, x{:}T, \Gamma_2 \vdash \kappa_1 \searrow \kappa_1' \quad \Gamma_1, x{:}T, \Gamma_2, X{:}\kappa \vdash \kappa_2 \searrow \kappa_2'}{\Gamma_1, x{:}T, \Gamma_2 \vdash \Pi\, X{:}\kappa_1.\, \kappa_2 \searrow \Pi\, X{:}\kappa_1'.\, \kappa_2'}$$

Similar to the above case.

Case: 3.

$$\frac{\Gamma_1, x{:}T, \Gamma_2 \vdash \kappa \searrow \kappa' \quad \Gamma_1, x{:}T, \Gamma_2, X{:}\kappa \vdash T_1 : \star \searrow T_1'}{\Gamma_1, x{:}T, \Gamma_2 \vdash \forall\, X{:}\kappa.\, T_1 : \star \searrow \forall\, X{:}\kappa'.\, T_1'}$$

By the IH (2.), $\Gamma_1, \Gamma_2[t/x] \vdash \kappa[t/x] \searrow \kappa'[t'/x]$. We can then derive $\vdash \Gamma_1, x{:}T, \Gamma_2, X{:}\kappa \searrow \Gamma_1', x{:}T', \Gamma_2', X{:}\kappa'$. Inoke the IH (3.) to obtain $\Gamma_1, \Gamma_2[t/x], X{:}\kappa[t/x] \vdash T_1[t/x] : \star \searrow T_1'[t'/x]$. Derive

$$\frac{\Gamma_1, \Gamma_2[t/x] \vdash \kappa[t/x] \searrow \kappa'[t'/x] \quad \Gamma_1, \Gamma_2[t/x], X{:}\kappa[t/x] \vdash T_1[t/x] \searrow T_1'[t'/x]}{\Gamma_1, \Gamma_2[t/x] \vdash \forall\, X{:}\kappa[t/x].\, T_1[t/x] : \star \searrow \forall\, X{:}\kappa'[t'/x].\, T_1'[t'/x]}$$

Case: 3.

$$\frac{\Gamma_1, x{:}T, \Gamma_2 \vdash T_1 : \star \searrow T_1' \quad \Gamma_1, x{:}T, \Gamma_2, y{:}T_1 \vdash T_2 : \star \searrow T_2'}{\Gamma_1, x{:}T, \Gamma_2 \vdash \forall\, y{:}T_1.\, T_2 : \star \searrow \forall\, y{:}T_1'.\, T_2'}$$

By the IH (3.), $\Gamma_1, \Gamma_2[t/x] \vdash T_1[t/x] : \star \searrow T_1'[t'/x]$, and we can thus derive $\vdash \Gamma_1, x{:}T, \Gamma_2, y{:}T_1 \searrow \Gamma_1', x{:}T', \Gamma_2', y{:}T_1'$. By the IH (3.), $\Gamma_1, \Gamma_2[t/x], y{:}T_1[t/x] \vdash T_2[t/x] : \star \searrow T_2'[t'/x]$. Conclude with the derivation

$$\frac{\Gamma_1, \Gamma_2[t/x] \vdash T_1[t/x] \searrow T_1'[t'/x] \quad \Gamma_1, \Gamma_2[t/x], y{:}T_1[t/x] \vdash T_2[t/x] \searrow T_2'[t'/x]}{\Gamma_1, \Gamma_2[t/x] \vdash \forall\, y{:}T_1[t/x].\, T_2[t/x] : \star \searrow \forall\, y{:}T_1'[t'/x].\, T_2'[t'/x]}$$

Case:

$$\frac{\Gamma \vdash T_1 : \star \searrow T_1' \quad \Gamma, x{:}T_1 \vdash T_2 : \star \searrow T_2'}{\Gamma \vdash \Pi\, x{:}T_1.\, T_2 : \star \searrow \Pi\, x{:}T_1'.\, T_2'}$$

Similar to the above case.

Case:

$$\frac{\Gamma \vdash T_1 : \star \searrow T_1' \quad \Gamma, x{:}T_1 \vdash T_2 : \star \searrow T_2'}{\Gamma \vdash \iota\, x{:}T_1.\, T_2 : \star \searrow \iota\, x{:}T_1'.\, T_2'}$$

Similar to the above case.

$$\frac{\Gamma \vdash T_1 : \star \searrow T_1' \quad \Gamma, x{:}T_1 \vdash T_2 : \kappa \searrow T_2'}{\Gamma \vdash \lambda\, x{:}T_1.\, T_2 : \Pi\, x{:}T_1.\, \kappa \searrow \lambda\, x{:}T_1'.\, T_2'}$$

Similar to the above case.

Case: 3.

$$\frac{\Gamma_1, x{:}T, \Gamma_2 \vdash \kappa_1 \searrow \kappa_1' \quad \Gamma_1, x{:}T, \Gamma_2, X{:}\kappa_1 \vdash T_2 : \kappa_2 \searrow T_2'}{\Gamma_1, x{:}T, \Gamma_2 \vdash \lambda\, X{:}\kappa_1.\, T_2 : \Pi\, X{:}\kappa_1.\, \kappa_2 \searrow \lambda\, X{:}\kappa_1'.\, T_2'}$$

By the IH (2.), $\Gamma_1, \Gamma_2[t/x] \vdash \kappa_1[t/x] \searrow \kappa_1'[t'/x]$, and we can form $\vdash \Gamma_1, x : T, \Gamma_2, X : \kappa_1 \searrow \Gamma_1', x : T', \Gamma_2', X : \kappa_1$. Invoke the IH (3.) to obtain $\Gamma_1, \Gamma_2[t/x], X : \kappa_1[t/x] \vdash T_2[t/x] \searrow T_2'[t'/x]$. Derive

$$\frac{\Gamma_1, \Gamma_2[t/x] \vdash \kappa_1[t/x] \searrow \kappa_1'[t'/x] \quad \Gamma_1, \Gamma_2[t/x], X{:}\kappa_1[t/x] \vdash T_2[t/x] : \kappa_2[t/x] \searrow T_2'[t'/x]}{\Gamma_1, \Gamma_2[t/x] \vdash \lambda\, X{:}\kappa[t/x].\, T_2[t/x] : \Pi\, X{:}\kappa_1[t/x].\, \kappa_2[t/x] \searrow \lambda\, X{:}\kappa_1'[t'/x].\, T_2'[t'/x]}$$

Case: 3.

$$\frac{\Gamma \vdash T_1 : \star \searrow T_1' \quad \Gamma, x{:}T_1 \vdash T_2 : \kappa \searrow T_2'}{\Gamma \vdash \lambda\, x{:}T_1.\, T_2 : \Pi\, x{:}T_1.\, \kappa \searrow \lambda\, x{:}T_1'.\, T_2'}$$

Similar to the above case.

Case: 3.

$$\frac{\Gamma_1, x{:}T, \Gamma_2 \vdash T_1 : \Pi\, y{:}T_2.\, \kappa \searrow T_1' \quad \Gamma_1, x{:}T, \Gamma_2 \vdash T_2 \ni t_1 \searrow t_1'}{\Gamma_1, x{:}T, \Gamma_2 \vdash T_1\ t_1 : \kappa[t_1/y]^{T_2} \searrow T_1'\ t_1'}$$

By the IH (3.), $\Gamma_1, \Gamma_2[t/x] \vdash T_1[t/x] : \Pi\, y{:}T_2[t/x].\, \kappa[t/x]$. By the IH (2.), $\Gamma_1, x : T, \Gamma_2 \vdash T_2[t/x] \ni t_1[t/x] \searrow t_1'[t'/x]$. By a straightforward property of substitution, $\kappa[t_1/y]^{T_2}[t/x] = \kappa[t/x][t_1[t/x]/y]^{T_2[t/x]}$. Conclude with the derivation

$$\frac{\Gamma_1, \Gamma_2[t/x] \vdash T_1[t/x] : \Pi\, y{:}T_2[t/x].\, \kappa[t/x] \quad \Gamma_1, \Gamma_2[t/x] \vdash T_2[t/x] \ni t_1[t/x] \searrow t_1'[t/x]}{\Gamma_1, \Gamma_2[t/x] \vdash T_1[t/x]\ t_1[t/x] : \kappa[t_1/y]^{T_2}[t/x] \searrow T_1'[t'/x]\ t_1'[t'/x]}$$

Case: 3.

$$\frac{\begin{array}{c}\Gamma_1, \Gamma_2[t/x] \vdash T_1 : \Pi\, X{:}\kappa_2.\, \kappa_1 \searrow T_1' \\ \Gamma_1, \Gamma_2[t/x] \vdash T_2 : \kappa_3 \searrow T_2' \quad \kappa_2 \cong \kappa_3\end{array}}{\Gamma_1, \Gamma_2[t/x] \vdash T_1 \cdot T_2 : \kappa_1[T_2/X] \searrow T_1' \cdot T_2'}$$

Similar to the above case (substitution preserves kind convertibility).

Case: 3.

$$\frac{\Gamma_1, x{:}T, \Gamma_2; \emptyset \vdash |t_1| \searrow t_1' \quad \Gamma_1, x{:}T, \Gamma_2; \emptyset \vdash |t_2| \searrow t_2'}{\Gamma_1, x{:}T, \Gamma_2 \vdash \{t_1 \simeq t_2\} : \star \searrow \{t_1' \simeq t_2'\}}$$

Note that we take $|t_1|[t/x] = |t_1[|t|/x]|$. By the IH (5.), $\Gamma_1, \Gamma_2[t/x]; \emptyset \vdash |t_1[|t|/x]| \searrow t_1'[|t'|/x]$ and $\Gamma_1, \Gamma_2[t/x]; \emptyset \vdash |t_2[|t|/x]|| \searrow t_2'[|t'|/x]$. We then derive

$$\frac{\Gamma_1, \Gamma_2[t/x]; \emptyset \vdash |t_1[|t|/x]| \searrow t_1'[|t|/x] \quad \Gamma_1, \Gamma_2[t/x]; \emptyset \vdash |t_2[|t|/x]| \searrow t_2'[|t|/x]}{\Gamma_1, \Gamma_2[t/x] \vdash \{t_1[t/x] \simeq t_2[t/x]\} : \star \searrow \{t_1'[|t|/x] \simeq t_2'[|t|/x]\}}$$

Case: 4.

$$\frac{(x{:}T) \in \Gamma_1, x{:}T, \Gamma_2}{\Gamma_1, x{:}T, \Gamma_2 \vdash x \in T \searrow x}$$

By assumption, we have $\Gamma_1 \vdash t \in T \searrow t'$. By IH (1.), $\vdash \Gamma_1, \Gamma_2[t/x] \searrow \Gamma_1', \Gamma_2'[t'/x]$. Apply weakening to obtain

$$\Gamma_1, \Gamma_2[t/x] \vdash t \in T \searrow t'$$

Case: 4.

$$\frac{\Gamma_1, x{:}T, \Gamma_2 \vdash t_1 \in T_2 \searrow t_1' \quad T_1 \cong T_2}{\Gamma_1, x{:}T, \Gamma_2 \vdash T_1 \ni t_1 \searrow t_1'}$$

By the IH (4.), $\Gamma_1, \Gamma_2[t/x] \vdash t_1[t/x] \in T_2[t/x] \searrow t_1'[t'/x]$. Substitution preserves convertibility, so $T_1[t/x] \cong T_2[t/x]$. Conclude with the derivation

$$\frac{\Gamma_1, \Gamma_2[t/x] \vdash t_1[t/x] \in T_2[t/x] \searrow t_1'[t'/x] \quad T_1[t/x] \cong T_2[t/x]}{\Gamma_1, \Gamma_2[t/x] \vdash T_1[t/x] \ni t_1[t/x] \searrow t_1'[t'/x]}$$

Case: 4.

$$\frac{T_3 \dashrightarrow^* \Pi\, y{:}T_1.\, T_2 \quad \Gamma_1, x{:}T, \Gamma_2, y{:}T_1 \vdash T_2 \ni t_3 \searrow t_3'}{\Gamma_1, x{:}T, \Gamma_2 \vdash T_3 \ni \lambda\, y.\, t_3 \searrow \lambda\, y.\, t_3'}$$

We have $T_3[t/x] \dashrightarrow \Pi\, y : T_1[t/x].\, T_2[t/x]$. By the IH (4.) $\Gamma_1, \Gamma_2[t/x] \vdash T_2[t/x] \ni t_3[t/x] \searrow t_3'[t'/x]$. Conclude with the derivation

$$\frac{T_3[t/x] \dashrightarrow^* \Pi\, y{:}T_1[t/x].\, T_2[t/x] \quad \Gamma_1, \Gamma_2[t/x], y{:}T_1[t/x] \vdash T_2[t/x] \ni t_3[t/x] \searrow t_3'[t'/x]}{\Gamma_1, \Gamma_2[t/x] \vdash T_3[t/x] \ni \lambda\, y.\, t_3[t/x] \searrow \lambda\, y.\, t_3'[t'/x]}$$

Case: 4.

$$\frac{T_3 \dashrightarrow^* \forall\, X{:}\kappa.\, T_2 \quad \Gamma_1, \Gamma_2[t/x], X{:}\kappa \vdash T_2 \ni t_2 \searrow t_2'}{\Gamma_1, x{:}T, \Gamma_2 \vdash T_3 \ni \Lambda\, X.\, t_2 \searrow \Lambda\, X.\, t_2'}$$

Similar to the above case.

Case: 4.

$$\frac{T_3 \dashrightarrow^* \forall\, y{:}T_1.\, T_2 \quad \Gamma_1, x{:}T, \Gamma_2, y{:}T_1 \vdash T_2 \ni t_3 \searrow t_3' \quad y \notin FV(|t_3|)}{\Gamma_1, x{:}T, \Gamma_2 \vdash T_3 \ni \Lambda\, y.\, t_3 \searrow \Lambda\, y.\, t_3'}$$

Similar to the above case (note $y \notin FV(|t_3[t/x]|)$)

Case: 4.

$$\frac{\Gamma_1, x{:}T, \Gamma_2 \vdash t_3 \stackrel{\dashrightarrow^*}{\in} \Pi\, y{:}T_1.\, T_2 \searrow t_3' \quad \Gamma_1, x{:}T, \Gamma_2 \vdash T_1 \ni t_1 \searrow t_1'}{\Gamma_1, x{:}T, \Gamma_2 \vdash t_3\ t_1 \in T_2[t_1/y]^{T_1} \searrow t_3'\ t_1'}$$

By the IH (4.) and substitution and reduction, $\Gamma_1, \Gamma_2[t/x] \vdash t_3[t/x] \stackrel{\dashrightarrow^*}{\in} \Pi\, y{:}T_1[t/x].\, T_2[t/x] \searrow t_3'[t'/x]$. Again by IH (4.), $\Gamma_1, \Gamma_2[t/x] \vdash T_1[t/x] \ni t_1[t/x] \searrow t_1'[t'/x]$. Observe $T_2[t_1/y]^{T_1}[t/x] = T_2[t/x][t_1[t/x]/y]^{T_1[t/x]}$. Conclude with the derivation

$$\frac{\Gamma_1, \Gamma_2[t/x] \vdash t_3[t/x] \stackrel{\dashrightarrow^*}{\in} \Pi\, y{:}T_1[t/x].\, T_2[t/x] \searrow t_3'[t'/x] \quad \Gamma_1, \Gamma_2[t/x] \vdash T_1[t/x] \ni t_1[t/x] \searrow t_1'[t'/x]}{\Gamma_1, \Gamma_2[t/x] \vdash t_3[t/x]\ t_1[t/x] \in T_2[t_1/y]^{T_1}[t/x] \searrow t_3'[t'/x]\ t_1'[t'/x]}$$

Case: 4.

$$\frac{\Gamma_1, x{:}T, \Gamma_2 \vdash t_3 \stackrel{\dashrightarrow^*}{\in} \forall\, X{:}\kappa.\, T_2 \searrow t_3' \quad \Gamma_1, x{:}T, \Gamma_2 \vdash T_1 : \kappa_1 \searrow T_1' \quad \kappa_1 \cong \kappa}{\Gamma_1, x{:}T, \Gamma_2 \vdash t_3 \cdot T_1 \in T_2[T_1/X] \searrow t_3' \cdot T_1'}$$

Similar to the above case.

Case: 4.

$$\frac{\Gamma_1, x{:}T, \Gamma_2 \vdash t_2 \stackrel{\dashrightarrow^*}{\in} \forall\, x{:}T_1.\, T_2 \searrow t_2' \quad \Gamma_1, x{:}T, \Gamma_2 \vdash T_1 \ni t_1 \searrow t_1'}{\Gamma_1, x{:}T, \Gamma_2 \vdash t_2\ {-}t_1 \in T_2[t_1/x]^{T_1} \searrow t_2'\ {-}t_1'}$$

Similar to the above case.

Case: 4.

$$\frac{T_3 \dashrightarrow^* \iota\, y{:}T_1.\, T_2 \quad \Gamma_1, x{:}T, \Gamma_2 \vdash T_1 \ni t_1 \searrow t_1' \quad \Gamma_1, x{:}T, \Gamma_2 \vdash T_2[t_1/y]^{T_1} \ni t_2 \searrow t_2' \quad |t_1| \cong |t_2|}{\Gamma_1, x{:}T, \Gamma_2 \vdash T_3 \ni [t_1, t_2] \searrow [t_1', t_2']}$$

Obtain $T_3[t/x] \dashrightarrow^* \iota\, y{:}T_1[t/x].\, T_2[t/x]$. By the IH (4.), $\Gamma_1, \Gamma_2[t/x] \vdash T_1[t/x] \ni t_1[t/x] \searrow t_1'[t'/x]$. By the IH (4.), $\Gamma_1, \Gamma_2[t/x] \vdash T_2[t_1/y]^{T_1}[t/x] \ni t_2[t/x] \searrow t_2'[t/x]$. Obtain $|t_1[|t|/x]| \cong |t_2[|t|/x]|$. Observe $T_2[t_1/y]^{T_1}[t/x] = T_2[t/x][t_1[t/x]/y]^{T_1[t/x]}$. Derive

$$\frac{T_3[t/x] \dashrightarrow^* \iota\, y{:}T_1[t/x].\, T_2[t/x] \quad \Gamma_1, \Gamma_2[t/x] \vdash T_1[t/x] \ni t_1[t/x] \searrow t_1'[t'/x] \quad \Gamma_1, \Gamma_2[t/x] \vdash T_2[t_1/y]^{T_1}[t/x] \ni t_2[t/x] \searrow t_2'[t'/x] \quad |t_1[|t|/x]| \cong |t_2[|t|/x]|}{\Gamma_1, \Gamma_2[t/x] \vdash T_3[t/x] \ni [t_1[t/x], t_2[t/x]] \searrow [t_1'[t'/x], t_2'[t'/x]]}$$

Case: 4.

$$\frac{\Gamma_1, x{:}T, \Gamma_2 \vdash t_1 \stackrel{\dashrightarrow^*}{\in} \iota\, y{:}T_1.\, T_2 \searrow t_1'}{\Gamma_1, x{:}T, \Gamma_2 \vdash t_1.1 \in T_1 \searrow t_1'.1}$$

By the IH (4.) and a property of substitution and reduction, $\Gamma_1, \Gamma_2[t/x] \vdash t_1[t/x] \stackrel{\dashrightarrow^*}{\in} \iota\, y{:}T_1.\, T_2 \searrow t_1'$. Derive

$$\frac{\Gamma_1, \Gamma_2[t/x] \vdash t_1[t/x] \stackrel{\dashrightarrow^*}{\in} \iota\, y{:}T_1[t/x].\, T_2[t/x] \searrow t_1[t/x]}{\Gamma_1, \Gamma_2[t/x] \vdash t_1.1[t/x] \in T_1[t/x] \searrow t_1'.1[t'/x]}$$

Case: 4.

$$\frac{\Gamma_1, x{:}T, \Gamma_2 \vdash t_1 \stackrel{\dashrightarrow^*}{\in} \iota\, y{:}T_1.\, T_2 \searrow t_1'}{\Gamma_1, x{:}T, \Gamma_2 \vdash t_1.2 \in T_2[t_1.1/x] \searrow t_1'.2}$$

Similar to the above case.

Case: 4.

$$\frac{T_1 \dashrightarrow^* \{t_1 \simeq t_2\} \quad \Gamma_1, x{:}T, \Gamma_2; \emptyset \vdash |t_3| \searrow t_3' \quad |t_1| \cong |t_2|}{\Gamma_1, x{:}T, \Gamma_2 \vdash T_1 \ni \beta\{t_3\} \searrow \beta\{t_3'\}}$$

Obtain $T_1[t/x] \dashrightarrow^* \{t_1[t/x] \simeq t_2[t/x]\}$. By the IH (5.), $\Gamma_1, \Gamma_2[t/x] \vdash |t_3[|t|/x]| \searrow t_3'[|t'|/x]$. Obtain $|t_1[|t|/x]| \simeq |t_2[|t|/x]|$. Derive

$$\frac{T_1[t/x] \dashrightarrow^* \{t_1[t/x] \simeq t_2[t/x]\} \quad \Gamma_1, \Gamma_2[t/x]; \emptyset \vdash |t_3[|t|/x]| \searrow t_3'[|t|/x] \quad |t_1[|t|/x]| \cong |t_2[|t|/x]|}{\Gamma_1, \Gamma_2[t/x] \vdash T_1[t/x] \ni \beta\{t_3[t/x]\} \searrow \beta\{t_3'[|t|/x]\}}$$

Case:

$$\frac{\Gamma \vdash t_1 \in T_2 \searrow t_1' \quad T_2 \cong \{\lambda\, x.\, \lambda\, y.\, x \simeq \lambda\, x.\, \lambda\, y.\, y\}}{\Gamma_1, x{:}T, \Gamma_2 \vdash T_1 \ni \delta \text{ - } t_1 \searrow \delta \text{ - } t_1'}$$

Easy application of the IH.

Case:

$$\frac{\begin{array}{c}\Gamma_1, x{:}T, \Gamma_2 \vdash t_4 \stackrel{\dashrightarrow^*}{\in} \{t_1 \simeq t_3\} \searrow t_4' \quad \Gamma_1, x{:}T, \Gamma_2; \emptyset \vdash |t_2| \searrow t_2' \\ |t_3| \cong |t_2| \quad \Gamma_1, x{:}T, \Gamma_2 \vdash T_1[t_2/y] : \star \searrow T_1' \\ \Gamma_1, x{:}T, \Gamma_2 \vdash T_1[t_2/y] \ni t_1 \searrow t_1' \quad T_1[t_1/y] \cong T_4\end{array}}{\Gamma_1, x{:}T, \Gamma_2 \vdash T_4 \ni \rho\ t_4\ @y\langle t_2\rangle T_1 \text{ - } t_1 \searrow \rho\ t_4'\ @x\langle t_2'\rangle.T_1' \text{ - } t_1'}$$

- By the IH (4.), $\Gamma_1, \Gamma_2[t/x] \vdash t_3[t/x] \stackrel{\dashrightarrow^*}{\in} \{t_1[t/x] \simeq t_2[t/x]\} \searrow t_4'[t/x]$.
- By the IH (5.), $\Gamma_1, \Gamma_2[t/x]; \emptyset \vdash |t_2[|t|/x]| \searrow t_2'[|t'|/x]$.
- Obtain $|t_3[|t|/x]| \cong |t_2[|t|/x]|$.
- By the IH (3.), $\Gamma_1, \Gamma_2[t/x] \vdash T_1[t_2/y][t/x] : \star \searrow T_1'[t'/x]$.
  Note $T_1[t_2/y][t/x] = T_1[t/x][t_2[t/x]/y]$.
- By the IH (4.), $\Gamma_1, \Gamma_2[t/x] \vdash T_1[t_2/y][t/x] \ni t_1[t/x] \searrow t_1'[t/x]$.
- Obtain $T_1[t_1/y][t/x] \cong T_4[t/x]$.

Conclude with the typing rule for ρ.

Case: 4.

$$\frac{\Gamma_1, x{:}T, \Gamma_2 \vdash t_2 \overset{\leftrightarrow}{\in} T_2 \searrow t_2' \quad \Gamma_1, x{:}T, \Gamma_2; \emptyset \vdash |t_3| \searrow t_3' \quad \Gamma_1, x{:}T, \Gamma_2 \vdash \{t_2 \simeq t_3\} \ni t_1 \searrow t_1'}{\Gamma_1, x{:}T, \Gamma_2 \vdash \varphi\ t_1\ \text{-}\ t_2\ \{t_3\} \overset{\leftrightarrow}{\in} T_2 \searrow \varphi\ t_1'\ \text{-}\ t_2'\ \{t_3'\}}$$

1. By the IH (4.), $\Gamma_1, \Gamma_2[t/x] \vdash t_2[t/x] \overset{\leftrightarrow}{\in} T_2[t/x] \searrow t_2'[t'/x]$.
2. By the IH (5.), $\Gamma_1, \Gamma_2[t/x]; \emptyset \vdash |t_3[|t|/x]| \searrow t_3'[t'/x]$
3. By the IH (4.), $\Gamma_1, \Gamma_2[t/x] \vdash \{t_2[t/x] \simeq t_3[t/x]\} \ni t_1[t/x] \searrow t_1'[t'/x]$

Conclude with an application of the rule for φ.

Case: 4.

$$\frac{\Gamma_1, x{:}T, \Gamma_2 \vdash T_1 : \star \searrow T_1' \quad \Gamma_1, x{:}T, \Gamma_2 \vdash T_1 \ni t_1 \searrow t_1'}{\Gamma_1, x{:}T, \Gamma_2 \vdash \chi\ T_1\ \text{-}\ t_1 \in T_1 \searrow \chi\ T_1'\ \text{-}\ t_1'}$$

1. By the IH (3.) and inversion, $\Gamma_1, \Gamma_2[t/x] \vdash T_1[t/x] : \star \searrow T_1'[t'/x]$.
2. By the IH (4.), $\Gamma_1, \Gamma_2[t/x] \vdash T_1[t/x] \ni t_1[t/x] \searrow t_1'[t'/x]$

Conclude with an application of the rule for χ. □

Lemma F.8 (Elaboration respects type constructor substitution)**.**

1. *If* $\vdash \Gamma_1, X : \kappa, \Gamma_2 \searrow \Gamma_1', X : \kappa', \Gamma_2'$ *and* $\Gamma_1 \vdash T \in \kappa \searrow T'$ *then* $\vdash \Gamma_1, \Gamma_2[T/X] \searrow \Gamma_1', \Gamma_2'[T'/X]$
2. *If* $\Gamma_1, X : \kappa, \Gamma_2 \vdash \kappa_1 \searrow \kappa_1'$ *and* $\Gamma_1 \vdash T \in \kappa \searrow T'$ *then* $\Gamma_1, \Gamma_2[T/X] \vdash \kappa_1[T/X] \searrow \kappa_1'[T/X]$
3. *If* $\Gamma_1, X : \kappa, \Gamma_2 \vdash T_1 : \kappa \searrow T_1'$ *and* $\Gamma_1 \vdash T \in \kappa \searrow T'$ *then* $\Gamma_1, \Gamma_2[T/X] \vdash T_1[T/X] : \kappa_1[T/X] \searrow T_1'[T'/X]$
4. *If* $\Gamma_1, X : \kappa, \Gamma_2 \vdash t_1 \overset{\leftrightarrow}{\in} T_1 \searrow t_1'$ *and* $\Gamma_1 \vdash T \in \kappa \searrow T'$ *then* $\Gamma_1, \Gamma_2[T/\kappa] \vdash t_1[T/X] \overset{\leftrightarrow}{\in} T_1[T/X] \searrow T_1'[T'/X]$

Proof. The proof is similar to Lemma F.7, and so omitted. □

Lemma F.9 (Substitution in constructor parameter telescopes)**.** *If* $\Gamma \vdash_{\textbf{\textit{data}}} \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$ *and* $\Gamma \vdash S : \star \searrow S'$ *then for all* $(c_i : \Pi\, \Delta_i.\, D) \in \Delta$, $\Gamma \vdash \Delta_i[S/D] \searrow \Delta_i'[S'/R]$ *where* $\Gamma, R : \star \vdash \Delta_i[R/D] \searrow \Delta_i'$

F.3.3 Theorem 5.46

Lemma F.10. *If* $\vdash \Gamma_1 \searrow \Gamma_1'$ *then*

- *if* $\Gamma_1 \vdash T : \star \searrow T'$ *and* $\Gamma_1, x{:}T; \Gamma_2 \vdash t \searrow t'$ *then* $\Gamma_1; x : \mathit{Top}, \Gamma_2 \vdash t \searrow t'$. *Furthermore, if* $x \notin \mathit{FV}(t)$ *then* $\Gamma_1; \Gamma_2 \vdash t \searrow t'$.
- *if* $\Gamma_1 \vdash \kappa \searrow \kappa'$ *and* $\Gamma_1, X{:}\kappa; \Gamma_2 \vdash t \searrow t'$ *then* $\Gamma_1; \Gamma_2 \vdash t \searrow t'$.

Proof. By two straightforward inductions on the assumed typing derivation. □

Lemma F.11. *If* $\vdash \Gamma_1, \Gamma_2 \searrow \Gamma'$ *and* $\Gamma_1; \Gamma_2 \vdash t \searrow t'$ *then* $|t'| = t$.

Proof. By a straightforward induction on the assumed derivation of well-formedness for the untyped term. □

Proof (of Theorem 5.46). We proceed by mutual induction on the typing derivation. Some obvious cases or cases very similar to others given are omitted.

Case:

$$\frac{T \dashrightarrow^* \Pi\, x{:}T_1.\, T_2 \quad \Gamma, x{:}T_1 \vdash T_2 \ni t \searrow t'}{\Gamma \vdash T \ni \lambda\, x.\, t \searrow \lambda\, x.\, t'}$$

Call-by-name reduction of types preserves well-formedness, so (by inversion) $\Gamma, x : T_1 \vdash T_2 : \star \searrow T_2'$. Invoke the IH to obtain $\Gamma, x : T_1; \emptyset \vdash |t| \searrow |t'|$. By Lemma F.10, $\Gamma; x{:}\mathit{Top} \vdash |t| \searrow |t'|$. Conclude with the derivation

$$\frac{\Gamma; x{:}\mathit{Top} \vdash |t| \searrow |t'|}{\Gamma; \emptyset \vdash \lambda\, x.\, |t| \searrow \lambda\, x.\, |t'|}$$

Case:

$$\frac{\Gamma \vdash t \overset{\dashrightarrow^*}{\in} \forall\, X{:}\kappa.\, T_2 \searrow t' \quad \Gamma \vdash T_1 : \kappa_1 \searrow T_1' \quad \kappa_1 \cong \kappa}{\Gamma \vdash t \cdot T_1 \in T_2[T_1/X] \searrow t' \cdot T_1'}$$

By the IH, $\Gamma; \emptyset \vdash |t| \searrow |t'|$. By erasure, this is what we need.

Case:

$$\frac{T \dashrightarrow^* \forall\, x{:}T_1.\, T_2 \quad \Gamma, x{:}T_1 \vdash T_2 \ni t \searrow t' \quad x \notin FV(|t|)}{\Gamma \vdash T \ni \Lambda\, x.\, t \searrow \Lambda\, x.\, t'}$$

By the IH, $\Gamma, x : T_1; \emptyset \vdash |t| \searrow |t'|$. By Lemma F.10 and the premise, $\Gamma; \emptyset \vdash |t| \searrow |t'|$. By erasure, this is what we need to conclude.

Case:

$$\frac{T \dashrightarrow^* \{t_1 \simeq t_2\} \quad \Gamma; \emptyset \vdash |t_3| \searrow t_3' \quad |t_1| \cong |t_2|}{\Gamma \vdash T \ni \beta\{t_3\} \searrow \beta\{t_3'\}}$$

By the premise and Lemma F.11, $|t_3'| = t_3'$, so we have $|\beta\{t_3'\}| = t_3'$. Therefore, we have $\Gamma; \emptyset \vdash |\beta\{t_3\}| \searrow |\beta\{t_3'\}|$. □

F.3.4 Theorem 5.48

Lemma F.12 (Convertibility soundness for typed elaboration)**.** *If* $\vdash \Gamma \searrow \Gamma'$ *then*

1. *If* $\Gamma \vdash \kappa_1 \searrow \kappa_1'$ *and* $\Gamma \vdash \kappa_2 \searrow \kappa_2'$ *and* $\kappa_1 \cong \kappa_2$ *then* $\kappa_1' \cong \kappa_2'$.
2. *If* $\Gamma \vdash T_1 : \kappa_1 \searrow T_1'$ *and* $\Gamma \vdash T_2 : \kappa_2 \searrow T_2'$ *and* $T_1 \cong T_2$ *then* $T_1' \cong T_2'$.
3. *If* $\Gamma \vdash t_1 \overset{\leftrightarrow}{\in} T_1 \searrow t_1$ *and* $\Gamma \vdash t_2 \overset{\leftrightarrow}{\in} T_2$ *and* $|t_1| \cong |t_2|$ *then* $|t_1'| \cong |t_2'|$.

Proof. By mutual induction on the assumed derivations. Part 1. consists entirely of congruence rules, which we omit. □

Proof (of Theorem 5.48). By mutual induction on the assumed derivation. We show some interesting cases.

Case:

$$\frac{}{\Gamma \vdash \star \searrow \star}$$

Immediate.

Case:

$$\frac{\Gamma \vdash T : \star \searrow T' \quad \Gamma, x : T \vdash \kappa \searrow \kappa'}{\Gamma \vdash \Pi\, x : T.\, \kappa \searrow \Pi\, x : T'.\, \kappa'}$$

By the IH and inversion, $\Gamma' \vdash T' : \star$, and we can derive $\vdash \Gamma, x : T \searrow \Gamma', x : T'$. Apply the IH again to obtain $\Gamma', x : T' \vdash \kappa'$, and conclude with

$$\frac{\Gamma' \vdash T' : \star \quad \Gamma', x : T' \vdash \kappa'}{\Gamma' \vdash \Pi\, x : T'.\, \kappa'}$$

Case:

$$\frac{\Gamma \vdash \kappa_1 \searrow \kappa_1' \quad \Gamma, X : \kappa \vdash \kappa_2 \searrow \kappa_2'}{\Gamma \vdash \Pi\, X : \kappa_1.\, \kappa_2 \searrow \Pi\, X : \kappa_1'.\, \kappa_2'}$$

Similar to the above case

Case:

$$\frac{(X : \kappa) \in \Gamma}{\Gamma \vdash X : \kappa \searrow X}$$

By Lemma E.1, $(X : \kappa') \in \Gamma'$ such that $\Gamma \vdash \kappa \searrow \kappa'$. Exhibit κ'.

Case:

$$\frac{\Gamma \vdash \kappa \searrow \kappa' \quad \Gamma, X : \kappa \vdash T : \star \searrow T'}{\Gamma \vdash \forall\, X : \kappa.\, T : \star \searrow \forall\, X : \kappa'.\, T'}$$

By the IH, $\Gamma' \vdash \kappa'$, so we obtain $\vdash \Gamma, X : \kappa \searrow \Gamma', X : \kappa'$. By the IH and inversion, $\Gamma', X : \kappa' \vdash T' : \star$. Exhibit $\star$ and conclude with the derivation

$$\frac{\Gamma' \vdash \kappa' \quad \Gamma', X : \kappa' \vdash T_2' : \star}{\Gamma' \vdash \forall\, X : \kappa'.\, T' : \star}$$

Case:

$$\frac{\Gamma \vdash T_1 : \star \searrow T_1' \quad \Gamma, x : T_1 \vdash T_2 : \star \searrow T_2'}{\Gamma \vdash \forall\, x : T_1.\, T_2 : \star \searrow \forall\, x : T_1'.\, T_2'}$$

By the IH and inversion, $\Gamma' \vdash T_1' : \star$, so we obtain $\vdash \Gamma, x : T_1 \searrow \Gamma', x : T_1'$. By the IH and inversion again, $\Gamma', x : T_1' \vdash T_2' : \star$. Exhibit $\star$ and conclude with the derivation

$$\frac{\Gamma' \vdash T_1' : \star \quad \Gamma', x : T_2' \vdash T_2' : \star}{\Gamma' \vdash \forall\, x : T_1'.\, T_2' : \star}$$

Case:

$$\frac{\Gamma \vdash T_1 : \star \searrow T_1' \quad \Gamma, x{:}T_1 \vdash T_2 : \star \searrow T_2'}{\Gamma \vdash \Pi\, x{:}T_1.\, T_2 : \star \searrow \Pi\, x{:}T_1'.\, T_2'}$$

Similar to the above case.

Case:

$$\frac{\Gamma \vdash T_1 : \star \searrow T_1' \quad \Gamma, x{:}T_1 \vdash T_2 : \star \searrow T_2'}{\Gamma \vdash \iota\, x{:}T_1.\, T_2 : \star \searrow \iota\, x{:}T_1'.\, T_2'}$$

Similar to the above case.

Case:

$$\frac{\Gamma \vdash T_1 : \star \searrow T_1' \quad \Gamma, x{:}T_1 \vdash T_2 : \kappa \searrow T_2'}{\Gamma \vdash \lambda\, x{:}T_1.\, T_2 : \Pi\, x{:}T_1.\, \kappa \searrow \lambda\, x{:}T_1'.\, T_2'}$$

By the IH and inversion, $\Gamma' \vdash T_1' : \star$, so we obtain $\vdash \Gamma, x{:}T_1 \searrow \Gamma', x{:}T_1'$. By the IH, $\Gamma', x{:}T_1' \vdash T_2' : \kappa'$ for some κ' such that $\Gamma, x{:}T_1 \vdash \kappa \searrow \kappa'$.

Exhibit $\Pi\, x{:}T_1'.\, \kappa'$ with derivations

$$\frac{\Gamma' \vdash T_1' : \star \quad \Gamma', x{:}T_1' \vdash T_2' : \kappa'}{\Gamma' \vdash \lambda\, x{:}T_1'.\, T_2' : \Pi\, x{:}T_1'.\, \kappa'} \quad \frac{\Gamma \vdash T_1 : \star \searrow T_1' \quad \Gamma, x{:}T_1 \vdash \kappa \searrow \kappa'}{\Gamma \vdash \Pi\, x{:}T_1.\, \kappa \searrow \Pi\, x{:}T_1'.\, \kappa'}$$

Case:

$$\frac{\Gamma \vdash \kappa_1 \searrow \kappa_1' \quad \Gamma, X{:}\kappa_1 \vdash T : \kappa_2 \searrow T'}{\Gamma \vdash \lambda\, X{:}\kappa_1.\, T : \Pi\, X{:}\kappa_1.\, \kappa_2 \searrow \lambda\, X{:}\kappa_1'.\, T'}$$

Similar to the above case.

Case:

$$\frac{\Gamma \vdash T : \Pi\, x{:}T_1.\, \kappa \searrow T' \quad \Gamma \vdash T_1 \ni t \searrow t'}{\Gamma \vdash T\ t : \kappa[t/x]^{T_1} \searrow T'\ t'}$$

By the IH, $\Gamma' \vdash T' : \kappa''$ for some κ'' with $\Gamma \vdash \Pi\, x{:}T_1.\, \kappa \searrow \kappa''$, and by inversion κ'' of the form $\Pi\, x{:}T_1'.\, \kappa'$ where

$$\frac{\Gamma \vdash T_1 : \star \searrow T_1' \quad \Gamma, x{:}T_1 \vdash \kappa \searrow \kappa'}{\Gamma \vdash \Pi\, x{:}T_1.\, \kappa \searrow \Pi\, x{:}T_1'.\, \kappa'}$$

By the IH (using $\Gamma \vdash T_1' : \star$), $\Gamma' \vdash T_1' \ni t'$. By Lemma F.7, $\Gamma \vdash \kappa[t/x]^{T_1} \searrow \kappa'[t'/x]^{T_1'}$

Conclude with the derivation

$$\frac{\Gamma' \vdash T' : \Pi\, x{:}T_1'.\, \kappa' \quad \Gamma' \vdash T_1' \ni t'}{\Gamma' \vdash T'\ t' : \kappa'[t'/x]^{T_1'}}$$

Case:

$$\frac{\Gamma \vdash T_1 : \Pi\, X{:}\kappa_2.\, \kappa_1 \searrow T_1' \quad \Gamma \vdash T_2 : \kappa_3 \searrow T_2' \quad \kappa_2 \cong \kappa_3}{\Gamma \vdash T_1 \cdot T_2 : \kappa_1[T_2/X] \searrow T_1' \cdot T_2'}$$

By the IH and inversion, $\Gamma' \vdash T_1' : \Pi\, X : \kappa_2'.\, \kappa_1'$ where

$$\frac{\Gamma \vdash \kappa_2 \searrow \kappa_2' \quad \Gamma, X : \kappa_2 \vdash \kappa_1 \searrow \kappa_1'}{\Gamma \vdash \Pi\, X : \kappa_2.\, \kappa_1 \searrow \Pi\, X : \kappa_2'.\, \kappa_1'}$$

By the IH again, $\Gamma' \vdash T_2' : \kappa_3'$ where $\Gamma' \vdash \kappa_3 \searrow \kappa_3'$. Apply Lemma F.12 to $\kappa_2 \cong \kappa_3$ to obtain $\kappa_2' \cong \kappa_3'$. Exhibit $\kappa_1'[T_2'/X]$ with the derivation

$$\frac{\Gamma' \vdash T_1' : \Pi\, X : \kappa_2'.\, \kappa_1' \quad \Gamma' \vdash T_2' : \kappa_3' \quad \kappa_2' \cong \kappa_3'}{\Gamma' \vdash T_1' \cdot T_2' : \kappa_1'[T_2'/X]}$$

and apply Lemma F.8 to obtain $\Gamma \vdash \kappa_1[T_2/X] \searrow \kappa_1'[T_2'/X]$.

Case: 2.

$$\frac{\Gamma; \emptyset \vdash |t_1| \searrow t_1' \quad \Gamma; \emptyset \vdash |t_2| \searrow t_2'}{\Gamma \vdash \{t_1 \simeq t_2\} : \star \searrow \{t_1' \simeq t_2'\}}$$

By Proposition F.3, we have $FV(t_1') \subseteq DV(\Gamma')$ and $FV(t_2') \subseteq DV(\Gamma')$. Derive

$$\frac{FV(t_1'\ t_2') \subseteq DV(\Gamma')}{\Gamma' \vdash \{t_1' \simeq t_2'\} : \star}$$

Case: 3.

$$\frac{\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D}}{\Gamma \vdash D : \star \searrow \mathcal{D}.D}$$

By Proposition F.1, $\Gamma \vdash_{\mathbf{data}} \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$. Invoke Theorem 5.10, satisfying the assumptions of Definition 4.26 with

- $\Gamma \vdash \mathit{Decl}[D, \Delta]$ *vs* by a global assumption
- $\vdash \Gamma \searrow \Gamma'$ by assumption
- Constructor paramater elaboration soundness by the induction hypotheses.

We therefore obtain $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{D})$ *valid*, which in particular means $\Gamma' \vdash \mathcal{D}.D : \star$. Exhibit $\star$.

Case: 2.

$$\frac{\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D}}{\Gamma \vdash \texttt{Is/}D : \star \to \star \searrow \mathit{IsD} \cdot \mathit{isig}(\mathcal{D})}$$

Similar to the above case; we also use the type formation rule for *IsD* from Figure 5.6. Exhibit $\star \to \star$.

Case: 3.

$$\frac{\Gamma \vdash \mathit{Con}[c] : \mathit{Decl}[D, \Delta] \searrow (\mathcal{D}, i) \quad (c : \Pi\, \Delta'.\, D) \in \Delta}{\Gamma \vdash c \in \Pi\, \Delta'.\, D \searrow \mathcal{D}.\theta(i)}$$

By Proposition F.1, $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$ with c the ith entry of Δ and thus $\mathcal{D}.\Delta$. Invoke Theorem 5.10 with

- $\Gamma \vdash \mathit{Decl}[D, \Delta]$ *vs* by a global assumption

- $\vdash \Gamma \searrow \Gamma'$ by assumption
- Constructor paramater elaboration soundness by the induction hypotheses.

We therefore obtain $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{D})$ *valid*, which in particular means $\Gamma' \vdash \mathcal{D}.\theta(i) : \mathcal{D}.\Delta(i)$.

Exhibit $\mathcal{D}.\Delta(i)$, as we can see $\Gamma \vdash \Pi\, \Delta'.\, D \searrow \mathcal{D}.\Delta(i)$

Case: 3.

$$\frac{\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D}}{\Gamma \vdash \mathtt{is}/D \in \mathtt{Is}/D \cdot D \searrow \mathit{isD} \cdot \mathit{isig}(\mathcal{D})\ \text{-}\mathcal{D}.\mathit{mono}}$$

By Proposition F.1, $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$. Invoke Theorem 5.10 with

- $\Gamma \vdash \mathit{Decl}[D, \Delta]$ *vs* by a global assumption
- $\vdash \Gamma \searrow \Gamma'$ by assumption
- Constructor parameter elaboration soundness by the induction hypothesis

We therefore obtain $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{D})$ *valid*, which in particular means

- $\Gamma' \vdash \mathit{isig}(\mathcal{D}) : \star \to \star$
- $\Gamma' \vdash \mathit{Mono} \cdot \mathit{isig}(\mathcal{D})\mathcal{D}.\mathit{mono}$

Conclude by applying the typing rule for *isD* (Figure 5.6).

Case: 3.

$$\frac{\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D}}{\begin{array}{l}\Gamma \vdash \mathtt{to}/D \in \forall\, X : \star.\, \mathtt{Is}/D \cdot X \Rightarrow X \to D \\ \quad \searrow \chi\ (\forall\, X : \star.\, \mathit{IsD} \cdot \mathit{isig}(\mathcal{D}) \cdot X \Rightarrow X \to \mu(\mathit{isig}(\mathcal{D}))) \\ \quad\quad \text{-}\ \Lambda\, X.\, \Lambda\, x.\, \mathit{elimCast}\ \text{-}(\mathit{toD} \cdot \mathit{isig}(\mathcal{R}) \cdot X\ \text{-}x)\end{array}}$$

Similar to the above case.

Case:

$$\frac{\begin{array}{c}\Gamma \vdash t \in S \searrow t' \\ \mathfrak{M} =_{\mathbf{df}} \mathsf{record}\ \mathit{MotiveSrc}\ \{S = S; s = s; T = T\} \\ \mathfrak{P} =_{\mathbf{df}} \mathsf{record}\ \mathit{PattSrc}\ \{S = S; s = s\} \\ \mathfrak{B} =_{\mathbf{df}} \mathsf{record}\ \mathit{BranchSrc}\ \{\gamma_n(i) = (c_i); (\varrho_i = \varrho_i)_{i \in \{1 \ldots n\}}; (t_i = t_i)_{i \in \{1 \ldots n\}}\} \\ \Gamma \vdash \mathfrak{M} \in \mathit{Decl}[D, \Delta] \searrow \mathcal{M} \quad \Gamma \vdash (\mathit{Decl}[D, \Delta], \mathfrak{M}, \mathfrak{P}) \ni \mathfrak{B} \searrow \mathcal{B}\end{array}}{\begin{array}{l}\Gamma \vdash \sigma\langle s \rangle\ t\ @T\ \{\ c_i\ \varrho_i \to t_i\}_{i \in \{1 \ldots n\}} \in T\ (\mathtt{to}/D\ \text{-}s\ t) \\ \quad \searrow \mathit{sigma} \cdot \mathit{isig}(\mathcal{M})\ \text{-}\mathcal{M}.\mathit{mono} \cdot \mathcal{M}.S\ \text{-}\mathcal{M}.s\ t' \cdot \mathcal{M}.T\ \mathit{ctree}(\mathcal{M}, \mathcal{B})\end{array}}$$

- By the IH (3.), exists S' with $\Gamma \vdash S : \star \searrow S'$ and $\Gamma' \vdash t' \in S'$.
- Invert the premise and inline the record projections

$$\frac{\begin{array}{c}y \notin \mathit{DV}(\Gamma) \quad \Gamma \vdash S : \star \searrow S' \quad \Gamma \vdash s \overset{\dashrightarrow^*}{\in} \mathtt{Is}/D \cdot S_1 \searrow s' \quad S_1 \cong S \\ \Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D} \quad \Gamma \vdash T : \kappa \searrow T' \quad \kappa \cong D \to \star\end{array}}{\Gamma \vdash \mathfrak{M} \in \mathit{Decl}[D, \Delta] \searrow \mathsf{record}\ \mathit{Motive}\ \{\mathcal{D} = \mathcal{D}; S = S'; s = s'; T = T'; y = y\}}$$

– By Proposition E.12, we can use the same meta-variable S' for the elaboration of S.

- By the IH (3.) and inversion, $\Gamma' \vdash s' \in \mathit{IsD} \cdot \mathit{isig}(\mathcal{D}) \cdot S_1'$ where $\Gamma \vdash S_1 \searrow S_1'$.
  By Lemma E.8, we may use the same meta-variable $\mathcal{D}$ as in the later premise.
- By Lemma F.12, $S_1' \cong S'$.
- By Proposition F.1, $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$. Invoke Theorem 5.10 with global and local assumptions and the IH to obtain $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{D})$ *valid*.
- By the IH (2.), $\Gamma' \vdash T' : \kappa'$ where $\Gamma \vdash \kappa \searrow \kappa'$.
  By Lemma F.12 and inversion, $\kappa' \cong \mu(\mathit{isig}(\mathcal{D})) \to \star$.

- Invert the premise and inline the record projections

$$\frac{\begin{array}{c}\mathfrak{B}.\gamma = \mathit{Seq}(\Delta) \quad (\vdash \varrho_i := \Delta_i)_{(c_i : \Pi\, \Delta_i.\, D) \in \Delta} \\ \Gamma \vdash S : \star \searrow S' \quad \Gamma \vdash s \overset{\dashrightarrow^*}{\in} \mathtt{Is}/D \cdot S_1 \searrow s' \quad S_1 \cong S \\ (\Gamma \vdash \Delta_i[S/D] \leqslant_{\mathtt{to}/D\ \text{-}\mathfrak{P}.s} \Delta_i \rightsquigarrow \xi_i)_{(c_i : \Pi\, \Delta_i.\, D) \in \Delta} \\ (\Gamma \vdash \Pi\, \Delta_i[S/D].\, T\ (c_i\ (\xi_i \odot \Delta_i)) \ni \lambda\, \varrho_i.\, \mathfrak{B}.t_i \searrow t_i')_{c_i : \Pi\, \Delta_i.\, D} \\ \underset{i \leq \#\Delta}{\tau(i) =_{\mathbf{df}} (t_i')}\end{array}}{\Gamma \vdash (\mathit{Decl}[D, \Delta], \mathfrak{M}, \mathfrak{L}) \ni \mathfrak{B} \searrow \mathsf{record}\ \mathit{Branch}\ \{S = S'; s = s'; \tau = \tau\}}$$

  - Invoke Theorem 4.18 with
    - $\Gamma \vdash \Delta_i[S/D] \searrow \Delta_i'[S'/R]$ and $\Gamma \vdash \Delta_i \searrow \Delta_i'[\mathcal{D}.D/R]$, where $\Gamma, R : \star \vdash \Delta_i[R/D] \searrow \Delta_i'$. This is obtained by Lemma F.9.
    - $\Gamma \vdash \mathtt{to}/D\ \text{-}s : S_1 \to D$ with $|\mathtt{to}/D\ \text{-}s| = \lambda\, x.\, x$

    In particular, this yields $\Gamma, \Delta_i[S/D] \vdash \xi_i \odot \Delta_i : \Delta_i \searrow \xi_i'$ for $i \in \{1 \ldots \#\Delta\}$.
  - Obtain $\vdash \Gamma, \Delta_i[S/D] \searrow \Gamma', \Delta_i'[S'/D]$
  - Obtain $\Gamma, \Delta_i[S/D] \vdash T\ (c_i\ (\xi_i\ \Delta_i)) : \star \searrow T'\ (\mathcal{D}.\theta(i)\ \xi_i')$ (this is just a matter of stringing together the appropriate elaborating inference rule from earlier obtained results).
  - Obtain $\Gamma \vdash \Pi\, \Delta_i[S/D].\, T\ (c_i\ (\xi_i \odot \Delta_i)) : \star \searrow \Pi\, \Delta_i[S/D].\, T'\ (\mathcal{D}.\theta(i)\ \xi_i')$
  - Invoke the IH (4.) and apply inversion to obtain $\Gamma \vdash \Pi\, \Delta_i[S/D].\, T'\ (\mathcal{D}.\theta(i)\ \xi_i') \ni \lambda\, \varrho_i.\, t_i''$ where $t_i' = \lambda\, \varrho_i.\, t_i''$.
    Note that this is only possible because of the premise $\vdash \varrho_i := \Delta_i$.

Now, to finish off the case. Exhibit type $T'\ (\mathit{toD}\ \text{-}s'\ t')$, which is clearly the elaboration of the synthesized type. What remains is checking this is the type synthesized by the elaborated term. Invoke the typing rule for *sigma* (Figure 5.6) with

- $F := \mathit{isig}(\mathcal{M}) = \mathit{isig}(\mathcal{D})$
- $t_1 := \mathcal{M}.\mathit{mono} = \mathcal{D}.\mathit{mono}$
- $S := \mathcal{M}.S = S'$
- $s := \mathcal{M}.s = s'$
- $T := T'$
- $t_3 := \mathit{ctree}(\mathcal{M}, \mathcal{B})$

We have everything we need to obtain $\Gamma \vdash (\mathit{Decl}[D, \Delta], \mathfrak{M}, \mathfrak{P}, \mathfrak{B}) \searrow (\Gamma', \mathcal{M}, \mathcal{B})$ *valid*, so invoke Proposition 5.27, which gives us what we need.

Case:

$$\frac{\begin{array}{c}\Gamma \vdash t \overset{\dashrightarrow^*}{\in} D \searrow t' \quad (\mathtt{isType}/x \notin FV(|t_i|))_{i \in \{1 \ldots n\}} \\ \begin{array}{rcl}\mathfrak{M} & =_{\mathbf{df}} & \mathsf{record}\ \mathit{MotiveSrc}\ \{T = T; S = D; s = \mathtt{is}/D\} \\ \mathfrak{P} & =_{\mathbf{df}} & \mathsf{record}\ \mathit{PattSrc}\ \{S = \mathtt{Type}/x; s = \mathtt{isType}/x\} \\ \mathfrak{B} & =_{\mathbf{df}} & \mathsf{record}\ \mathit{BranchSrc}\ \{\gamma_n(i) = (c_i); (\varrho_i = \varrho_i)_{i \in \{1 \ldots n\}}; (t_i = t_i)_{i \in \{1 \ldots n\}}\}\end{array} \\ \Gamma \vdash \mathfrak{M} \in \mathit{Data}[D, \Delta] \searrow \mathcal{M} \quad y \notin DV(\Gamma) \cup \{\mathtt{isType}/x, x\} \\ \Gamma, \mathit{MuLoc}(D, \mathfrak{M}, x, y) \vdash (\mathit{Data}[D, \Delta], \mathfrak{M}, \mathfrak{P}) \ni \mathfrak{B} \searrow \mathcal{B}\end{array}}{\begin{array}{l}\Gamma \vdash \mu\ x.\ t\ @T\ \{\ c_i\ \varrho_i \to t_i\}_{i \in \{1 \ldots n\}} \in T\ t \\ \quad \searrow \mathit{mu} \cdot \mathit{isig}(\mathcal{M})\ \text{-}\mathit{mono}(\mathcal{M})\ t' \cdot \mathcal{M}.T\ (\lambda\, \mathit{MuLoc}(D, \mathfrak{M}, x, y).\ \mathit{ctree}(\mathcal{M}, \mathcal{B}))\end{array}}$$

- By the IH (3.) and inversion, $\Gamma' \vdash t' \in \mathcal{D}.D$ where $\Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$.
- Invert the premise and inline the record projections

$$\frac{\begin{array}{c}y \notin DV(\Gamma) \quad \Gamma \vdash D : \star \searrow \mathcal{D}.D \quad \Gamma \vdash \mathtt{is}/D \overset{\dashrightarrow^*}{\in} \mathtt{Is}/D \cdot D \searrow \mathit{isD} \cdot \mathit{isig}(\mathcal{D})\ \text{-}\mathcal{D}.\mathit{mono} \\ \Gamma \vdash \mathit{Data}[D] : \mathit{Decl}[D, \Delta] \searrow \mathcal{D} \quad \Gamma \vdash T : \kappa \searrow T' \quad \kappa \cong D \to \star\end{array}}{\Gamma \vdash \mathfrak{M} \in \mathit{Decl}[D, \Delta] \searrow \mathsf{record}\ \mathit{Motive}\ \{\mathcal{D} = \mathcal{D}; S = S'; s = s'; T = T'; y = y\}}$$

  - By Lemma E.8, we can use the same meta-variable $\mathcal{D}$ as earlier.
  - By Proposition F.1, $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow \mathcal{D}$. Invoke Theorem 5.10 with global and local assumptions and the IH to obtain $\Gamma \vdash \mathit{Decl}[D, \Delta] \searrow (\Gamma', \mathcal{D})$ *valid*.
  - By the IH (2.), $\Gamma' \vdash T' : \kappa'$ where $\Gamma \vdash \kappa \searrow \kappa'$.
    By Lemma F.12, $\kappa' \cong \mathcal{D}.D \to \star$.

- Invert the premise and inline the record projections

$$\frac{\begin{array}{c}\mathfrak{B}.\gamma = \mathit{Seq}(\Delta) \quad (\vdash \varrho_i := \Delta_i)_{(c_i : \Pi\, \Delta_i.\, D) \in \Delta} \\ \Gamma, \mathit{MuLoc}(D, \mathfrak{M}, x, y) \vdash \mathtt{Type}/x : \star \searrow \mathtt{Type}/x \\ \Gamma, \mathit{MuLoc}(D, \mathfrak{M}, x, y) \vdash \mathtt{isType}/x \overset{\dashrightarrow^*}{\in} \mathtt{Is}/D \cdot \mathtt{Type}/x \searrow \mathtt{isType}/x \\ \big(\Gamma, \mathit{MuLoc}(D, \mathfrak{M}, x, y) \vdash \Delta_i[\mathtt{Type}/x/D] \leqslant_{\mathtt{to}/D\ \text{-}\mathtt{isType}/x} \Delta_i \rightsquigarrow \xi_i\big)_{(c_i : \Pi\, \Delta_i.\, D) \in \Delta} \\ \big(\Gamma, \mathit{MuLoc}(D, \mathfrak{M}, x, y) \vdash \Pi\, \Delta_i[\mathtt{Type}/x/D].\, T\ (c_i\ (\xi_i \odot \Delta_i)) \ni \lambda\, \mathfrak{B}.\varrho_i.\, \mathfrak{B}.t_i \searrow t_i'\big)_{c_i : \Pi\, \Delta_i.\, D} \\ \underset{i \leq \#\Delta}{\tau(i)} =_{\mathbf{df}} (t_i')\end{array}}{\Gamma \vdash (\mathit{Decl}[D, \Delta], \mathfrak{M}, \mathfrak{P}) \ni \mathfrak{B} \searrow \mathsf{record}\ \mathit{Branch}\ \{S = S'; s = s'; \tau = \tau\}}$$

  - By Lemma E.4, $\mathtt{isType}/x \notin FV(|t_i'|)$
  - Invoke Theorem 4.18 with
    - $\Gamma, \mathit{MuLoc}(D, \mathfrak{M}, x, y) \vdash \Delta_i[\mathtt{Type}/x/D] \searrow \Delta_i'[\mathtt{Type}/x/R]$ and $\Gamma, \mathit{MuLoc}(D, \mathfrak{M}, x, y) \vdash \Delta_i[\mathcal{D}.D/R]$, where $\Gamma, \mathit{MuLoc}(D, \mathfrak{M}, x, y), R : \star \vdash \Delta_i[R/D] \searrow \Delta_i'$.
      This is obtained by Lemma F.9.

  - $\Gamma, \mathit{MuLoc}(D, \mathfrak{M}, x, y) \vdash \texttt{to/}D\ \texttt{-isType/}x \in \texttt{Type/}x \to D$
    with $|\texttt{to/}D\ \texttt{-isType/}x| =_{\beta\eta} \lambda\, x.\, x$
    In particular, this yields $\Gamma, \mathit{MuLoc}(D, \mathfrak{M}, x, y), \Delta_i[\texttt{Type/}x/D] \vdash \xi_i \odot \Delta_i : \Delta_i \searrow \xi_i'$ for $i \in \{1 \ldots \#\Delta\}$
- Obtain $\vdash \Gamma, \mathit{MuLoc}(D, \mathfrak{M}, x, y), \Delta_i[\texttt{Type/}x/D] \searrow \Gamma', \texttt{Type/}x\star : , \texttt{isType/}x : \mathit{IsD} \cdot \mathit{isig}(\mathcal{D}), x : \Pi\, y : \texttt{Type/}x.\, T'\ (\mathit{elimCast}\ \texttt{-}\ldots\ y), \Delta_i'[\texttt{Type/}x/R]$
- $\Gamma, \mathit{MuLoc}(D, \mathfrak{M}, x, y)\Delta_i[\texttt{Type/}x/D] \vdash T\ (c_i\ (\xi_i\ \Delta_i)) : \star \searrow T'\ (\mathcal{D}.\theta(i)\ \xi_i')$ (this is just a matter of stringing together the appropriate elaborating inference rule from earlier obtained results).
- Obtain $\Gamma, \mathit{MuLoc}(D, \mathfrak{M}, x, y) \vdash \Pi\, \Delta_i[\texttt{Type/}x/D].\, T\ (c_i\ (\xi_i \odot \Delta_i)) : \star \searrow \Pi\, \Delta_i[\texttt{Type/}x/D].\, T'\ (\mathcal{D}.\theta(i)\ \xi_i')$
- Invoke the IH (4.) and apply inversion to obtain $\Gamma, \mathit{MuLoc}(D, \mathfrak{M}, x, y) \vdash \Pi\, \Delta_i[\texttt{Type/}x/D].\, T'\ (\mathcal{D}.\theta(i)\ \xi_i') \ni \lambda\, \varrho_i.\, t_i''$ where $t_i' = \lambda\, \varrho_i.\, t_i''$.
  Note that this is only possible because of the premise $\vdash \varrho_i := \Delta_i$.

Now, to finish off the case. Exhibit type $T'\ t'$, which is clearly the elaboration of the synthesized type. What remains is checking this is the type synthesized by the elaborated term.

Invoke the typing rule for *mu* (Figure 5.6) with

- $F := \mathit{isig}(\mathcal{M}) = \mathit{isig}(\mathcal{D})$
- $t_1 := \mathcal{M}.\mathit{mono} = \mathcal{D}.\mathit{mono}$
- $t_2 := t'$
- $T = T'$
- $t_3 := \lambda\, \mathit{MuLoc}(D, \mathfrak{M}, x, y).\, \mathit{ctree}(\mathcal{M}, \mathcal{B})$
  We have everything we need to obtain $\Gamma, \mathit{MuLoc}(D, \mathfrak{M}, x, y) \vdash (\mathit{Decl}[D, \Delta], \mathfrak{M}, \mathfrak{P}, \mathfrak{B}) \searrow (\Gamma', \mathcal{M}, \mathcal{B})\ \mathit{valid}$, so invoke Proposition 5.27 and then λ-abstract, which gives us what we need to conclude the case.

□

REFERENCES

[1] Andreas Abel. MiniAgda: Integrating sized and dependent types. In Ekaterina Komendantskaya, Ana Bove, and Milad Niqui, editors, *Partiality and Recursion in Interactive Theorem Provers, PAR@ITP 2010, Edinburgh, UK, July 15, 2010*, volume 5 of *EPiC Series*, pages 18–33. EasyChair, 2010.

[2] Andreas Abel and Thierry Coquand. Failure of normalization in impredicative type theory with proof-irrelevant propositional equality, 2019.

[3] Pedro Abreu, Benjamin Delaware, Alex Hubers, Christa Jenkins, J. Garrett Morris, and Aaron Stump. A type-based approach to divide-and-conquer recursion in coq. *Proc. ACM Program. Lang.*, 7(POPL):61–90, 2023.

[4] Ki Yung Ahn. *The Nax Language: Unifying Functional Programming and Logical Reasoning in a Language based on Mendler-style Recursion Schemes and Term-indexed Types*. PhD thesis, Portland State University, 2014.

[5] Ki Yung Ahn and Tim Sheard. A hierarchy of Mendler style recursion combinators: taming inductive datatypes with negative occurrences. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 234–246, New York, NY, USA, 2011. ACM.

[6] Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and E. Moran. Innovations in computational type theory using NuPRL. *J. Applied Logic*, 4(4):428–469, 2006.

[7] Robert Atkey. Syntax and semantics of quantitative type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65. ACM, 2018.

[8] Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, 2013.

[9] Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. In Roberto M. Amadio, editor, *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, volume 4962 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 2008.

[10] Gilles Barthe and Maria João Frade. Constructor subtyping. In S. Doaitse Swierstra, editor, *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*, volume 1576 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 1999.

[11] Gilles Barthe, Maria João Frade, Eduardo Giménez, Luís Pinto, and Tarmo Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004.

[12] Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nord. J. Comput.*, 10(4):265–289, 2003.

[13] Richard S. Bird and Oege de Moor. *Algebra of programming.* Prentice Hall International series in computer science. Prentice Hall, 1997.

[14] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[15] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: An experimental applicative language, 1980.

[16] C. Böhm, M. Dezani-Ciancaglini, P. Peretti, and S.Ronchi Della Rocca. A discrimination algorithm inside λ-β-calculus. *Theoretical Computer Science*, 8(3):271 – 291, 1979.

[17] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 33–46. ACM, 2014.

[18] Mario Cerneiro. The type theory of lean, 2019.

[19] Jonathan H.W. Chan. *Sized dependent types via extensional type theory.* PhD thesis, University of British Columbia, 2022.

[20] Alonzo Church. *The calculi of lambda-conversion.* Princeton University Press, 1941.

[21] Jesper Cockx. Yet another way `--without-K` is incompatible with univalence. URL `https://lists.chalmers.se/pipermail/agda/2014/006367.html`. On the Agda mailing list, 2014.

[22] Jesper Cockx. *Dependent Pattern Matching and Proof-Relevant Unification.* PhD thesis, Katholieke Universiteit Leuven, Belgium, 2017.

[23] Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, Robert Harper, Douglas J. Howe, Todd B. Knoblock, N. P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing mathematics with the Nuprl proof development system.* Prentice Hall, 1986.

[24] Thierry Coquand. Pattern matching with dependent types. In Bengt Nordström, Kent Petersson, and Gordon Plotkin, editors, *Electronic Proceedings of the Third Annual Workshop on Logical Frameworks*, volume 92, pages 66–79, Båstad, Sweden, 1992.

[25] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95 – 120, 1988.

[26] Karl Crary. Foundations for the implementation of higher-order subtyping. In Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman, editors, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*, pages 125–135. ACM, 1997.

[27] Pierre-Évariste Dagand. *A cosmology of datatypes : reusability and dependent types.* PhD thesis, University of Strathclyde, Glasgow, UK, 2013.

[28] Pierre-Evariste Dagand and Conor McBride. Elaborating inductive definitions, 2012.

[29] Pierre-Évariste Dagand and Conor McBride. Transporting functions across ornaments. *J. Funct. Program.*, 24(2-3):316–383, 2014.

[30] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[31] Larry Diehl, Denis Firsov, and Aaron Stump. Generic zero-cost reuse for dependent types. *Proc. ACM Program. Lang.*, 2(ICFP):104:1–104:30, jul 2018.

[32] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Comput. Surv.*, 54(5):98:1–98:38, 2022.

[33] Peter Dybjer and Erik Palmgren. Intuitionistic Type Theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2020 edition, 2020.

[34] Denis Firsov, Richard Blair, and Aaron Stump. Efficient Mendler-style lambda-encodings in Cedille. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 235–252, Cham, 2018. Springer International Publishing.

[35] Denis Firsov, Larry Diehl, Christopher Jenkins, and Aaron Stump. Course-of-value induction in Cedille, 2018.

[36] Denis Firsov and Aaron Stump. Generic derivation of induction for impredicative encodings in cedille. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 215–227, New York, NY, USA, 2018. Association for Computing Machinery.

[37] Herman Geuvers. Induction is not derivable in second order dependent type theory. In Samson Abramsky, editor, *International Conference on Typed Lambda Calculi and Applications*, pages 166–181, Berlin, Heidelberg, 2001. Springer.

[38] Herman Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(Part 1):3–25, 2009.

[39] Herman Geuvers. The Church-Scott representation of inductive and coinductive data. (unpublished manuscript), 2014.

[40] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan M. Smith, editors, *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 1994.

[41] Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, *Algebra, Meaning, and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, volume 4060 of *Lecture Notes in Computer Science*, pages 521–540, Berlin, Heidelberg, 2006. Springer.

[42] Tatsuya Hagino. *Categorical programming language*. PhD thesis, The University of Edinburgh, 1987.

[43] Christa Jenkins, Denis Firsov, Larry Diehl, Colin McDonald, and Aaron Stump. Elaborating course-of-values induction in Cedille. `https://cwjnkins.github.io/assets/JFDMS20_Elaborating-CV-Induction.pdf`, 2020.

[44] Christa Jenkins, Andrew Marmaduke, and Aaron Stump. Simulating large eliminations in cedille. In Henning Basold, Jesper Cockx, and Silvia Ghilezan, editors, *27th International Conference on Types for Proofs and Programs, TYPES 2021, June 14-18, 2021, Leiden, The Netherlands (Virtual Conference)*, volume 239 of *LIPIcs*, pages 9:1–9:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[45] Christopher Jenkins, Andrew Marmaduke, and Aaron Stump. Simulating large eliminations in Cedille. `https://cwjnkins.github.io/`, 2021.

[46] Christopher Jenkins, Colin McDonald, and Aaron Stump. Elaborating inductive datatypes and course-of-values pattern matching to Cedille. *arXiv preprint arXiv:1903.08233*, 2019.

[47] Christopher Jenkins and Aaron Stump. Monotone recursive types and recursive data representations in Cedille. Under consideration for J. Mathematically Strucured Computer Science, 2020.

[48] Christopher Jenkins and Aaron Stump. Monotone recursive types and recursive data representations in Cedille. *Mathematical Structures in Computer Science*, page 1–64, Dec 2021.

[49] Christopher Jenkins, Aaron Stump, and Larry Diehl. Efficient lambda encodings for mendler-style coinductive types in cedille. In Max S. New and Sam Lindley, editors, *Proceedings Eighth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2020, Dublin, Ireland, 25th April 2020*, volume 317 of *EPTCS*, pages 72–97, 2020.

[50] S.C. Kleene. Classical Extensions of Intuitionistic Mathematics. In Y. Bar-Hillel, editor, *LMPS 2*, pages 31–44. North-Holland Publishing Company, 1965.

[51] Stephen Cole Kleene. *Introduction to Metamathematics*. Princeton, NJ, USA: North Holland, 1952.

[52] Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. In *Proceedings of 18th IEEE Symposium on Logic in Computer Science (LICS 2003), 22-25 June 2003, Ottawa, Canada*, LICS '03, pages 86–95. IEEE Computer Society, 2003.

[53] Jean-Louis Krivine and Michel Parigot. Programming with proofs. *J. Inf. Process. Cybern.*, 26(3):149–167, 1990.

[54] Joachim Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103(2):151–161, 1968.

[55] Daniel Leivant. Reasoning about functional programs and complexity classes associated with type disciplines. In *24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, 7-9 November 1983*, pages 460–469. IEEE Computer Society, 1983.

[56] Zhaohui Luo. Ecc, an extended calculus of constructions. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 386–395. IEEE Computer Society, 1989.

[57] Zhaohui Luo. Coercive subtyping. *J. Logic and Computation*, 9(1):105–130, 1999.

[58] Grant Malcolm. Data structures and program transformation. *Sci. Comput. Program.*, 14(2-3):255–279, 1990.

[59] Andrew Marmaduke, Christopher Jenkins, and Aaron Stump. Zero-cost constructor subtyping. In *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages*, IFL 2020, page 93–103, New York, NY, USA, 2020. Association for Computing Machinery.

[60] Ralph Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*. PhD thesis, Ludwig-Maximilians-Universität München, 1998.

[61] Conor McBride. Ornamental algebras, algebraic ornaments. Manuscript, 2011.

[62] Conor McBride. I got plenty o' nuttin'. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016.

[63] Nax P Mendler. Recursive types and type constraints in second-order lambda calculus. In *LICS*, volume 87, pages 30–36, 1987.

[64] Nax Paul Mendler. Predicative type universes and primitive recursion. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, pages 173–184. IEEE Computer Society, 1991.

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

[66] James Hiram Morris Jr. *Lambda-calculus models of programming languages*. PhD thesis, Massachusetts Institute of Technology, 1969.

[67] Bengt Nordstrom, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. International Series of Monographs on Computer Science. Oxford University Press, USA, 1990.

[68] Ulf Norell, Andreas Abel, and Nils Anders Danielsson. Release notes for Agda 2 version 2.3.2. URL `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.Version-2-3-2`, 2012.

[69] Michel Parigot. Programming with proofs: A second order type theory. In H. Ganzinger, editor, *ESOP '88*, pages 145–159, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[70] Michel Parigot. On the representation of data in lambda-calculus. In Egon Börger, Hans Kleine Büning, and Michael M. Richter, editors, *CSL '89*, pages 309–321, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[71] Frank Pfenning. On bidirectional type checking (lecture notes). `https://www.cs.cmu.edu/~fp/courses/15312-f04/handouts/15-bidirectional.pdf`, October 2001.

[72] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[73] Dana Scott. A system of functional abstraction (1968). lectures delivered at university of california, berkeley. *Cal*, 63, 1962.

[74] Dana Scott. Lambda calculus: Some models, some philosophy. In Jon Barwise, H. Jerome Keisler, and Kenneth Kunen, editors, *The Kleene Symposium*, volume 101 of *Studies in Logic and the Foundations of Mathematics*, pages 223–265. Elsevier, 1980.

[75] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Elsevier, 2006.

[76] Zdzisław Spławski and Paweł Urzyczyn. Type fixpoints: Iteration vs. recursion. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, ICFP '99, page 102–113, New York, NY, USA, 1999. Association for Computing Machinery.

[77] Thomas Streicher. *Investigations into intensional type theory*. Habilitation, Ludwig Maximilian Universität, 1993.

[78] Aaron Stump. *Programming language foundations*. John Wiley & Sons, 2013.

[79] Aaron Stump. The calculus of dependent lambda eliminations. *J. Funct. Program.*, 27:e14, 2017.

[80] Aaron Stump. From realizability to induction via dependent intersection. *Ann. Pure Appl. Logic*, 169(7):637–655, 2018.

[81] Aaron Stump. Syntax and typing for Cedille core. 2018.

[82] Aaron Stump and Peng Fu. Efficiency of lambda-encodings in total type theory. *J. Funct. Program.*, 26:e3, 2016.

[83] Aaron Stump and Christopher Jenkins. Syntax and semantics of Cedille. 2018.

[84] Aaron Stump, Christopher Jenkins, Stephan Spahn, and Colin McDonald. Strong functional pearl: Harper's regular-expression matcher in Cedille. *Proc. ACM Program. Lang.*, 4(ICFP):122:1–122:25, 2020.

[85] Agda Development Team. Equality is incompatible with sized types (issue 2820). GitHub, Oct 2017.

[86] The Coq Development Team. *The Coq Reference Manual, version 8.13*, 2021. Available electronically at `https://coq.github.io/doc/v8.13/refman/`.

[87] The Coq Development Team. *The Coq Reference Manual, version 8.13*, 2021. Available electronically at `https://coq.github.io/doc/v8.13/refman/`.

[88] D. A. Turner. Elementary strong functional programming. In *Functional Programming Languages in Education, First International Symposium, FPLE'95, Nijmegen, The Netherlands, December 4-6, 1995, Proceedings*, pages 1–13, 1995.

[89] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[90] Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nordic J. of Computing*, 6(3):343–361, sep 1999.

[91] Tarmo Uustalu and Varmo Vene. Coding recursion a la Mendler (extended abstract). In *Proc. of the 2nd Workshop on Generic Programming, WGP 2000, Technical Report UU-CS-2000-19*, pages 69–85. Dept. of Computer Science, Utrecht University, 2000.

[92] Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, page 347–359, New York, NY, USA, 1989. ACM.

[93] DHD Warren. *Applied Logic – Its Use and Implementation as a Programming Language Tool*. PhD thesis, Univ. of Edinburgh, 1977. PhD. Thesis.

[94] Stephanie Weirich and Chris Casinghino. Generic programming with dependent types. In Jeremy Gibbons, editor, *Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, volume 7470 of *Lecture Notes in Computer Science*, pages 217–258. Springer, 2010.

[95] Benjamin Werner. *Une théorie des constructions inductives*. PhD thesis, 1994.

[96] Zhixuan Yang and Nicolas Wu. Fantastic morphisms and where to find them - A guide to recursion schemes. In Ekaterina Komendantskaya, editor, *Mathematics of Program Construction - 14th International Conference, MPC 2022, Tbilisi, Georgia, September 26-28, 2022, Proceedings*, volume 13544 of *Lecture Notes in Computer Science*, pages 222–267. Springer, 2022.

Index