Well-founded Recursion in Terms and Types

Rohan Ben Jacob-Rao
School of Computer Science
McGill University, Montreal

August 2017

A thesis submitted to McGill University in partial fulfillment of
the requirements of the degree of Master of Science.

Abstract

Recursion is a fundamental tool for writing useful programs in functional languages. When viewed from a logical perspective via the Curry-Howard correspondence, well-founded recursion corresponds to inductive reasoning. Another concept in programming languages is that of *indexed types*, which allows one to express fine-grained program properties using terms from a separate index language. From a logical perspective, programming with indexed types corresponds to first-order reasoning. In this thesis, we explore the combination of well-founded recursion and indexed types, which is manifested on two levels: firstly as a program-level construct for traversing indexed data types (providing inductive proofs of first-order properties) and secondly as a type-level construct for defining further indexed types.

Concretely, our contribution is a core language called Tores that features indexed types and inductive reasoning via well-founded recursion. The primary forms of types are *indexed recursive types*, over which we support reasoning via Mendler-style recursion. Additionally, Tores features *index-stratified types*, which allow further definitions of types via well-founded recursion over indices. The main difference between the two forms is that recursive types are more flexible, allowing induction, while stratified types only support unfolding based on their indices. The combination of the two features is especially powerful for formalizing metatheory involving logical relations. This is partly because type definitions in Tores do not require positivity, a condition used in other systems to ensure termination and in turn logical consistency. Despite this, we are able to prove termination of Tores programs using a semantic interpretation of types. Moreover, we believe that our development is modular and extensible to sophisticated index languages and reasoning techniques.

Abrégé

La récursivité est un outil fondamental pour l'écriture en langages fonctionnels de nombreux programmes utiles. À travers la correspondance de Curry-Howard, une fonction récursive bien fondée se traduit en raisonnement inductif. Un autre concept des langages de programmation sont les types indexés qui permettent d'exprimer des propriétés précises des programmes en annotant le type par un argument d'un langage donné. D'un point de vue logique, la programmation avec des types indexés représente la logique de premier ordre. Dans ce mémoire, nous explorerons la combinaison de fonction récursives bien fondées avec les types indexés. Ceux-ci apparaîtrons de deux façons: en tant construction au niveau des programmes nous permettant de traverser les types indexés classifiant des données, ainsi qu'en tant que construction au niveau des types permettant de définir de nouveaux types indexés.

Concrètement, notre contribution est un langage interne nommé Tores qui fait usage des types indexés et de raisonnement inductif grâce aux functions récursives bien fondées. Les principaux types utilisés sont les *types récursifs indexés* qui supportent un raisonnement logique à travers la récursivity au sens de Mendler. De surcroît, Tores possède des *types aux annotations stratifiés* qui permettent des définitions récursives additionelles. La différence principale entre les deux est qu'alors que les types récursifs sont plus flexibles et offrent un principe d'induction, les types stratifiés ne permettent seulement une récursivité basée sur les annotations. Cependant, la combinaison de ces deux sortes offre une puissance considérable quand il vient temps de formaliser des méta-théories avec des relations logiques. Une des raisons pour cette puissance est parce que les définitions des types de Tores ne requièrent pas la positivité, une condition utilisées par d'autres systèmes de façon à s'assurer de la terminaison et donc de la cohérence logique. Malgré cela, nous sommes en mesure de prouver la terminaison des programmes écrits dans Tores en utilisant une interprétation sémantique des types. De plus, nous croyons que notre système est modulaire et qu'il est possible de l'enrichir avec des langages d'annotations et des méthodes de raisonnement plus sophistiqués.

Acknowledgments

Firstly I would like to thank my academic advisor, Prof. Brigitte Pientka, who put her faith in me as a student from the outset. Brigitte patiently allowed me to explore different research paths until we settled on a fruitful and mutually engaging topic. Her financial support gave me the opportunity to live in an entirely different part of the world, attend stimulating conferences and make lasting personal connections. From writing papers together, I learned a lot about how to motivate and present my work.

Next, I would like to thank my unofficial academic mentors, Andrew Cave and Francisco Ferreira. Andrew's early guidance on a class project was the catalyst for our joint work on Mendler recursion in Chapter 2. Attending the POPL 2016 conference with Andrew inspired me to continue his work on stratified types, which led to the results in Chapter 3. I also owe much to Francisco, for his technical advice as well as his support with the meta-struggles of graduate school. I could not have succeeded without the help of these two. I would also like to thank the rest of McGill's Complogic Lab: David Thibodeau, Stefan Knudsen, Shawn Otis and (past member) Steven Thephsourinthone. Shawn and Steven in particular were great companions in the starts of our research careers. Everyone in the Complogic Lab made it a fun and welcoming place to work.

Importantly, I would like to thank my friends and family across the globe. In Montreal, my roommates Selina Liu and Sophie Silkes were great emotional supports. Kerwin Wong was my partner in crime. Back in Sydney, Gaspar Tse and Stella Halena are my second family and supported me as I finished out my thesis. My parents Dilip Rao and Angelica Jacob have given me every opportunity in the world and encouraged me at every step.

Finally, I would like to thank the external reviewer of this thesis, Prof. Alwen Tiu of Nanyang Technological University, for his rigorous review and helpful comments.

Contents

Chapter 1

Introduction

1.1 Recursion in types

Recursion is a fundamental concept of computer science. As a programming technique, it can be used to concisely and elegantly implement a range of algorithms. It is particularly amenable to inductive reasoning, allowing programmers to prove (formally or informally) properties about the behaviour and performance of their programs. Because of this, it is adopted by *functional* programming languages as the primary method for writing iterating programs.

Recursion does not only appear in programs: it is also used in the design of many popular data structures. A simple example is that of a linked list: it consists of a "head" element followed by a "tail" which is itself a list. We can encode this as a data type in a functional language (using the syntax of the ML family of languages) as follows.

```
data list = Nil | Cons of element * list
```

Here `Nil` corresponds to the empty list, and `Cons` constructs a nonempty list with a head of type `element` and a tail of the same type `list` that we are defining. This syntax gives a clear and concise definition of a linked list.

Similarly, we can define a basic binary tree as either an empty tree or a root node with an element and a left and a right subtree.

```
data tree = Empty | Node of tree * element * tree
```

These structures, called *recursive types*, are pervasive in typed functional programming. With such definitions, the common tasks of building and traversing the corresponding data generally have elegant solutions as recursive programs.

There is yet another way we could use recursion to define a data type. Consider defining a type that depends on a natural number, which we call an *index* [Zenger, 1997, Xi and Pfenning, 1999]. This index

can track extra information about data that belongs to the associated type. For example, a list type could contain a numeric index representing the length of the list.

Now, how could we define such an indexed type? One way is by a simple recursion on the natural number index. This means that we must give a base case for the index 0 and a recursive case for indices $n+1$. Using a fictional syntax for now, we can define a length-indexed list `ilist` as follows.

```
ilist 0 = unit
ilist (n + 1) = element * ilist n
```

Here `unit` simply refers to a type with no information, representing the empty list.

The structure of `ilist` is much the same as `list` above; the difference is that the length is now tracked in the type. Theoretically, the advantage of such a type is that we can state finer-grained properties of our programs using the types. In such a framework, we could express list functions that are truly type-safe, i.e. which do not cause run-time exceptions or errors for lists of unexpected lengths (as demonstrated by Xi and Pfenning [1998]).

For example, consider the program `tail` that takes a nonempty list and returns the list without the first element. Using the `list` type above, there is no way to guarantee that the input list is not empty. Thus we would have to return some default value or throw an exception for an empty list input. However, using the type `ilist`, we can express the tail program precisely.

```
tail : ilist (n + 1) -> ilist n
tail (h, t) = t
```

The input type ensures that the list is nonempty, so we only have to handle that case. The program is both shorter and more precise than one using the `list` type.

In this thesis we present specific approaches to supporting both recursive types and types defined by recursion on an index. Our treatment of the latter concept, which we call *index-stratified types*, is the main technical novelty. Though our motivation so far has used examples from programming, our true goal is to provide a sound logical system for writing (inductive) proofs. The next section leads to this idea by explaining the multifaceted nature of types.

1.2 Types and proofs

In designing a programming language, the organizing principle we use is that of *types*. The original concept of types was introduced by Russell in his foundation for mathematics [Whitehead and Russell, 1912]. Before him, Frege had been working on a foundational set theory, in which Russell famously discovered a paradox due to the unbounded quantification of variables over sets. Russell was able to avoid such paradoxes in his theory by qualifying variables in logical formulae by the domains in which they make sense, otherwise known as their types.

In computer programming, a different notion of types has been adopted as a way of categorizing data used in programs. In an imperative language like C [Kernighan, 1988], the main purpose of types is to guide

the compiler to allocate memory correctly and perform optimizations. Their secondary purpose is to provide annotations for programmers and a limited amount of compile-time checking for programming errors.

In languages with more sophisticated type systems, especially functional languages such as those in the ML family [Milner et al., 1997], the second purpose is significantly enhanced. In these languages, types are a trusted tool for designing data structures, programs and entire systems (through the use of modules and interfaces). Used in this way, types can provide a software system with formal specification which is checked automatically at compile-time.

Remarkably, the two different notions of types we mentioned - as a logical device in mathematics and as a structuring tool in programming - are in fact strongly related. The connection between these two concepts is known as the *Curry-Howard correspondence* [Howard, 1980]. The Curry-Howard correspondence states that a type is analogous to a logical proposition. By the same analogy, a program of some type corresponds to a proof of the related proposition. For instance, a program of function type $A \to B$ corresponds to a proof of the logical implication between the propositions corresponding to A and B. The correspondence generally holds up as we scale the types and propositions to more complex systems. For example, a system with indexed types (explained in Section 1.4) can accurately model a first-order logic with universal and existential predicates. Thus we have a powerful two-way connection: our hard work in programming languages can offer valuable perspectives in logic and vice versa. In particular, a programming language we design, if it meets certain conditions, can be used as a computer-checked environment in which to write logical proofs!

The Curry-Howard correspondence raises the bar of what is possible in both mathematics and software. A proof environment as we mentioned enables rigorous formalization of mathematical proofs, avoiding the ambiguities and errors present in the conventional mixture of natural language and mathematical notation. By involving computers in proof writing and checking, we can increase the trust we give to proofs. Perhaps more important in modern society is that we can increase the trust we give to software. Large software systems are notoriously hard to get right, and the consequence of a single error ranges from inconvenient (application crashing) to catastrophic (vehicle crashing). The tools of type systems and proof environments empower us to *prove* properties of software, much stronger than by other methods such as testing. This approach has been successfully exploited to prove correctness of complex software systems such as an operating system [Klein et al., 2009] and a compiler [Leroy, 2009].

In this thesis, we design a programming language in the tradition of the Curry-Howard correspondence. Our main contribution Tores is intended as the foundation of a practical proof environment. In particular, it is designed for writing and reasoning about logical specifications.

1.3 Writing logical specifications

We explained in the previous section that the Curry-Howard correspondence gives a deep connection between programming and mathematics. In this thesis we will exploit the correspondence directly: we will design programming languages with features useful for writing logical specifications and proofs.

In fact we have particular kinds of logical specifications in mind: so-called systems of inference and

their metatheory. A system of inference is anything we can describe by a finite set of inference rules. Its metatheory is what we can prove about that system (the theory). Let's take an example from computer science: a deterministic finite automaton, or DFA. We can describe a particular DFA by specifying the states and transitions as inference rules, which we model as types in our specification (programming) language. Then we can formulate properties about our DFA, such as reachability of certain states or the words accepted by it, again as types in our programming language. We can prove such properties by writing programs that satisfy the corresponding types. These are called metatheoretic properties, collectively the metatheory, because they are proved on top of the object theory (in this case the DFA specification) in a so-called metalanguage (our programming language).

Most of the examples we are interested in are not automata or transition systems, but actually programming languages and logics themselves! The reason is that often when we design a language or logic for a particular application, we would like to formally model it and prove its metatheoretic properties. To avoid confusion, we will try to distinguish between the object language/logic (the one being modelled) and the meta-language/logic (the one in which it is modelled). The ideas we develop in this thesis are specifically designed for reasoning in the metalanguage.

1.4 Technical approach: indexed types

The technical approach taken in this thesis follows the tradition of *indexed types* as used in the programming language and proof environment Beluga [Pientka and Cave, 2015]. The general idea of indexed types is to increase the expressivity of a type system by allowing types to be parameterized by terms from an *index language* or *index domain*. This was pioneered by Zenger [1997] and by Xi and Pfenning [1999] in the design of Dependent ML, where indices are integer constraints which can for example express acceptable sizes or values used in data structures. These constraints are then solved automatically by the type checker.

A different approach to indexed types is used in the proof environment Beluga. Beluga is a two-level framework building on an index language of Contextual LF [Pientka, 2008, Pientka and Dunfield, 2008, Nanevski et al., 2008]. LF [Harper et al., 1993] is a dependently typed specification language with support for issues that commonly occur in logical specifications. These include variable binding using higher-order abstract syntax as well as capture-avoiding substitution. Contextual LF enriches LF terms with their contexts of discourse to allow reasoning about open terms and hypothetical derivation trees. Using Contextual LF as an index domain, Beluga includes a programming language with inductive data types as well as pattern matching and recursion. This allows us to write logical specifications in the index language and inductive proofs about them in the programming language.

Our development of Tores follows this approach to indexed types. In our case we focus on the programming language component and aim to leave the index language abstract. We often use examples involving a simple index language of natural numbers, but lay out general requirements for other more realistic index languages. This means that the programming language features we develop in Tores are applicable to a practical proof environment such as Beluga. In fact, the idea of index-stratified types has already been

implemented in Beluga by the development team (particularly Brigitte Pientka, Tao Xue and Andrew Cave) and used in some interesting case studies such as a normalization proof [Cave and Pientka, 2015] and an algorithm for normalization by evaluation [Cave and Pientka, 2012]. A key contribution of our work is to lay the theoretical foundation for this language feature.

1.5 Core language features: Mendler recursion and stratified types

In Section 1.1 we briefly presented the ideas of recursive and index-stratified types from a programming perspective. Now that we have our main application of logical specification and proofs in mind, we can give a better idea of how we intend our language features to be used.

Recursive types are indeed fundamental for defining data structures, and from a logical perspective they are a powerful tool for writing recursive specifications. This is essential for our application of modelling programming languages and logics, where recursive structure abounds. For example, evaluation of terms (programs) in a programming language is defined by evaluating the subterms and combining the results. We would like our metalanguage to support programming with such recursive specifications, which corresponds to writing inductive proofs. There are several approaches to this problem, but we choose one which we believe provides a balance of power, simplicity and genericity. This is the technique of *Mendler-style recursion*.

Mendler-style recursion [Mendler, 1988] is a type-based method for providing sound induction principles over recursively defined types. Its beauty is that it is generic to the type being described: any recursive type automatically provides an induction principle via Mendler recursion. The price of this generality is two-fold. First, the induction principle is relatively weak, corresponding to primitive recursion. Second, our use of recursive types is necessarily restricted: we can reason about them via the induction principle, but we cannot unfold their definition directly. This restriction is important: without it, we could encode nonterminating programs which correspond to proofs of logical fallacies!

This limitation motivates the need for our second core language feature. Though the Mendler induction principle is often sufficient for logical reasoning, occasionally we need to unfold recursive definitions directly. This comes up particularly in a proof technique called *logical relations*, which is commonly used in the metatheory of programming languages. At a basic level, the technique builds a model of the programming language following the structure of its types. Since the type structure is typically recursive, so is the logical relation. However, the relation should be used not by induction but by simply unfolding its definition, which is not allowed in our treatment of recursive types.

With this issue in mind, we developed the notion of a stratified type. This is a recursively defined type which has no principle of induction, but does allow direct unfolding according to its definition. Hence it is ideal for encoding logical relations.

Note that typical treatments of recursive types do allow direct unfolding, but involve a restriction on the formation of recursive types such that type variables can only occur "positively". This treatment, called *inductive types*, is currently used in Beluga. The positivity restriction again prohibits encodings of logical relations, which use type variables "negatively" for modelling function types. Hence the addition of stratified

types is needed regardless of the choice of treatment of recursive types.

1.6 Contributions

This thesis has two main chapters with the following contributions.

In Chapter 2, we provide a minimal account of Mendler-style recursion in a simply typed lambda calculus with recursive types. We prove program termination using a logical predicate argument. The novelty in this work is our definition of the pre-fixed point used in the semantic interpretation of recursive types. Additionally, we describe our mechanization of the termination proof in Coq, which is to our knowledge the first mechanization of the metatheory of Mendler-style recursion. This chapter of the thesis is drawn from a publication with Andrew Cave and Brigitte Pientka at the 11th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice in Porto, Portugal in June 2016. The semantics and proof were developed by Andrew Cave, the mechanization by the author, and the description by the author and Brigitte Pientka.

In Chapter 3 we present Tores, an indexed type system combining the features of indexed recursive types (with Mendler-style recursion), first-class equality as well as index-stratified types. Our formulation of index-stratified types is novel, adapting work on stratified definitions from proof theory to a type-theoretic setting. We prove subject reduction and our main result of termination using a logical predicate semantics. Termination implies the logical consistency of Tores, justifying its use as a proof system. This work complements that of Pientka and Abel [2015], who prove termination of a Beluga-like core language with pattern matching and recursion, as we justify a more powerful type language with recursive and stratified types. This is described in a recent technical report with Brigitte Pientka. The initial ideas and theory of Tores are due to Andrew Cave, and were further developed and described by the author and Brigitte Pientka.

1.7 How to read this thesis

This thesis is designed to cater to both beginners and experts in the areas of programming languages and logic.

For someone less familiar with programming language syntax and semantics, Chapter 2 is a good introduction. It explains Mendler-style recursion in a simply typed lambda calculus using our preferred styles of syntax, operational semantics, logical predicate semantics and termination proof. We reuse this presentation in the later chapter. This chapter ends with a description of our mechanization, which is useful for someone intending to undertake a similar mechanization or who is interested in the issues of reasoning in meta-languages.

Someone who is comfortable with the ideas in Chapter 2 can proceed to Chapter 3. There we provide a comprehensive presentation of our indexed type system, Tores, with Mendler-style recursion, index-stratified types and first-class equality. This language builds on the ideas of Chapter 2 to form a practical proof system.

We again present a proof of termination, generalizing the simply typed case.

The final chapter gives an in-depth comparison to related work in the field, and future directions for our research.

Chapter 2

Simple Recursive Types with Mendler Recursion

2.1 Introduction

2.1.1 Treatments of recursive types

Recursive types are crucial in functional programming languages for defining common data types and in proof systems for supporting inductive reasoning. Such types are defined using recursive formulae that can be unrolled or unfolded to describe arbitrarily large data structures. However, even in the simplest setting of the simply typed lambda calculus with recursive types, one can express well-typed programs that do not terminate. This is a major problem for proof systems, where termination is required for consistency of the logic.

The issue of nontermination arises from the ability to unfold certain exotic recursive types. Proof systems such as Coq [Bertot and Castéran, 2004] and Agda [Norell, 2007] rule out such exotic types by imposing a *strict positivity* condition. This means that recursive type variables cannot occur to the left of a function arrow (which is a negative position). Recursive types with this restriction are called *inductive* as they correspond to inductive definitions in proof theory. From such types, we can systematically derive induction principles [Paulin-Mohring, 1993, Dybjer, 1994] which allow us to write inductive proofs. Similar approaches involving positivity or *functoriality* conditions are taken in proof theoretic foundations that support (co-)inductive definitions [Tiu and Momigliano, 2012, Baelde, 2012].

A different approach to reasoning about recursive types is called *Mendler-style* recursion [Mendler, 1988]. This is a scheme for eliminating recursive types which does not impose a positivity restriction. In this case we cannot directly unfold recursive types (as it would lead to nontermination), but we can still reason inductively about them. The typing rule for recursion given by Mendler ensures that all recursive calls are made on structurally smaller arguments. In this thesis we will use Mendler's approach to enrich our languages with

the power of recursive types while ensuring termination.

2.1.2 Chapter overview

In this chapter, we will study a simply typed lambda calculus extended with recursive types and Mendler-style recursion. After specifying the language, the main technical focus will be on proving termination of evaluation of well-typed programs. Our goal is to introduce some fundamental concepts and notation that we will use for our later development of Tores in Chapter 3.

The particular concepts we emphasize are:

- Recursive types with Mendler-style recursion
- Big-step operational semantics using environments
- Logical predicate semantics of types as sets of values
- Proof of termination using fixed point reasoning

Our termination proof uses the standard technique of *logical relations*, which goes back to Tait [1967] and Girard [1972]. In particular, we use a unary logical relation or *logical predicate* which interprets each type in our language as the set of values that should inhabit that type. The interesting case is the interpretation of recursive types, which relies on the notion of pre-fixed points in the lattice of sets. Our definition of the least pre-fixed point is particularly amenable to reasoning about Mendler recursion, and to our knowledge is a novel formulation. This proof framework will extend smoothly to the termination of Tores, where we will generalize our semantics to interpret richer types and type constructors as functions over sets of values.

To complement our proof on paper, we have also formalized it in the proof assistant Coq. This mechanization[1] serves a dual purpose. First, it verifies that our proof is correct with a machine-checked certificate. Second, it explains the transformation between such a proof written on paper and one written as a checked program in Coq. This can be a guide to others wishing to mechanize proofs about language metatheory, and also exposes areas for improvement in current proof environments.

The rest of the chapter is organized as follows. In Section 2.2, we will specify our language of study in terms of its syntax, type system and operational semantics (evaluation rules), giving examples of recursive types and Mendler recursion along the way. In Section 2.3, we describe the logical predicate semantics for our language and use it to state and prove termination. Finally in Section 2.4, we share the intricacies involved in implementing our proof in Coq, particularly in modelling substitutions, explaining the solutions we used.

[1] Available at `http://cs.mcgill.ca/~rjacob18/rectypes.v`.

2.2 Language of Study

2.2.1 Syntax

Our language of study is a simply typed lambda calculus with recursive types and Mendler-style recursion. In order to build algebraic data types using recursive types, we include the unit type and product and sum types. The introduction forms are $\langle\rangle$ for unit, $\langle t_1, t_2\rangle$ for products, and $\mathtt{in}_1\, t$ and $\mathtt{in}_2\, t$ for injections into sums. The corresponding elimination forms are $\mathtt{fst}\, t$ and $\mathtt{snd}\, t$ which project from a pair, and the case statement $\mathtt{case}\, t\, \mathtt{of}\, \mathtt{in}_1\ x_1 \mapsto s_1 \mid \mathtt{in}_2\ x_2 \mapsto s_2$ which selects a branch by on analyzing t. As usual, a function type is introduced by a lambda term $\lambda\, x.\, t$ and eliminated by application $t\, s$.

$$
\begin{array}{lll}
\text{Types} & T ::= & 1 \mid T_1 \times T_2 \mid T_1 + T_2 \mid T_1 \to T_2 \mid X \mid \mu X.\, T \\
\text{Terms} & t, s ::= & \langle\rangle \mid x \mid \lambda\, x.\, t \mid \mathtt{rec}\, f.\, t \mid t\, s \mid \\
& & \langle t_1, t_2\rangle \mid \mathtt{fst}\, t \mid \mathtt{snd}\, t \mid \mathtt{in}_1\, t \mid \mathtt{in}_2\, t \mid \\
& & (\mathtt{case}\, t\, \mathtt{of}\, \mathtt{in}_1\ x_1 \mapsto s_1 \mid \mathtt{in}_2\ x_2 \mapsto s_2)
\end{array}
$$

The interesting part of our language pertains to recursive types. A recursive type $\mu X.\, T$ introduces a type variable X which may appear in the body of the type T. For example, we can define natural numbers and lists over a fixed type A as follows:

$$
\begin{array}{lcl}
\mathsf{Nat} & = & \mu X.\, 1 + X \\
\mathsf{List}\ A & = & \mu X.\, 1 + A \times X
\end{array}
$$

Each of these types can describe arbitrarily large data by its *unfolding*: that is, the substitution of the entire recursive type for the type variable. In this chapter, we take an *equirecursive* approach; this means there is no explicit introduction form for recursive types. From this view, $\mathtt{in}_1\, \langle\rangle$ may represent the number 0 of type Nat or it may represent the empty list of type $\mathsf{List}\ A$. There is no tag around the term to indicate the *folding* from a sum type into a recursive type. The natural number 2 is simply written $\mathtt{in}_2\, (\mathtt{in}_2\, (\mathtt{in}_1\, \langle\rangle))$. This approach will lighten the syntax in this chapter; in Chapter 3 we will see the *isorecursive* approach applied to both recursive and stratified types.

Crucially, we omit the usual elimination form which unfolds recursive types: this is a source of nontermination when there is no restriction on the syntax of recursive types. Instead we can eliminate recursive types using a Mendler-style recursion term $\mathtt{rec}\, f.\, t$. This construct is used to write recursive functions that analyze data of recursive types, making recursive calls to f on strictly smaller data. The condition on recursive calls is enforced by the type system, which exploits the syntactic structure of recursive types.

2.2.2 Typing

We define typing of terms using two separate contexts: Γ for typing assumptions about term-level variables, and Δ to keep track of type variables that are in scope.

$$\begin{array}{lrcl} \text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : T \\ \text{Type Variable Contexts} & \Delta & ::= & \cdot \mid \Delta, X \end{array}$$

$\boxed{\Delta; \Gamma \vdash t : T}$ Term t has type T in the context Γ with type variables from Δ.

$$\frac{x : T \in \Gamma}{\Delta; \Gamma \vdash x : T}\ \texttt{t-var} \qquad \frac{\Delta; \Gamma, x{:}S \vdash t : T}{\Delta; \Gamma \vdash \lambda\, x.\, t : S \to T}\ \texttt{t-lam} \qquad \frac{\Delta; \Gamma \vdash t : S \to T \quad \Delta; \Gamma \vdash s : S}{\Delta; \Gamma \vdash t\, s : T}\ \texttt{t-app}$$

$$\frac{}{\Delta; \Gamma \vdash \langle\rangle : 1}\ \texttt{t-unit} \qquad \frac{\Delta; \Gamma \vdash t_1 : T_1 \quad \Delta; \Gamma \vdash t_2 : T_2}{\Delta; \Gamma \vdash \langle t_1, t_2 \rangle : T_1 \times T_2}\ \texttt{t-pair} \qquad \frac{\Delta; \Gamma \vdash t : T_1 \times T_2}{\Delta; \Gamma \vdash \texttt{fst}\, t : T_1}\ \texttt{t-fst} \qquad \frac{\Delta; \Gamma \vdash t : T_1 \times T_2}{\Delta; \Gamma \vdash \texttt{snd}\, t : T_2}\ \texttt{t-snd}$$

$$\frac{\Delta; \Gamma \vdash t : T_i}{\Delta; \Gamma \vdash \texttt{in}_i\, t : T_1 + T_2}\ \texttt{t-in}_i \qquad \frac{\Delta; \Gamma \vdash t : T_1 + T_2 \quad \Delta; \Gamma, x_1{:}T_1 \vdash s_1 : T \quad \Delta; \Gamma, x_2{:}T_2 \vdash s_2 : T}{\Delta; \Gamma \vdash (\texttt{case}\, t\, \texttt{of}\, \texttt{in}_1\ x_1 \mapsto s_1 \mid \texttt{in}_2\ x_2 \mapsto s_2) : T}\ \texttt{t-case}$$

$$\frac{\Delta; \Gamma \vdash t : T[\mu X.\, T / X]}{\Delta; \Gamma \vdash t : \mu X.\, T}\ \texttt{t-fold} \qquad \frac{\Delta, X; \Gamma, f : X \to T \vdash t : S \to T \quad X \notin \Delta}{\Delta; \Gamma \vdash \texttt{rec}\, f.\, t : \mu X.\, S \to T}\ \texttt{t-rec}$$

Figure 2.1: Typing rules for lambda calculus with Mendler-style recursion

The typing rules for lambdas, applications, pairs, projections, injections and case expressions are standard. Note that lambda and case expressions bind variables (x, x_1 and x_2) which we assume to be fresh, i.e. not appearing in the typing context Γ. The `t-fold` rule follows the equirecursive style, where a term checks against a recursive type if it checks against its unfolding. Note that we do not enforce type uniqueness in this system, since for example $\texttt{in}_1\, \langle\rangle$ can be interpreted both as zero and the empty list.

The Mendler-style typing rule `t-rec` specifies recursive functions taking inputs of a recursive type $\mu X.\, S$ to produce outputs of type T. The term $\texttt{rec}\, f.\, t$ binds a function variable f which is assumed to be fresh to Γ. The trick to the rule is that, in checking the body of the function t, the typing assumption says that f works only on inputs of the type variable X. Since X is a fresh type variable bound by the recursive type $\mu X.\, S$, the only possible data of type X is that obtained from unwrapping the input data by exactly one layer. For example, when recursing over an input of type Nat, the only recursive call allowed is on the predecessor. This strict condition ensures that recursive functions must terminate.

To illustrate, consider the program that computes the length of a list, which has type $\mathsf{List} \to \mathsf{Nat}$.

$$\begin{array}{l} \texttt{rec}\, len.\, \lambda\, l.\, \texttt{case}\, l\, \texttt{of} \\ \quad \mid \texttt{in}_1\, x_1 \mapsto \texttt{in}_1\, \langle\rangle \\ \quad \mid \texttt{in}_2\, x_2 \mapsto \texttt{in}_2\, (len\, (\texttt{snd}\, x_2)) \end{array}$$

The program builds a recursive function *len* that traverses the input list l to reconstruct its length. Note that the $\texttt{in}_i$ syntax appears in two distinct settings: in the syntax for case analysis and as injections to construct natural numbers. For natural numbers, $\texttt{in}_1\, \langle\rangle$ represents 0 and $\texttt{in}_2$ acts as the successor function. In the first branch of the case expression, l is empty so we return 0. In the case that l is not empty, $\texttt{snd}\, x_2$

refers to the tail of l ($\texttt{fst}\, x_2$ would be the head). The call to *len* gives the length of the tail which we then increment with $\texttt{in}_2$.

Why is the recursive call to *len* allowed in this case? Let us see the typing assumptions gathered on the way to type checking the application of *len*:

$$len : X \to \mathsf{Nat},\ l : 1 + A \times X,\ x_1 : 1,\ x_2 : A \times X.$$

We can see that the tail ($\texttt{snd}\, x_2$) has type X, making it a valid argument to *len*. In fact it is the *only* thing of type X, so applying *len* is the only possible use for both *len* and the tail $\texttt{snd}\, x_2$.

This typing discipline ensures termination, but it is also severely restrictive: consider the dual consequence of the previous sentence. First, there is no way to make recursive calls to data that is smaller by any other metric. For example, we could not write a sort function that makes recursive calls to both halves of the input list. Second, we cannot use structurally smaller data outside of a recursive call. This means for example that we cannot write a constant-time predecessor function for natural numbers!

The situation is not so bad, however. Recall that the primary motivation for our language is for use in a proof system. The (weak) induction principles gained from Mendler recursion are sufficient for many inductive proofs (consider proof by induction on a derivation tree, for example). Also, though we do not explore them here, Mendler-style recursion has a number of variants and generalizations which permit much more sophisticated terminating programs (see Abel [2010] for an approachable survey). Our goal here is to introduce the reader to the tool of Mendler recursion and techniques to formalize its metatheory, which should generalize to more sophisticated systems.

2.2.3 Evaluation

We describe evaluation of our language using a big-step operational semantics with environments to track substitutions of values for variables. Environments have two main advantages over the traditional eager substitutions. First, they model a practical implementation of evaluation, where variables can be looked up efficiently in a map data structure. Second, they integrate smoothly into our metatheory: we avoid having to prove lemmas about term-level substitution (which is especially convenient in our mechanization).

Let us begin with the syntax of values and environments.

$$\begin{array}{lrcl} \text{Closures} & c & ::= & (\lambda\, x.\, t)[\rho] \mid (\texttt{rec}\, f.\, t)[\rho] \\ \text{Values} & v & ::= & \langle\rangle \mid \langle v_1, v_2\rangle \mid \texttt{in}_1\, v \mid \texttt{in}_2\, v \mid c \\ \text{Environments} & \rho & ::= & \cdot \mid \rho, v/x \end{array}$$

Values arise from introduction forms (unit, pairs and injections) and from closures. A closure c is formed from a function, $\lambda\, x.\, t$ or $\texttt{rec}\, f.\, t$, paired with an environment ρ. Environments are simply substitutions providing values for free variables. They appear not only in closures but also in the main evaluation judgment.

The evaluation relation is defined using two mutually defined judgments. The first judgment, written $t[\rho] \Downarrow v$, says that a term t evaluates to a value v under an environment ρ. The second judgment, written $c \cdot v \Downarrow w$, says that the application of a closure c to a value v evaluates to another value w. This second

judgment allows us to separate the application of terms from the application of values in our evaluation rules, which will be useful in defining the semantic function space.

$\boxed{t[\rho] \Downarrow v}$ Term t under environment ρ evaluates to v.

$$\frac{}{\langle\rangle[\rho] \Downarrow \langle\rangle}\ \texttt{e-unit} \qquad \frac{t_1[\rho] \Downarrow v_2 \quad t_2[\rho] \Downarrow v_2}{\langle t_1, t_2\rangle[\rho] \Downarrow \langle v_1, v_2\rangle}\ \texttt{e-pair} \qquad \frac{t[\rho] \Downarrow \langle v_1, v_2\rangle}{(\texttt{fst}\, t)[\rho] \Downarrow v_1}\ \texttt{e-fst} \qquad \frac{t[\rho] \Downarrow \langle v_1, v_2\rangle}{(\texttt{snd}\, t)[\rho] \Downarrow v_2}\ \texttt{e-snd}$$

$$\frac{v/x \in \rho}{x[\rho] \Downarrow v}\ \texttt{e-var} \qquad \frac{t[\rho] \Downarrow v}{(\texttt{in}_i\, t)[\rho] \Downarrow \texttt{in}_i\, v}\ \texttt{e-in}_i \qquad \frac{t[\rho] \Downarrow \texttt{in}_i\, v \quad s_i[\rho, v/x_i] \Downarrow v}{(\texttt{case}\, t\, \texttt{of}\, \texttt{in}_1\ x_1 \mapsto s_1 \mid \texttt{in}_2\ x_2 \mapsto s_2)[\rho] \Downarrow v}\ \texttt{e-case-in}_i$$

$$\frac{}{(\lambda\, x.\, t)[\rho] \Downarrow (\lambda\, x.\, t)[\rho]}\ \texttt{e-lam} \qquad \frac{}{(\texttt{rec}\, f.\, t)[\rho] \Downarrow (\texttt{rec}\, f.\, t)[\rho]}\ \texttt{e-rec} \qquad \frac{t[\rho] \Downarrow c \quad s[\rho] \Downarrow v \quad c \cdot v \Downarrow w}{(t\, s)[\rho] \Downarrow w}\ \texttt{e-app}$$

$\boxed{c \cdot v \Downarrow w}$ Closure c applied to value v evaluates to w.

$$\frac{t[\rho, v/x] \Downarrow w}{(\lambda\, x.\, t)[\rho] \cdot v \Downarrow w}\ \texttt{e-app-lam} \qquad \frac{t[\rho, (\texttt{rec}\, f.\, t)[\rho]/f] \Downarrow c \quad c \cdot v \Downarrow w}{(\texttt{rec}\, f.\, t)[\rho] \cdot v \Downarrow w}\ \texttt{e-app-rec}$$

Figure 2.2: Operational semantics for lambda calculus with Mendler-style recursion

The evaluation rules are given in Fig. 2.2. The rules for unit, pairs, projections and injections are straightforward. To evaluate a variable, we look up its value in the environment. To evaluate a case expression, we evaluate the scrutinized term t and, picking the appropriate branch, evaluate the subterm s_i under an extended environment. A function g, which is either $\lambda\, x.\, t$ or $\texttt{rec}\, f.\, t$, evaluates under ρ to the closure $(g)[\rho]$. A term application $t\, s$ evaluates by evaluating the subterms and invoking the value application judgment. Value application is evaluated according to the closure. For a lambda $\lambda\, x.\, t$, we evaluate the body t under the environment extended with the argument value v. For recursion $\texttt{rec}\, f.\, t$, we evaluate the unravelled body t (with a substitution for the recursive variable f) and invoke the value application judgment once more. Note that the operational semantics for recursion are equivalent to a standard presentation; it is the type system alone that ensures termination.

2.3 Termination Proof

2.3.1 Logical Predicate Semantics

The main idea of the proof, following the logical predicate method, is to construct a semantic model of our language. In this case we interpret each type as a set of values. Our main result is that given a term t of type T, the result of evaluating t is a value in the semantic interpretation of T. As the proof is by induction on the typing derivation, this statement of the theorem will provide a stronger induction hypothesis to use in each case of the proof.

As usual, the theorem statement also requires generalization to open terms, i.e. terms with free variables

from a typing context. This involves a semantic interpretation of typing contexts as environments, i.e. substitutions of variables with well-typed values. In addition, our types may contain free type variables which arise from recursive types. Hence we require a semantic interpretation of type variable contexts as substitutions with type interpretations. We start by describing such type variable environments.

Let Ω be the set of all values allowed by our syntax. A *type variable environment* is a function mapping each type variable to a subset of values (possibly the empty set). The function $\mathcal{D}$ gives the set of type variable environments associated with a given type variable context.

$$\begin{array}{lcl} \mathcal{D}[\cdot] & = & \{\cdot\} \\ \mathcal{D}[\Delta, X] & = & \{\eta, \mathcal{C}/X \mid \eta \in \mathcal{D}[\Delta], \mathcal{C} \in \mathcal{P}(\Omega)\}. \end{array}$$

The definition says that a type variable environment conforms to a type variable context as long as it does not refer to type variables outside that context.

The interpretation of types $\mathcal{V}$ is more interesting, describing the set of values of a type under a type variable environment. For that we define some semantic analogues of our syntactic type formers. For $\mathcal{A}, \mathcal{B} \in \mathcal{P}(\Omega)$, we define the following sets.

$$\begin{array}{l} \mathcal{A} \times \mathcal{B} = \{\langle v_1, v_2 \rangle \mid v_1 \in \mathcal{A}, v_2 \in \mathcal{B}\} \\ \mathcal{A} + \mathcal{B} = \{\mathtt{in}_1\, v \mid v \in \mathcal{A}\} \cup \{\mathtt{in}_2\, v \mid v \in \mathcal{B}\} \\ \mathcal{A} \to \mathcal{B} = \{c \in \Omega \mid \forall v \in \mathcal{A}.\ \exists w \in \mathcal{B}.\ c \cdot v \Downarrow w\} \end{array}$$

The first two sets are set analogues of the product and sum type constructors respectively, and we overload the connectives to reflect this. The last set is a semantic analogue of the function type, containing closures which, when applied to any value in the first set, evaluate to some value in the second set. This is where the value application judgment comes in handy: it allows us to talk about function application without mentioning general terms.

With this notation we define the semantic interpretation $\mathcal{V}$, taking a type T and a type variable environment η to give a subset of Ω.

$$\begin{array}{lcl} \mathcal{V}[1](\eta) & = & \{\langle\rangle\} \\ \mathcal{V}[T \times S](\eta) & = & \mathcal{V}[T](\eta) \times \mathcal{V}[S](\eta) \\ \mathcal{V}[T + S](\eta) & = & \mathcal{V}[T](\eta) + \mathcal{V}[S](\eta) \\ \mathcal{V}[T \to S](\eta) & = & \mathcal{V}[T](\eta) \to \mathcal{V}[S](\eta) \\ \mathcal{V}[X](\eta) & = & \eta(X) \\ \mathcal{V}[\mu X.\, F](\eta) & = & \boldsymbol{\mu}(\mathcal{X} \mapsto \mathcal{V}[F](\eta, \mathcal{X}/X)) \end{array}$$

where for any $\mathcal{F} : \mathcal{P}(\Omega) \to \mathcal{P}(\Omega)$,

$$\boldsymbol{\mu}\mathcal{F} = \bigcap \{\mathcal{C} \subseteq \Omega \mid \forall \mathcal{X}.\ \mathcal{X} \subseteq \mathcal{C} \implies \mathcal{F}(\mathcal{X}) \subseteq \mathcal{C}\}.$$

We first explain the simple cases of the interpretation. The unit case is clear, since $\langle\rangle$ is the only possible value of type 1. The product, sum and function type cases utilize our semantic type formers

with interpretations of the component types. The case for a type variable simply invokes the type variable environment to look up the set of values associated with that type variable.

To interpret a recursive type $\mu X.\, F$, we take the fixed point of the meta-level function mapping a set of values $\mathcal{X}$ to the interpretation of the body F, where the type variable environment is extended by $\mathcal{X}$. Here $\boldsymbol{\mu}$ is a semantic fixed point operator, mirroring the syntactic μ construct. It transforms a meta-level function $\mathcal{F}$ on sets of values to the least set satisfying a *pre-fixed point* property, that subsets $\mathcal{X}$ of $\mathcal{C}$ remain subsets under $\mathcal{F}$. The minimality of $\boldsymbol{\mu}\mathcal{F}$ is achieved via a set intersection over all pre-fixed points.

The definition of $\boldsymbol{\mu}$ is novel due to our pre-fixed point condition on each set $\mathcal{C}$ in the intersection. This condition is stronger that the standard one that $\mathcal{F}(\mathcal{C}) \subseteq \mathcal{C}$. (Taking $\mathcal{X}$ to be $\mathcal{C}$ in particular gives the latter condition.) However, the standard pre-fixed point requires $\mathcal{F}$ to be monotone, which is not necessarily the case in our treatment of Mendler recursion, as we do not require recursive types to be positive. We discuss an alternative encoding of fixed points in Section 4.1.

The final piece of machinery we need is a semantic interpretation $\mathcal{G}$ of typing contexts, which describes the value substitutions (i.e. environments) that are well-typed under a given context. $\mathcal{G}$ takes a context Γ and a type variable environment η to give the set of environments that match the context. That is, if $\rho \in \mathcal{G}[\Gamma](\eta)$, then ρ maps each variable x to a value v in the interpretation of the type of x in Γ. This is expressed in the following definition.

$$\begin{array}{lcl} \mathcal{G}[\cdot](\eta) & = & \{\cdot\} \\ \mathcal{G}[\Gamma, x : T](\eta) & = & \{\rho, v/x \mid \rho \in \mathcal{G}[\Gamma](\eta), v \in \mathcal{V}[T](\eta)\}. \end{array}$$

2.3.2 Theorem Statement

The main theorem is stated using the semantic machinery we built in the previous section. It is required to give us a sufficiently strong induction hypothesis. The assumptions in the theorem are that we have a well-typed term t under contexts Δ and Γ, and type variable and value environments that conform to Δ and Γ respectively. The conclusion is that t evaluates to some value v in the semantic interpretation of the type.

Theorem 1. *If* $\Delta; \Gamma \vdash t : T$, $\eta \in \mathcal{D}[\Delta]$ *and* $\rho \in \mathcal{G}[\Gamma](\eta)$ *then* $t[\rho] \Downarrow v$ *for some* $v \in \mathcal{V}[T](\eta)$.

A simple statement of termination for closed terms can be recovered by considering the case in which Δ and Γ are empty. The empty type variable and value environments satisfy the latter assumptions, giving the following corollary. We assume a natural syntax for our judgments with empty contexts and environment.

Corollary 2. *If* $\vdash t : T$ *then* $t \Downarrow v$ *for some value* v.

The proof of the main theorem is by induction on the typing derivation $\mathcal{D}$. This gives a case for each typing rule, which we explain in the remainder of this section.

2.3.3 Simple Cases

To give a sense of the proof structure, we sketch some of the simple cases here. As a warm-up, suppose the root of the typing derivation is `t-unit`. In this case $t = \langle\rangle$ and $T = 1$. By `e-unit`, $\langle\rangle[\rho] \Downarrow \langle\rangle$ and we observe that $\langle\rangle$ is the sole member of $V[1](\eta)$. This completes the unit case.

For the variable case, assume the root of the typing derivation is `t-var`, so $t = x$ and $x{:}T \in \Gamma$. We need to show that $\exists v \in \mathcal{V}[T](\eta)$ such that $v/x \in \rho$. This is proved by induction on the structure of Γ, though we omit the proof here. Using this fact and `e-var`, we obtain that $x[\rho] \Downarrow v \in \mathcal{V}[T](\eta)$, completing the variable case.

The last simple case we show is the lambda abstraction case. Suppose the root of the typing derivation is `t-lam`, so $t = \lambda\, x.\, s$ and $T = R \to S$. We claim that $v = (\lambda\, x.\, s)[\rho] \in \mathcal{V}[R \to S](\eta)$, since we know that $(\lambda\, x.\, s)[\rho] \Downarrow (\lambda\, x.\, s)[\rho]$ by `e-lam`. Unfolding the definition of $\mathcal{V}[R \to S](\eta)$, it remains to show that $\forall u \in \mathcal{V}[R](\eta)$, $v \cdot u \Downarrow w$ for some $w \in \mathcal{V}[S](\eta)$. Assume $u \in \mathcal{V}[R](\eta)$. Then $\rho, u/x \in \mathcal{G}[\Gamma, x{:}R](\eta)$, and $\Delta; \Gamma, x{:}R \vdash s : S$ by `t-lam`. Using the induction hypothesis, we learn that $\exists w \in \mathcal{V}[S](\eta)$ such that $s[\rho, u/x] \Downarrow w$. Finally by `e-app-lam` we get that $(\lambda\, x.\, s)[\rho] \cdot u \Downarrow w$, which is what we needed.

We omit the cases for `t-app`, `t-pair`, `t-fst`, `t-snd`, `t-inl`, `t-inr` and `t-case`. They follow the structure we have shown above and do not provide any novelty. The cases of interest in this paper are the `t-fold` and `t-rec` cases, which involve recursive types.

2.3.4 Fold Case

To prove the `t-fold` case, we will first need a lemma regarding type substitutions. This is because recursive types are unrolled using type substitution. The lemma says that the semantic interpretation of a type under a substitution is equal to the type interpretation under the appropriately extended type variable environment.

Lemma 3 (Semantics of type substitution)**.** *For types T and S and a type variable environment η,*

$$\mathcal{V}[T[S/X]](\eta) = \mathcal{V}[T](\eta, \mathcal{V}[S](\eta)/X).$$

The proof of this lemma is by induction on the structure of T, using the definition of type substitutions. Though not complicated, we will see later that this proof is not so straight-forward in the mechanization. There, we will require a whole framework of substitution mechanisms and a suite of more general lemmas. We shall omit the proof here and revisit it when we describe the mechanization.

The other lemma we need for this case is a semantic one, showing that the semantic fixed point operator $\boldsymbol{\mu}$ is really a (pre-)fixed point.

Lemma 4 (Pre-fixed point)**.** *Let $\mathcal{F} : \mathcal{P}(\Omega) \to \mathcal{P}(\Omega)$ and $\boldsymbol{\mu}$ as defined in our semantics. Then $\mathcal{F}(\boldsymbol{\mu}\mathcal{F}) \subseteq \boldsymbol{\mu}\mathcal{F}$.*

Proof. As the set on the right-hand side is an intersection, we can show inclusion by showing inclusion in each set in the intersection. So let $\mathcal{C}$ be an arbitrary set in the intersection, that is a subset of Ω satisfying the property that $\forall \mathcal{X}.\ \mathcal{X} \subseteq \mathcal{C} \implies \mathcal{F}(\mathcal{X}) \subseteq \mathcal{C}$. We need to show that $\mathcal{F}(\boldsymbol{\mu}\mathcal{F}) \subseteq \mathcal{C}$. From the assumed

property of $\mathcal{C}$, it suffices to show that $\boldsymbol{\mu}\mathcal{F} \subseteq \mathcal{C}$. We now have a set inclusion with an intersection on the left. To prove it, we need just one set in the intersection that is a subset of $\mathcal{C}$. That is, we need a $\mathcal{D} \subseteq \mathcal{C}$ such that $\forall \mathcal{X}.\ \mathcal{X} \subseteq \mathcal{D} \implies \mathcal{F}(\mathcal{X}) \subseteq \mathcal{D}$. However, $\mathcal{C}$ itself satisfies this property from our original assumption. □

To prove the `t-fold` case, we start by applying the induction hypothesis to the typing subderivation. This says that there is a $v \in \mathcal{V}[F[\mu X.\, F/X]](\eta)$ such that $t[\rho] \Downarrow v$. Define $\mathcal{F} : \mathcal{P}(\Omega) \to \mathcal{P}(\Omega)$ by $\mathcal{F}(\mathcal{X}) = \mathcal{V}[F](\eta, \mathcal{X}/X)$, so that $\mathcal{V}[\mu X.\, F](\eta) = \boldsymbol{\mu}\mathcal{F}$ by definition. To finish the case we need to only show that $v \in \mathcal{V}[\mu X.\, F](\eta) = \boldsymbol{\mu}\mathcal{F}$. By the type substitution lemma, we have $\mathcal{V}[F[\mu X.\, F/X]](\eta) = \mathcal{V}[F](\eta, \mathcal{V}[\mu X.\, F](\eta)/X)$. Rewriting the right-hand set further using $\mathcal{F}$ gives $\mathcal{V}[F](\eta, \boldsymbol{\mu}\mathcal{F}/X)$, and using the definition once more yields $\mathcal{F}(\boldsymbol{\mu}\mathcal{F})$. However, by the previous lemma we know this is a subset of $\boldsymbol{\mu}\mathcal{F}$. Hence $v \in \boldsymbol{\mu}\mathcal{F}$, completing the proof of this case.

2.3.5 Recursion Case

The only remaining case is the `t-rec` case. To prove it we will need another semantic result regarding the semantic fixed point $\boldsymbol{\mu}$. It gives a sufficient condition for membership in the semantic function space from a fixed point.

Lemma 5 (Function space from a pre-fixed point)**.** *Suppose $v \in \Omega$, $\mathcal{C} \subseteq \Omega$ and $\mathcal{F} : \mathcal{P}(\Omega) \to \mathcal{P}(\Omega)$.*
If $\forall \mathcal{X} \subseteq \Omega.\ v \in \mathcal{X} \to \mathcal{C} \implies v \in \mathcal{F}(\mathcal{X}) \to \mathcal{C}$, then $v \in \boldsymbol{\mu}\mathcal{F} \to \mathcal{C}$.

Proof. Let us define some useful notation. For $v \in \Omega$ and $\mathcal{B} \subseteq \Omega$, let $\mathcal{E}_v(\mathcal{B}) = \{u \in \Omega \mid \exists w \in \mathcal{B}\ v \cdot u \Downarrow w\}$. Note that this definition is closely related to that of a semantic function space, as for $\mathcal{A} \subseteq \Omega$, $v \in \mathcal{A} \to \mathcal{B} \iff \mathcal{A} \subseteq \mathcal{E}_v(\mathcal{B})$. Restating the lemma using our $\mathcal{E}$ notation, it says that if $\forall \mathcal{X} \subseteq \Omega.\ \mathcal{X} \subseteq \mathcal{E}_v(\mathcal{C}) \implies \mathcal{F}(\mathcal{X}) \subseteq \mathcal{E}_v(\mathcal{C})$ then $\boldsymbol{\mu}\mathcal{F} \subseteq \mathcal{E}_v(\mathcal{C})$.

Assume the premise is true. Unfolding the definition of $\boldsymbol{\mu}\mathcal{F}$, we need to prove that

$$\bigcap \{\mathcal{D} \subseteq \Omega \mid \forall \mathcal{X}.\ \mathcal{X} \subseteq \mathcal{D} \implies \mathcal{F}(\mathcal{X}) \subseteq \mathcal{D}\} \subseteq \mathcal{E}_v(\mathcal{C}).$$

As in the last lemma, it suffices to show that just one $\mathcal{D}$ in the intersection is included in $\mathcal{E}_v(\mathcal{C})$. However, $\mathcal{E}_v(\mathcal{C})$ itself is in the intersection because of the assumption. Moreover it is a subset of itself so we are done. □

The final lemma we need is a form of backward closure for recursive terms. It says that if the unrolling of a recursive term evaluates to a function value, then the recursive term itself is a function value.

Lemma 6 (Backward closure)**.** *Let t be a term, ρ an environment and $\mathcal{A}, \mathcal{B} \subseteq \Omega$.*
If $t[\rho, (\texttt{rec}\, f.\, t)[\rho]/f] \Downarrow v$ for some $v \in \mathcal{A} \to \mathcal{B}$, then $(\texttt{rec}\, f.\, t)[\rho] \in \mathcal{A} \to \mathcal{B}$.

Proof. Suppose $u \in \mathcal{A}$. Since $v \in \mathcal{A} \to \mathcal{B}$, by definition there exists a $w \in \mathcal{B}$ such that $v \cdot u \Downarrow w$. Since also $t[\rho, (\texttt{rec}\, f.\, t)[\rho]/f] \Downarrow v$, `e-app-rec` implies that $(\texttt{rec}\, f.\, t)[\rho] \cdot u \Downarrow w$. But u is arbitrary, so the definition of the semantic function space gives us $(\texttt{rec}\, f.\, t)[\rho] \in \mathcal{A} \to \mathcal{B}$. □

We can finally address the `t-rec` case. By `e-rec` we know $(\texttt{rec}\, f.\, s)[\rho] \Downarrow (\texttt{rec}\, f.\, s)[\rho]$, so we need only show that

$$(\texttt{rec}\, f.\, s)[\rho] \in \mathcal{V}[\mu X.\, F \to R](\eta).$$

Simplifying the right-hand set using the semantics for function types and recursive types, this is equivalent to showing

$$(\texttt{rec}\, f.\, s)[\rho] \in \boldsymbol{\mu}(\mathcal{X} \mapsto \mathcal{V}[F](\eta, \mathcal{X}/X)) \to \mathcal{V}[R](\eta).$$

By Lemma 5, it suffices to show that, for all $\mathcal{X} \subseteq \Omega$,

$$(\texttt{rec}\, f.\, s)[\rho] \in \mathcal{X} \to \mathcal{V}[R](\eta) \implies (\texttt{rec}\, f.\, s)[\rho] \in \mathcal{V}[F](\eta, \mathcal{X}/X) \to \mathcal{V}[R](\eta).$$

So suppose $\mathcal{X} \subseteq \Omega$ such that $(\texttt{rec}\, f.\, s)[\rho] \in \mathcal{X} \to \mathcal{V}[R](\eta)$. By backward closure (Lemma 6), it suffices to show that

$$\exists v \in \mathcal{V}[F](\eta, \mathcal{X}/X) \to \mathcal{V}[R](\eta) \text{ such that } s[\rho, (\texttt{rec}\, f.\, s)[\rho]/f] \Downarrow v.$$

Now, the induction hypothesis says that

$$\forall \eta' \in \mathcal{D}[\Delta, X].\ \forall \rho' \in \mathcal{G}[\Gamma, f : X \to R](\eta').\ \exists v \in \mathcal{V}[F \to R](\eta').\ s[\rho'] \Downarrow v.$$

We have $\eta, \mathcal{X}/X \in \mathcal{D}[\Delta, X]$ since $\eta \in \mathcal{D}[\Delta]$ and $\mathcal{X} \subseteq \Omega$. Also, $\rho, (\texttt{rec}\, f.\, s)[\rho]/f \in \mathcal{G}[\Gamma, f : X \to R](\eta')$ because $\rho \in \mathcal{G}[\Gamma](\eta)$ implies $\rho \in \mathcal{G}[\Gamma](\eta')$ and

$$(\texttt{rec}\, f.\, s)[\rho] \in \mathcal{V}[X \to R](\eta') = \mathcal{X} \to \mathcal{V}[R](\eta).$$

Thus instantiating $\eta' = \eta, \mathcal{X}/X$ and $\rho' = \rho, (\texttt{rec}\, f.\, s)[\rho]/f$ in the induction hypothesis means there is a

$$v \in \mathcal{V}[F \to R](\eta') = \mathcal{V}[F](\eta, \mathcal{X}/X) \to \mathcal{V}[R](\eta)$$

such that $s[\rho'] = s[\rho, (\texttt{rec}\, f.\, s)[\rho]/f] \Downarrow v$. But this is exactly what we were required to show! Hence we have completed the `t-rec` case and the entire proof.

2.4 Mechanization

2.4.1 Overview

We have mechanized the above termination proof using the proof assistant Coq. Coq consists of a powerful type theory representing proofs and propositions (via the Curry-Howard correspondence) combined with a convenient suite of "tactics" through which the user can develop proofs interactively. It is this combination of expressive power and usability that motivated our choice to use Coq. Technically, we require Coq's support for *higher-kinded* (impredicative) polymorphism to encode our pre-fixed point definition, as we explain in Section 2.4.5. This feature is missing from many other proof assistants such as Beluga (which does not yet support polymorphism) and Agda (which uses a predicative type theory).

The challenges of our mechanization are similar to others documented mechanizations, most notably the normalization proofs for System F by Berardi [1990] and Altenkirch [1993]. As is usual with mechanization of language metatheory, we devote significant effort towards reasoning about substitutions, which in our case arise from variable binding in recursive types. We take a fairly direct approach to this problem, representing variables by de Bruijn indices and using *simultaneous* or *parallel* substitutions and renamings. Apart from this issue, most of our proof translates smoothly into the appropriate encodings in Coq. This speaks to the elegance of the logical relations method and our interpretation of recursive types. Our goal for the rest of the section is to highlight some intricacies of the implementation, particularly in modelling binding (Section 2.4.2) and substitution (Sections 2.4.3 and 2.4.4), and our set-theoretic semantics (Section 2.4.5).

2.4.2 Representation of Syntax

We represent the syntax of our language using Coq's inductive data types (recursive types with a positivity restriction). Similarly the typing and evaluation relations are represented as inductive types, which we reason about using the induction principles generated by Coq. In particular, we are able to perform induction on typing derivations, which is required for the main proof.

The only subtle aspect to encoding our object language is how we represent variable binding. Binding occurs in two places in our language: on the level of terms due to lambda expressions, and on the level of types due to recursive types. Binding and substitution on the term level is actually harmless in our case, because our evaluation relation explicitly builds parallel substitutions in the form of environments. We do not need to implement the usual term-level substitution operation. Hence the technique of de Bruijn indices is appropriate here: natural numbers are the right abstraction for looking up a variable by index in an environment or typing context.

Unfortunately, we do not have the same luxury for type-level binding and substitution. Since our typing rules treat type substitution as a meta-level operation, we must reason about it explicitly in our proof. Specifically we need to show that the interpretation of types associates with type-level substitution. The required lemmas are tedious but not difficult: much of the proof could certainly be automated by scripting in Coq's tactic language. In fact, we believe these lemmas could be both generated and proven automatically using the recent Autosubst library [Schäfer et al., 2015]. Nevertheless we explain our chosen approach for instructive purposes.

Note that there are a wide variety of approaches to modelling binding and substitution in metalanguages. We do not attempt to acknowledge them all here, but do give a sample of possible alternatives. Closest in appearance to an on-paper development are the use of *raw named terms*. These are clumsy to work with because they do not inherently respect α-equivalence ($\lambda\, x.\, x$ and $\lambda\, y.\, y$ are different a priori). Hence the equivalence must be defined and all relations on the syntax must be proven to respect it. Another simple but effective encoding is via de Bruijn indices, which we employ here. This names each variable in a term by the distance to its binder, inherently identifying α-equivalent terms. However, this introduces the need for *shifting* (incrementing all indices beyond a given bound) for certain operations. This representation is often considered too low-level, but for this mechanization we chose to accept the cost. A hybrid approach

which has proven effective in practice is the *locally nameless* representation [Charguéraud, 2011]. This uses de Bruijn indices to represent bound variables (identifying α-equivalent terms) and names for free variables (avoiding the need for shifting). This seems to attain a local optimum in this space, which we would like to try in the future.

Finally we mention the fundamentally different approach of handling binding and substitution by abstractions in the metalanguage. Beluga and Abella for example support encodings of syntax with binding using higher order abstract syntax (HOAS) and λ-tree syntax respectively. The idea here is to model binding and substitution directly using the corresponding notions in the metalanguage. Unfortunately, these languages do not support higher-kinded polymorphism, which we desire for encoding pre-fixed points.

2.4.3 Substitution Framework

We use the idea of parallel, or simultaneous, substitutions on types. This means that substitutions can replace not just a single type variable, but several at a time. This approach gives a general way to compose and reason about substitutions.

One issue we must deal with is showing that our definitions of type substitution and composition of substitutions are well-founded, i.e. terminating. These mutual definitions are not structurally recursive and using a naive approach to parallel substitutions makes it non-trivial to prove them terminating in Coq. We favour a treatment in which well-foundedness is checked automatically by Coq's termination checking.

To do this we introduce the auxiliary notion of a *renaming*, using the framework described by Adams [2006] and Benton et al. [2012]. A renaming is very similar to a substitution except that it only replaces variables with other variables, as opposed to replacing them with arbitrary types. With this notion we then define the composition of substitutions with renamings instead of with other substitutions. This relies on definitions of types under renamings as well as composition of renamings. Through this extra layer of indirection, our definitions will be evidently total according to Coq's termination checker. The cost of this approach, as in the previously referenced work, is a bloated set of lemmas to prove about the interaction between these definitions and our semantics.

Before we define this framework, let us restate the definition of the type language using the de Bruijn representation of type variables.

$$\text{Types} \quad T, S ::= 1 \mid T \to S \mid \operatorname{Var} k \mid \mu T \mid T \times S \mid T + S$$

Note that variables are represented by natural numbers, and that recursive types no longer bind a variable by name, but instead introduce a variable of index 0, while "shifting" the other variables up by one. To implement this in type checking, we require the notion of substitutions and renamings as mentioned.

Substitutions and renamings consist either of a shift, which adds to each of the indices representing free variables, or an extension with a type (variable). We overload the notation for a shift to use it in both syntactic categories.

$$\begin{array}{lrcl} \text{Substitutions} & \sigma & ::= & \uparrow^n \mid \sigma, T \\ \text{Renamings} & \pi & ::= & \uparrow^n \mid \pi, k \end{array}$$

Observe that the only difference between the two constructs is that substitutions are extended with types whereas renamings are extended with natural numbers representing type variables.

In order to define type substitution we first need some more primitive concepts, namely: composition of renamings, types under renamings, and substitutions composed with renamings. In fact, each of these operations is required to define the next one, resulting in a chain of dependencies.

Building from the bottom up, the first operation we need is that of composing renamings. The syntax $\pi_1[\pi_2]$ refers to the renaming obtained by composing the two renamings π_1 and π_2. The definition is by cases on the first renaming.

$$\begin{aligned}\uparrow^n [\pi] &= \mathrm{pop}\, n\, \pi \\ (\pi', k)[\pi] &= \pi'[\pi], \pi(k)\end{aligned}$$

We rely on an auxiliary "pop" function that drops indices from a renaming, or adds to the shift if the renaming is a shift. The other operation used is that of a lookup $\pi(k)$, which simply indexes into the list. We will assume identical pop and lookup functions for substitutions later.

Now we can define types under renamings, denoted $T[\pi]$, this time by cases on T.

$$\begin{aligned}1[\pi] &= 1 \\ (T \times S)[\pi] &= T[\pi] \times S[\pi] \\ (T + S)[\pi] &= T[\pi] + S[\pi] \\ (T \to S)[\pi] &= T[\pi] \to S[\pi] \\ (\mathrm{Var}\, k)[\pi] &= \mathrm{Var}(\pi(k)) \\ (\mu T)[\pi] &= \mu(T[\pi[\uparrow^1], 0])\end{aligned}$$

Most of the cases distribute the renaming π according to the structure of the type. In the variable case, we look up the renaming to get the correct de Bruijn index. In the last case, we carry π under the μ binder by introducing a new variable at index 0 and shifting the variables in π up by one.

With this definition we can define substitutions composed with renamings, in the form $\sigma[\pi]$.

$$\begin{aligned}\uparrow^n [\pi] &= \langle \mathrm{pop}\, n\, \pi \rangle \\ (\sigma', T)[\pi] &= \sigma'[\pi], T[\pi]\end{aligned}$$

Here we have denoted with angle brackets the conversion from a renaming to a substitution, which is straightforward (wrapping each index with the variable constructor Var). In the second case of an extended substitution, we rely on our previous operation of a type under a renaming.

At the top of our chain of definitions, we finally have type substitution.

$$
\begin{aligned}
1[\sigma] &= 1 \\
(T \times S)[\sigma] &= T[\sigma] \times S[\sigma] \\
(T + S)[\sigma] &= T[\sigma] + S[\sigma] \\
(T \to S)[\sigma] &= T[\sigma] \to S[\sigma] \\
(\mathrm{Var}\, k)[\sigma] &= \sigma(k) \\
(\mu T)[\sigma] &= \mu(T[\sigma[\uparrow^1], \mathrm{Var}\, 0])
\end{aligned}
$$

This is extremely similar to the definition of types under renamings. Again the first four cases are structural on the type, and the variable case simply looks up the appropriate entry of the substitution. The final case carries σ under the binder by shifting the free variables and adding a new type variable at index 0. This gives us all the notions we need to prove semantic properties of types under substitutions.

2.4.4 Semantics of Substitutions

To use the substitution framework we described, we need lemmas that allow us to reason about them. For that we need semantic interpretations of both substitutions and renamings. These interpretations will in fact produce type variable environments.

Type variable environments are modelled as lists of sets of values. They are implemented using the following syntax, where χ represents a set of values.

$$\text{Type variable environments} \quad \eta \quad ::= \quad \cdot \mid \eta, \chi$$

This representation is sufficiently expressive because it takes advantage of our de Bruijn representation of type variables. Type variable environments refer to type variables implicitly by index in the list.

Now let us look at the semantic interpretation of renamings.

$$
\begin{aligned}
\mathcal{V}^{**}[\uparrow^n](\eta) &= \mathrm{pop}\, n\, \eta \\
\mathcal{V}^{**}[\pi', k](\eta) &= \mathcal{V}^{**}[\pi'](\eta), \eta(k)
\end{aligned}
$$

Here we refer to a "pop" function which removes elements from a type variable environment, and a lookup function $\eta(k)$ which retrieves the set at a given index. These are similar to the operations we used earlier for renamings and substitutions. They are all defined in our mechanization and we proved some composition properties about them.

Next we see the interpretation of substitutions.

$$
\begin{aligned}
\mathcal{V}^{*}[\uparrow^n](\eta) &= \mathrm{pop}\, n\, \eta \\
\mathcal{V}^{*}[\sigma', T](\eta) &= \mathcal{V}^{*}[\sigma'](\eta), \mathcal{V}[T](\eta)
\end{aligned}
$$

The idea of both of these definitions is to capture the notion of a shift and to bootstrap the semantic interpretation of types.

The $\mathcal{V}^*$ and $\mathcal{V}^{**}$ notation here is intentionally similar to the $\mathcal{V}$ notation of semantic types. These semantic notions are all intended to compose in a regular fashion. This is made precise in the following set of lemmas, which correspond to the four operations we defined in Section 6.3.

$$\mathcal{V}^{**}[\pi_1[\pi_2]](\eta) = \mathcal{V}^{**}[\pi_1](\mathcal{V}^{**}[\pi_2](\eta)) \tag{2.1}$$

$$\mathcal{V}[T[\pi]](\eta) = \mathcal{V}[T](\mathcal{V}^{**}[\pi](\eta)) \tag{2.2}$$

$$\mathcal{V}^{*}[\sigma[\pi]](\eta) = \mathcal{V}^{*}[\sigma](\mathcal{V}^{**}[\pi](\eta)) \tag{2.3}$$

$$\mathcal{V}[T[\sigma]](\eta) = \mathcal{V}[T](\mathcal{V}^{*}[\sigma](\eta)) \tag{2.4}$$

We have proved all of these properties in our mechanization. As with the definitions involving renamings and substitutions, each one depends on the one before it.

2.4.5 Modelling Sets and Semantics

In this section we briefly explain how we encode sets and some of our semantic structures in Coq. In particular we explain our need for higher-kinded polymorphism, which motivated the decision to use Coq.

Let us first look at our encoding of sets, which are the target of our semantic interpretation $\mathcal{V}$. We parameterize sets by their element type `A`, so that a set is simply a function from `A` to `Prop`, the *kind* of propositions.

```
Definition set (A : Type) : Type := A -> Prop.
```

For example, a set of values $\mathcal{C} \in \mathcal{P}(\Omega)$ is represented by a function of type `value -> Prop`, indicating whether a given value is in the set or not. This representation works well for implementing concepts such as set intersection, the semantic function space, and our pre-fixed point operator $\boldsymbol{\mu}$. The latter is defined in Coq as below.

```
Definition prefp (F : set value -> set value) : set value :=
  fun v => forall C : set value,
    (forall X, subset X C -> subset (F X) C) -> C v.
```

Note that in this definition we universally quantify over `C` and `X` which are *type constructors* (of type `value -> Prop`). This requires *higher-kinded* polymorphism, as opposed to ordinary (System F style) polymorphism which only allows quantification over types. This is the main reason we chose Coq over other reasoning languages.

There is a challenge in reasoning with sets due to Coq's notion of equality. Because we encode sets as functions, the natural definition of set equality is by extensionality: that is, two sets should be equal if they contain exactly the same elements. Once we define set equality, we would like to use it by freely substituting sets for equal sets in larger formulae. However, Coq does not immediately recognize our notion of equality as it is different to the definitional, intensional equality. The same situation occurs for other structures with custom definitions of equality such as type variable environments.

For example, if we know the type variable environments η and η' are equal, we would like to rewrite $\mathcal{V}[T](\eta)$ to $\mathcal{V}[T](\eta')$ in a larger formula. Unfortunately this transformation is not directly justified in Coq because sets and type variable environments both use custom notions of equality. Hence we must prove instances of substitutivity (replacing "equals for equals") for each context in which we hope to substitute equal structures. We must also prove symmetry, reflexivity and transitivity of our equality relations in order to use them effectively. Though technically easy, these proofs are tedious and should perhaps be avoided using richer facilities for equality reasoning.

This issue suggests that a logic based on rewriting instead of type theory (where equality is a notoriously hard issue) could be effective here. For example, the proof assistant Isabelle/HOL [Nipkow et al., 2002] allows the user to add definitions and lemmas as rewrite rules, which can be used to directly rewrite logical formulae. We should also mention Isabelle's support for variable binding via the Nominal package [Urban, 2008] based on the nominal logic of Pitts [2003]. It would be fruitful to investigate how much of our mundane proofs about equality and substitutions are alleviated in an Isabelle formalization, and whether new technical issues are introduced.

2.5 Conclusion

We have given a tutorial-style explanation of a termination proof for a simply typed lambda calculus with Mendler-style recursion. We hope to have familiarized the reader with some concepts which are fundamental for the remainder of this thesis. In particular we presented theory of recursive types, Mendler recursion, environment-based evaluation, and proof of termination using a logical predicate semantics. We will use a similar setup for the theory of Tores in Section 3.3. Finally, we described our mechanization of this chapter's theory in Coq, with both its strengths and shortcomings.

Chapter 3

Indexed Recursive and Stratified Types

3.1 Introduction

In this chapter, we describe our core language Tores, which supports writing inductive proofs as total recursive programs about an index domain. It allows us to reason directly by induction on index terms and by Mendler-style recursion on indexed recursive types. In addition, Tores offers stratified types that are defined by well-founded recursion on terms in the index language. This guarantees their well-foundedness while avoiding any positivity restriction. Unlike indexed recursive types, stratified types can only be unfolded according to their index arguments. Stratified types are particularly convenient when working with nonpositive recursive definitions, which appear in logical relations arguments.

Stratified types offer similar advantages to an indexed type theory as large eliminations do in a fully fledged dependently typed language, or as the use of rewriting to express type-level computation does in the proof theory described in [Baelde and Nadathur, 2012]. In contrast to these approaches, we avoid general type-level computation by offering only restricted unfolding of stratified types. This results in a simpler metatheoretic development. Type checking remains decidable and proving subject reduction is straightforward. Our main result in this paper is a termination proof for Tores using a logical predicate semantics. The combination of indexed recursive and stratified types in our language seems to provide a sweet-spot in leveraging the expressiveness of our logic with the simplicity of our metatheory.

We already use the combination of indexed recursive types and stratified types in the programming and proof environment Beluga, where the index language is an extension of the logical framework LF together with first-class contexts and substitutions [Nanevski et al., 2008, Pientka, 2008, Cave and Pientka, 2012]. This allows elegant implementations of proofs using logical relations [Cave and Pientka, 2013, 2015] and normalization by evaluation [Cave and Pientka, 2012]. Tores can be seen as small kernel into which we elaborate total Beluga programs, thereby providing post-hoc justification of viewing Beluga programs as

inductive proofs.

3.2 Index Language for Tores

The design of Tores is parametric over an index language. We follow the model of Thibodeau et al. [2016], staying as abstract as possible from the specific index language and stating the general conditions it must satisfy. However, there are aspects of Tores that require inspection of the particular index language, namely the structure of stratified types and induction terms. We will draw attention to these points in the development.

To illustrate the required structure for a concrete index language, we use the simple example of natural numbers. In practice, however, we can consider using more useful index languages such as those of strings, types [Cheney and Hinze, 2003, Xi et al., 2003], or (contextual) LF [Cave and Pientka, 2012]. It is important to note that, for most of our design, we accommodate a general index language up to the complexity of Contextual LF. Thus we treat index types and Tores kinds as dependently typed, although we use natural numbers in stratified types and induction terms.

The abstract requirements of our index language are listed throughout this section. To summarize them here, they are: decidable type checking, decidable equality, standard substitution principles, decidable unification as well as sound and complete matching. Implicitly, we also require that each index type intended for use in stratified types and induction terms should have a well-founded recursion scheme, i.e. an induction principle. For an index language of Contextual LF, for example, the recursion scheme can be generated using a covering set of index terms for each index type [Pientka and Abel, 2015]. This inductive structure is necessary to show decidability of type checking (Thm 14) and termination (Thm 31) of Tores.

3.2.1 General Structure

We refer to a term in the index language as an *index term* M, which may have an *index type* U. In the case of natural numbers, there is a single index type nat, and index terms are built from 0, suc, and variables u which must be declared in an *index context* Δ.

Index types	U	$:= \mathsf{nat}$
Index terms	M	$:= 0 \mid \mathsf{suc}\, M \mid u$
Index contexts	Δ	$:= \cdot \mid \Delta, u{:}U$
Index substitutions	Θ	$:= \cdot \mid \Theta, M/u$

Tores relies on typing for index terms which we give below for natural numbers (see Fig. 3.1). The equality judgment for natural numbers is given simply by reflexivity (syntactic equality). We also give typing for index substitutions, which supply an index term for each index variable in the domain Δ. Note that the definitions of well-formed contexts and well-typed index substitutions are generic to the particular index terms and types.

$\boxed{\vdash \Delta\ \mathsf{ictx}}$ Index context Δ is well-formed

$$\frac{}{\vdash \cdot\ \mathsf{ictx}} \qquad \frac{\vdash \Delta\ \mathsf{ictx} \quad \Delta \vdash U\ \mathsf{itype}}{\vdash \Delta, u{:}U\ \mathsf{ictx}}$$

$\boxed{\Delta \vdash U\ \mathsf{itype}}$ Index type U is well-kinded

$$\frac{}{\Delta \vdash \mathsf{nat}\ \ \mathsf{itype}}$$

$\boxed{\Delta \vdash M : U}$ Index term M has index type U in index context Δ

$$\frac{u{:}U \in \Delta}{\Delta \vdash u : U} \qquad \frac{}{\Delta \vdash 0 : \mathsf{nat}} \qquad \frac{\Delta \vdash M : \mathsf{nat}}{\Delta \vdash \mathsf{suc}\, M : \mathsf{nat}}$$

$\boxed{\Delta \vdash M = N}$ Index term M is equal to N

$$\frac{}{\Delta \vdash M = M}$$

$\boxed{\Delta' \vdash \Theta : \Delta}$ Index substitution Θ maps index variables from Δ to Δ'

$$\frac{}{\Delta' \vdash \cdot : \cdot} \qquad \frac{\Delta' \vdash \Theta : \Delta \quad \Delta' \vdash M : U[\Theta]}{\Delta' \vdash \Theta, M/u : \Delta, u{:}U}$$

Figure 3.1: Index language structure

We require that both typing and equality of index terms be decidable in order for type checking of Tores programs to be decidable.

Requirement 1. *Index type checking is decidable.*

Requirement 2. *Index equality is decidable.*

3.2.2 Substitutions

We describe our index language using both a single index substitution operation $M[N/u]$ and a simultaneous substitution operation $M[\Theta]$. For example, simultaneous substitution on natural numbers is defined below.

Definition 7 (Simultaneous substitution on natural numbers)**.**

$$0[\Theta] = 0$$
$$(\mathsf{suc}\, M)[\Theta] = \mathsf{suc}\, M[\Theta]$$
$$u[\Theta] = \Theta(u)$$

Simultaneous substitutions can also be composed (generically to the index language) as follows.

Definition 8 (Composition of index substitutions)**.** Suppose $\Delta_1 \vdash \Theta_1 : \Delta$ and $\Delta_2 \vdash \Theta_2 : \Delta_1$. Then $\Delta_2 \vdash \Theta_1[\Theta_2] : \Delta$ where

$$(\cdot)[\Theta_2] = \Theta_2$$
$$(\Theta_1', M/u)[\Theta_2] = \Theta_1'[\Theta_2], M[\Theta_2]/u$$

With these definitions we can state our required properties of single and simultaneous substitutions. These say that substitutions preserve typing (3.1 and 3.2) and equality (3.3) and are associative (3.4).

Requirement 3 (Index substitution principles)**.**

3.1. If $\Delta_1, u{:}U', \Delta_2 \vdash M : U$ *and* $\Delta_1 \vdash N : U'$ *then* $\Delta_1, \Delta_2[N/u] \vdash M[N/u] : U[N/u]$.

3.2. If $\Delta' \vdash \Theta : \Delta$ *and* $\Delta \vdash M : U$ *then* $\Delta' \vdash M[\Theta] : U[\Theta]$.

3.3. If $\Delta' \vdash \Theta : \Delta$ *and* $\Delta \vdash M = N$ *then* $\Delta' \vdash M[\Theta] = N[\Theta]$.

3.4. If $\Delta \vdash M : U$ *and* $\Delta_1 \vdash \Theta_1 : \Delta$ *and* $\Delta_2 \vdash \Theta_2 : \Delta_1$ *then* $M[\Theta_1][\Theta_2] = M[\Theta_1[\Theta_2]]$.

3.2.3 Unification

The next aspect of the index language we rely on is a unification procedure to generate a *most general unifier (MGU)*. A *unifier* for index terms M and N in a context Δ is a substitution Θ which transforms M and N into syntactically equal terms in another context Δ'. That is, $\Delta' \vdash \Theta : \Delta$ and $\Delta' \vdash M[\Theta] = N[\Theta]$. Θ is "most general" if it does not make more commitments to variables than absolutely necessary. A unifying substitution Θ only makes sense together with its range Δ', so we usually write them as a pair $(\Delta' \mid \Theta)$. In general, there may be more than one MGU for a particular unification problem, or none at all. However, we require here that each problem has at most one MGU up to α-equivalence. We write the generation of an MGU using the judgment $\Delta \vdash M \doteqdot N \searrow P$, where P is either the MGU $(\Delta' \mid \Theta)$ if it exists or $\#$ representing that unification failed. We show the unification rules for natural numbers for illustration.

$\boxed{\Delta \vdash M \doteqdot N \searrow P}$ Index terms M and N have MGU P

$$\frac{}{\Delta \vdash 0 \doteqdot 0 \searrow (\Delta \mid \cdot)} \qquad \frac{\Delta \vdash M \doteqdot N \searrow P}{\Delta \vdash \mathsf{suc}\, M \doteqdot \mathsf{suc}\, N \searrow P} \qquad \frac{}{\Delta \vdash u \doteqdot u \searrow (\Delta \mid \cdot)}$$

$$\frac{\Delta = \Delta_1, u{:}U, \Delta_2 \quad \Delta' = \Delta_1, \Delta_2[M/u]}{\Delta \vdash u \doteqdot M \searrow (\Delta' \mid M/u)} \qquad \frac{\Delta = \Delta_1, u{:}U, \Delta_2 \quad \Delta' = \Delta_1, \Delta_2[M/u]}{\Delta \vdash M \doteqdot u \searrow (\Delta' \mid M/u)} \qquad u \notin \mathrm{FV}(M)$$

$$\frac{}{\Delta \vdash 0 \doteqdot \mathsf{suc}\, M \searrow \#} \qquad \frac{}{\Delta \vdash \mathsf{suc}\, M \doteqdot 0 \searrow \#}$$

The unification procedure is required for type checking the equality elimination forms $\mathtt{eq}\, s\, \mathtt{with}\, (\Delta'.\Theta \mapsto t)$ and $\mathtt{eq\_abort}\, s$ in Tores, which we explain in Section 3.3.2. In each form, the term s is a witness of an index equality $\Delta \vdash M = N$. In order to use this equality (or determine that it is spurious), we

perform unification $\Delta \vdash M \doteqdot N \searrow P$ and check that the result P matches the source term. For the term $\texttt{eq}\, s\, \texttt{with}\, (\Delta'.\Theta \mapsto t)$, we check that P is an α-equivalent unifier to the provided one $(\Delta' \mid \Theta)$. For the failure term $\texttt{eq\_abort}\, s$ we check that P is $\#$, yielding a contradiction. Hence our type checking algorithm for Tores relies on a sound and complete unification procedure. We summarize our requirements for unification below.

Requirement 4 (Decidable unification)**.** *Given index terms M and N in a context Δ, the judgment $\Delta \vdash M \doteqdot N \searrow P$ is decidable. Either P is $(\Delta' \mid \Theta)$, the unique MGU up to α-equivalence, or P is $\#$ and there is no unifier.*

3.2.4 Matching

Another notion, similar to unification, which we require of the index language is that of index *matching*. This is an asymmetric form of unification: given terms M in Δ and N in Δ', matching identifies a substitution Θ such that $\Delta' \vdash M[\Theta] = N$. We write this using the judgment $\Delta \vdash M \doteq N \searrow (\Delta' \mid \Theta)$.

Matching is used during evaluation of the equality elimination $\texttt{eq}\, s\, \texttt{with}\, (\Delta'.\Theta \mapsto t)$ to extend the substitution Θ to a full index environment (grounding substitution) θ. To achieve this, we must lift the notion of matching to the level of index substitutions. This can be done generically given an algorithm for matching index terms. The judgment $\Delta \vdash \Theta_1 \doteq \Theta_2 \searrow (\Delta' \mid \Theta)$ says that matching discovered a substitution Θ such that $\Delta' \vdash \Theta_1[\Theta] = \Theta_2$.

To illustrate an algorithm for index matching, we provide the rules for our natural number domain in Fig. 3.2. We also show the generic lifting of the algorithm to match index substitutions.

$\boxed{\Delta \vdash M \doteq N \searrow (\Delta' \mid \Theta)}$ Index term M matches index term N under Θ

$$\frac{\Delta = \Delta_1, u : \mathsf{nat}, \Delta_2 \quad \Delta' = \Delta_1, \Delta_2}{\Delta \vdash u \doteq N \searrow (\Delta' \mid \mathsf{id}_{\Delta_1}, N/u, \mathsf{id}_{\Delta_2})}$$

$$\frac{}{\Delta \vdash z \doteq z \searrow (\Delta \mid \mathsf{id}_\Delta)} \qquad \frac{\Delta \vdash M \doteq N \searrow (\Delta' \mid \Theta)}{\Delta \vdash \mathsf{suc}\, M \doteq \mathsf{suc}\, N \searrow (\Delta' \mid \Theta)}$$

$\boxed{\Delta \vdash \Theta_1 \doteq \Theta_2 \searrow (\Delta' \mid \rho)}$ Index substitution Θ_1 matches index substitution Θ_2 under ρ

$$\frac{}{\Delta \vdash \cdot \doteq \cdot \searrow (\Delta \mid \mathsf{id}_\Delta)} \qquad \frac{\Delta \vdash \Theta_1 \doteq \Theta_2 \searrow (\Delta_1 \mid \rho_1) \quad \Delta_1 \vdash M[\rho_1] \doteq N \searrow (\Delta_2 \mid \rho_2)}{\Delta \vdash (\Theta_1, M/u) \doteq (\Theta_2, N/u) \searrow (\Delta_2 \mid \rho_1[\rho_2])}$$

Figure 3.2: Index matching for natural numbers and generic substitutions

We then require that index (substitution) matching is both sound and complete. We make these properties precise in our final requirements below.

Requirement 5 (Soundness of index matching)**.**

5.1. If $\Delta \vdash M : U$ *and* $\Delta \vdash M \doteq N \searrow (\Delta' \mid \Theta)$ *then* $\Delta' \vdash \Theta : \Delta$ *and* $\Delta' \vdash M[\Theta] = N$.

5.2. If $\Delta_1 \vdash \Theta_1 : \Delta$ *and* $\Delta_1 \vdash \Theta_1 \doteq \Theta_2 \searrow (\Delta_2 \mid \Theta)$ *then* $\Delta_2 \vdash \Theta : \Delta_1$ *and* $\Delta_2 \vdash \Theta_1[\Theta] = \Theta_2$.

Requirement 6 (Completeness of index matching)**.** *Suppose* $\vdash \theta : \Delta$ *and* $\vdash M[\theta] = N[\theta]$ *and* $\Delta \vdash M \doteqdot N \searrow (\Delta' \mid \Theta)$. *Then* $\Delta' \vdash \Theta \doteq \theta \searrow (\cdot \mid \theta')$.

3.2.5 Spines

Finally, we can lift the kinding, typing, equality and matching rules to *spines* of index terms and types generically. We write $\cdot$ and $(\cdot)$ for the empty spines of terms and types respectively. If M_0 is a term and $\overrightarrow{M}$ is a spine, then $M_0, \overrightarrow{M}$ is a spine. Similarly if $u_0{:}U_0$ is a type declaration and $(\overrightarrow{u{:}U})$ is a spine of type declarations, then $(u_0{:}U_0, \overrightarrow{u{:}U})$ is a spine of type declarations. The use of spines is convenient in setting up the types and terms of Tores. Unlike index contexts Δ and the corresponding index substitutions Θ which are built from right to left, spines are built from left to right.

Below we define well-kinded spines of index types and well-typed spines of index terms, which are generic to the particular index language.

$\boxed{\Delta \vdash (\overrightarrow{u{:}U})\ \mathsf{itype}}$ Spine $\overrightarrow{u{:}U}$ of index types is well-kinded

$$\frac{}{\Delta \vdash (\cdot)\ \mathsf{itype}} \qquad \frac{\Delta \vdash U_0\ \mathsf{itype} \quad \Delta, u_0{:}U_0 \vdash (\overrightarrow{u : U})\ \mathsf{itype}}{\Delta \vdash (u_0{:}U_0, \overrightarrow{u : U})\ \mathsf{itype}}$$

$\boxed{\Delta \vdash \vec{M} : (\overrightarrow{u{:}U})}$ Spine $\vec{M}$ of index terms have index types $(\overrightarrow{u{:}U})$

$$\frac{}{\Delta \vdash \cdot : (\cdot)} \qquad \frac{\Delta \vdash M_0 : U_0 \quad \Delta \vdash \vec{M} : (\overrightarrow{u{:}U})[M_0/u_0]}{\Delta \vdash M_0, \vec{M} : (u_0{:}U_0, \overrightarrow{u{:}U})}$$

Lemma 9. *Type checking of index spines is decidable.*

Proof. Simply rely on decidable type checking of single index terms (Req. 1). □

3.3 Specification of Tores

3.3.1 Types and Kinds

Besides unit, products and sums, Tores includes a nonstandard function type $(\overrightarrow{u{:}U}); T_1 \to T_2$ which combines a dependent function type and a simple function type. It binds a number of index variables $\overrightarrow{u{:}U}$ which may appear in both T_1 and T_2. This will be convenient in defining our Mendler-style recursion operator. If the spine of type declarations is empty, then $(\cdot); T_1 \to T_2$ degenerates to the simple function space. Note that we can abstract over the index variables $\overrightarrow{u{:}U}$ in any given type T by $(\overrightarrow{u{:}U}); 1 \to T$. We also include a type $\Sigma u{:}U.\, T$ quantifying existentially over an index, and a type of index equality $M = N$. These two types are useful for expressing equality constraints on indices.

Kinds	K	$::= * \mid \Pi u{:}U.\, K$
Types	T	$::= 1 \mid T_1 \times T_2 \mid T_1 + T_2 \mid (\overrightarrow{u{:}U});\, T_1 \to T_2 \mid \Sigma u{:}U.\, T \mid M = N$
		$\mid\ T\, M \mid \Lambda\, u.\, T \mid X \mid \mu X{:}K.\, T \mid T_{\mathtt{Rec}}$
Stratified Types	$T_{\mathtt{Rec}}$	$::= \mathtt{Rec}_K (0 \mapsto T_0 \mid \mathsf{suc}\, u,\, X \mapsto T_s)$
Index Contexts	Δ	$::= \cdot \mid \Delta, u{:}U$
Type Var. Contexts	Ξ	$::= \cdot \mid \Xi, X{:}K$
Typing Contexts	Γ	$::= \cdot \mid \Gamma, x{:}T$

We model recursive and stratified types as type constructors of kind $\Pi\overrightarrow{u{:}U}.\, *$. Both forms introduce type variables X, which we track in the type variable context Ξ. There is no positivity condition on recursive types, as the typing rules for Mendler-recursion enforce termination without it. A stratified type is defined by primitive recursion on an index term. Here we must rely on the specific index language, which we show for natural numbers. For the index type nat, we define a stratified type $T_{\mathtt{Rec}} = \mathtt{Rec}_K\, (0 \mapsto T_0 \mid \mathsf{suc}\, u,\, X \mapsto T_s)$. The two branches correspond to the two constructors 0 and suc. Intuitively, $T_{\mathtt{Rec}}\, 0$ will behave like T_0 and $T_{\mathtt{Rec}}\, (\mathsf{suc}\, M)$ will behave like $T_s[M/u, T_{\mathtt{Rec}}\, M/X]$, where we substitute the smaller index and type for the index variable and type variable respectively.

Our kinding rules are shown in Fig. 3.3. We employ a bidirectional kinding system to make it evident when kinds can be inferred and when kinding annotations are necessary. Kinding depends on two contexts: the index context Δ, since index variables may appear in types and kinds, and the type variable context Ξ. Note that in general the kinds assigned to type variables in Ξ may depend on the index variables in Δ. The checking judgment $\Delta; \Xi \vdash T \Leftarrow K$ takes all expressions as inputs as verifies that the type is well-kinded. The inference judgment $\Delta; \Xi \vdash T \Rightarrow K$ takes the contexts and type as input to produce a kind as output.

In our rules, the kind $*$ of types is always checked, and may rely on inference via the conversion rule. In addition, type-level lambdas $\Lambda\, u.\, T$ are checked against a kind $\Pi u{:}U.\, K$ by checking the body T under an extended index context $\Delta, u{:}U$. On the other hand, kinds are inferred (synthesized) for type variables by looking them up in the context Ξ, as well as for type-level applications, recursive types and stratified types. Subtly, the kinding for type applications $T\, M$ requires that T synthesize a kind: in particular T cannot be a type-level lambda, which would be checked against a kind. This means that types in $*$ do not arise from reducible lambda applications: lambdas must occur within recursive or stratified types. Finally, recursive and stratified types synthesize the kinds in their annotations. Recursive type variables are added to the context Ξ for checking the body of the type. Stratified types require checking the constituent types using the index information gleaned in each branch: T_0 is checked against $K'[0/u]$ and T_s against $K'[\mathsf{suc}\, u/u]$.

Example 10. To illustrate indexed recursive types and stratified types, we consider vectors, i.e. lists that are indexed by their length, where we use A for the type of elements. Vectors are of kind $\Pi n{:}\, \mathsf{nat}\, .*$. We omit the kind annotation for better readability in the subsequent type definitions. We first define vectors using an indexed recursive type, explicit equality and an existential type: $\mathsf{Vec}_\mu \equiv \mu V.\, \Lambda\, n.\, n = 0 + \Sigma m{:}\, \mathsf{nat}\, .\, n = \mathsf{suc}\, m \times (A \times V\, m)$

Vectors can also be defined as a stratified type: $\mathsf{Vec}_S \equiv \mathtt{Rec}\, (0 \mapsto 1 \mid \mathsf{suc}\, m,\, V \mapsto A \times V)$

$$\boxed{\Delta;\Xi \vdash T \Leftarrow K} \qquad \text{Check type } T \text{ against kind } K$$

$$\frac{}{\Delta;\Xi \vdash 1 \Leftarrow *} \qquad \frac{\Delta;\Xi \vdash T_1 \Leftarrow * \quad \Delta;\Xi \vdash T_2 \Leftarrow *}{\Delta;\Xi \vdash T_1 \times T_2 \Leftarrow *} \qquad \frac{\Delta;\Xi \vdash T_1 \Leftarrow * \quad \Delta;\Xi \vdash T_2 \Leftarrow *}{\Delta;\Xi \vdash T_1 + T_2 \Leftarrow *}$$

$$\frac{\Delta \vdash (\overrightarrow{u{:}U})\ \mathsf{itype} \quad \Delta, \overrightarrow{u{:}U};\Xi \vdash S \Leftarrow * \quad \Delta, \overrightarrow{u{:}U};\Xi \vdash T \Leftarrow *}{\Delta;\Xi \vdash (\overrightarrow{u{:}U});\ S \to T \Leftarrow *} \qquad \frac{\Delta \vdash U\ \mathsf{itype} \quad \Delta, u{:}U;\Xi \vdash T \Leftarrow *}{\Delta;\Xi \vdash \Sigma u{:}U.\, T \Leftarrow *}$$

$$\frac{\Delta \vdash M : U \quad \Delta \vdash N : U}{\Delta;\Xi \vdash M = N \Leftarrow *} \qquad \frac{\Delta, u{:}U;\Xi \vdash T \Leftarrow K}{\Delta;\Xi \vdash \Lambda\, u.\, T \Leftarrow \Pi u{:}U.\, K} \qquad \frac{\Delta;\Xi \vdash T \Rightarrow *}{\Delta;\Xi \vdash T \Leftarrow *}$$

$$\boxed{\Delta;\Xi \vdash T \Rightarrow K} \qquad \text{Infer a kind } K \text{ for type } T$$

$$\frac{X{:}K \in \Xi}{\Delta;\Xi \vdash X \Rightarrow K} \qquad \frac{\Delta;\Xi \vdash T \Rightarrow \Pi u{:}U.\, K \quad \Delta \vdash M : U}{\Delta;\Xi \vdash T\, M \Rightarrow K[M/u]} \qquad \frac{\Delta;\Xi, X{:}K \vdash T \Leftarrow K}{\Delta;\Xi \vdash \mu X{:}K.\, T \Rightarrow K}$$

$$\frac{K = \Pi u{:}\mathsf{nat}\,.\, K' \quad \Delta;\Xi \vdash T_0 \Leftarrow K'[0/u] \quad \Delta, u{:}\mathsf{nat};\Xi, X{:}K' \vdash T_s \Leftarrow K'[\mathsf{suc}\,u/u]}{\Delta;\Xi \vdash \mathtt{Rec}_K(0 \mapsto T_0 \mid \mathsf{suc}\,u,\ X \mapsto T_s) \Rightarrow K}$$

Figure 3.3: Kinding rules for TORES

In this case equality reasoning is implicit. While we have a choice how to define vectors, there are of course types that have to be defined either as a recursive or a stratified type, as not all type definitions directly recurse on an index.

Example 11. To illustrate the usefulness of stratified types, we give here the definition of reducible sets of simply-typed lambda terms. In this example, our index domain consists of index objects modelling simple types, `unit` and `arr` A B, of index type `tp` and index objects modelling lambda terms, i.e. (), `lam` $x.M$, and `app` M N, of index type `tm`.

We can then define reducible sets using a stratified type of kind $\Pi a{:}\mathtt{tp}.\Pi m{:}\mathtt{tm}.*$. We rely on a recursive type definition of `Halt` m to describe that the term m steps to a value.

$$\begin{aligned}\mathtt{Rec}\ (\ &\mathtt{unit} && \mapsto \Lambda m.\mathtt{Halt}\ m\\ \mid\ &\mathtt{arr}\ a\ b, R_a, R_b && \mapsto \Lambda m.\mathtt{Halt}\ m \times (n{:}\mathtt{tm});\ R_a\ n \to R_b\ (\mathtt{app}\ m\ n)\ )\end{aligned}$$

We will revisit this example in detail in Section 3.5.

3.3.2 Terms

TORES contains many common constructs found in functional programming languages, such as unit, pairs, case expressions, injections and fold operators to construct data. We focus here on the less standard constructs: our definition and use of functions, witnesses for equality, support for recursion on data structures defined by a recursive type and support for induction on index objects.

$$
\begin{array}{rcl}
\text{Terms } t, s & ::= & x \mid \langle\rangle \mid \lambda\, \vec{u}, x.\, t \mid t\, \vec{M}\, s \mid \langle t_1, t_2\rangle \mid \mathtt{split}\, s\, \mathtt{as}\, \langle x_1, x_2\rangle\, \mathtt{in}\, t \\
& \mid & \mathtt{in}_i\, t \mid (\mathtt{case}\, t\, \mathtt{of}\, \mathtt{in}_1\ x_1 \mapsto t_1 \mid \mathtt{in}_2\ x_2 \mapsto t_2) \\
& \mid & \mathtt{pack}\, (M, t) \mid \mathtt{unpack}\, t\, \mathtt{as}\, (u, x)\, \mathtt{in}\, s \\
& \mid & \mathtt{refl} \mid \mathtt{eq}\, s\, \mathtt{with}\, (\Delta.\Theta \mapsto t) \mid \mathtt{eq\_abort}\, s \\
& \mid & \mathtt{in}_\mu\, t \mid \mathtt{rec}\, f.\, t \mid \mathtt{in}_l\, t \mid \mathtt{out}_l\, t \mid \mathtt{ind}\, t_0\, (u,\ f.\, t_s) \mid t{:}T
\end{array}
$$

Recall that in our type language, we combine the dependent function space with the simple function type in $(\overrightarrow{u{:}U});\, T_1 \to T_2$. Similarly, we combine abstraction over index variables $\vec{u}$ and term variable x in our function definition, written as $\lambda\, \vec{u}, x.\, t$. The corresponding application form is written $t\, \vec{M}\, s$. The term t of function type $(\overrightarrow{u{:}U});\, T_1 \to T_2$ receives first a spine $\vec{M}$ of index objects followed by a term s. Each equality type $M = N$ has at most one inhabitant $\mathtt{refl}$ witnessing the equality. There are two elimination forms for equality: the term $\mathtt{eq}\, s\, \mathtt{with}\, (\Delta.\Theta \mapsto t)$ uses an equality proof s for $M = N$ together with a unifier Θ to refine the body t in a new index context Δ. It may also be the case that the equality witness s is false, in which case we have reached a contradiction and abort using the term $\mathtt{eq\_abort}\, s$. Both forms are necessary to make use of equality constraints that arise from indexed type definitions and to show that some cases are impossible.

Let us explain the introduction and elimination forms for recursive and stratified types. The "fold" syntax $\mathtt{in}_\mu$ introduces terms of recursive types, and $\mathtt{in}_l$ introduces terms of stratified types. Here l ranges over constructors in the index language, for example 0 and suc. An important difference is in how we eliminate recursive and stratified types.

We can analyze data defined by a recursive type using Mendler-style recursive programs $\mathtt{rec}\, f.\, t$. This gives a powerful means of recursion in our language while still ensuring termination. Stratified types can only be unfolded using $\mathtt{out}_l$ according to the index. To take full advantage of stratified types, we also allow programmers to use well-founded recursion over index objects, writing $\mathtt{ind}\, t_0\, (u,\ f.\, t_s)$. Intuitively, if the index object is 0, then we pick the first branch and execute t_0; if the index object is $\mathsf{suc}\, M$ then we pick the second branch instantiating u with M and allowing recursive calls f inside t_s.

Example 12. Here we continue the example of vectors from Example 10. Recall that vectors can be defined as Vec_μ using an indexed recursive type, or as Vec_S using a stratified type. As we demonstrate, the definition we choose impacts how we write programs to analyze vectors. We show the difference by writing a simple function that recursively copies a vector, using Mendler-style recursion for Vec_μ and induction on natural numbers for Vec_S.

We first show the program using recursive types.

$$
\begin{array}{l}
\mathit{copy} : (n{:}\,\mathsf{nat});\, \mathsf{Vec}_\mu\, n \to \mathsf{Vec}_\mu\, n \equiv \mathtt{rec}\, f.\, \lambda\, n, v.\, \mathtt{case}\, v\, \mathtt{of} \\
\quad \mid \mathtt{in}_1\, z \ \mapsto\ \mathtt{in}_\mu\, (\mathtt{in}_1\, z) \\
\quad \mid \mathtt{in}_2\, s \ \mapsto\ \mathtt{unpack}\, s\, \mathtt{as}\, (m, p)\, \mathtt{in} \\
\qquad\qquad\ \ \mathtt{split}\, p\, \mathtt{as}\, \langle e, p'\rangle\, \mathtt{in} \\
\qquad\qquad\ \ \mathtt{split}\, p'\, \mathtt{as}\, \langle h, t\rangle\, \mathtt{in} \\
\qquad\qquad\ \ \mathtt{in}_\mu\, (\mathtt{in}_2\, (\mathtt{pack}\, (m, \langle e, \langle h, f\, m\, t\rangle\rangle))))
\end{array}
$$

Here we use recursion and case analysis of the input vector to reconstruct the vector as output. In the case where we receive a non-empty list, we take it apart and expose the equality proofs, before reassembling the list. The recursion is valid according to the Mendler typing rule since the recursive call to f is made on the tail of the input vector.

Now we show the program using induction on natural numbers and unfolding the stratified type definition of Vec_S. Here we first split on the natural number argument and then unfold the vector itself. In this case the equality constraints are handled silently by the type checker.

$$
\begin{array}{l}
\mathit{copy} : (n\colon \mathsf{nat});\ 1 \to \mathsf{Vec}_S\ n \to \mathsf{Vec}_S\ n \equiv \\
\quad \mathtt{ind}\,(\ 0 \quad\ \mapsto\ \ \lambda v.\, \mathtt{in}_0\ \langle\rangle \\
\qquad \mid\ \mathsf{suc}\, m,\ f_m \ \mapsto\ \ \lambda v.\, \mathtt{split}\,(\mathtt{out}_{\mathsf{suc}}\, v)\ \mathtt{as}\ \langle h, t\rangle\ \mathtt{in}\ \ \mathtt{in}_{\mathsf{suc}}\ \langle h, f_m\, t\rangle)
\end{array}
$$

Example 13. To motivate our final small example, note that Tores does not include an explicit notion of falsehood. This is because falsehood is definable using existing constructs: we can define the empty type as a recursive type $\bot \equiv \mu X\colon *\,.\, X$, and a contradiction term $\mathtt{abort} \equiv \mathtt{rec}\, f.\, f : \bot \to C$, for any type C. Our forthcoming termination result with the logical relation in Section 3.4.3 will show that the $\bot$ type contains no values and hence no closed terms, which implies logical consistency of Tores (not all propositions can be proven).

3.3.3 Typing Rules

We define a bidirectional type system in Fig. 3.4 with two mutually defined judgments: checking a term t against a type T and synthesizing a type T for a term t. We can move from checking mode to synthesis mode via the conversion rule (given first). Dually we can move from synthesis to checking if we have a type annotation.

Lambda terms are checked against the combined function type $(\overrightarrow{u{:}U});\ S \to T$ by checking the body under the contexts extended with the index variable and term argument typing assumptions. The introduction forms for unit, products and sums are all checked in the standard way. The introduction for an existential type is checked by checking the first component using index type checking and checking the second component against a type substituted with the index term. This differs from usual dependent pairs where both components come from the same term language.

Products, sums and existential types are all eliminated using pattern matching constructs and are checked in similar ways. Each elimination form is checked by synthesizing the type of the scrutinee and then checking the body or branches under contexts extended with typing assumptions for bound variables.

The introduction for an index equality type is simply `refl`, which is checked by checking equality in the index domain. The equality elimination rules are somewhat more interesting. Both elimination forms contain a term s for which an index equality type is synthesized. This equality is then realized via unification in the index domain, whose result must correspond to the elimination term (as described in Section 3.2.3). Specifically, for an $\mathtt{eq\_abort}\, s$ term, the unification must fail, establishing a contradiction and allowing the term to check against any type (proving any proposition). For the term $\mathtt{eq}\, s\, \mathtt{with}\, (\Delta'.\Theta \mapsto t)$, the unification

$\boxed{\Delta;\Xi;\Gamma \vdash t \Leftarrow T}$ Term t checks against input type T

$$\frac{\Delta;\Xi;\Gamma \vdash t \Rightarrow T}{\Delta;\Xi;\Gamma \vdash t \Leftarrow T} \qquad \frac{}{\Delta;\Xi;\Gamma \vdash \langle\rangle \Leftarrow 1} \qquad \frac{\Delta, \overrightarrow{u{:}U};\Xi;\Gamma, x{:}S \vdash t \Leftarrow T}{\Delta;\Xi;\Gamma \vdash \lambda\vec{u}, x.\, t \Leftarrow (\overrightarrow{u{:}U});\ S \to T}$$

$$\frac{\Delta;\Xi;\Gamma \vdash t_1 \Leftarrow T_1 \quad \Delta;\Xi;\Gamma \vdash t_2 \Leftarrow T_2}{\Delta;\Xi;\Gamma \vdash \langle t_1, t_2\rangle \Leftarrow T_1 \times T_2} \qquad \frac{\Delta;\Xi;\Gamma \vdash p \Rightarrow T_1 \times T_2 \quad \Delta;\Xi;\Gamma, x_1{:}T_1, x_2{:}T_2 \vdash t \Leftarrow T}{\Delta;\Xi;\Gamma \vdash \mathtt{split}\, p\, \mathtt{as}\, \langle x_1, x_2\rangle\, \mathtt{in}\, t \Leftarrow T}$$

$$\frac{\Delta;\Xi;\Gamma \vdash t \Leftarrow T_i}{\Delta;\Xi;\Gamma \vdash \mathtt{in}_i\, t \Leftarrow T_1 + T_2} \qquad \frac{\Delta;\Xi;\Gamma \vdash t \Rightarrow T_1 + T_2 \quad \Delta;\Xi;\Gamma, x_1{:}T_1 \vdash t_1 \Leftarrow S \quad \Delta;\Xi;\Gamma, x_2{:}T_2 \vdash t_2 \Leftarrow S}{\Delta;\Xi;\Gamma \vdash (\mathtt{case}\, t\, \mathtt{of}\, \mathtt{in}_1\, x_1 \mapsto t_1 \mid \mathtt{in}_2\, x_2 \mapsto t_2) \Leftarrow S}$$

$$\frac{\Delta \vdash M : U \quad \Delta;\Xi;\Gamma \vdash t \Leftarrow T[M/u]}{\Delta;\Xi;\Gamma \vdash \mathtt{pack}\,(M, t) \Leftarrow \Sigma u{:}U.\, T} \qquad \frac{\Delta;\Xi;\Gamma \vdash t \Rightarrow \Sigma u{:}U.\, T \quad \Delta, u{:}U;\Xi;\Gamma, x{:}T \vdash s \Leftarrow S}{\Delta;\Xi;\Gamma \vdash \mathtt{unpack}\, t\, \mathtt{as}\, (u, x)\, \mathtt{in}\, s \Leftarrow S}$$

$$\frac{\Delta \vdash M = N}{\Delta;\Xi;\Gamma \vdash \mathtt{refl} \Leftarrow M = N} \qquad \frac{\Delta;\Xi;\Gamma \vdash s \Rightarrow M = N \quad \Delta \vdash M \doteq N \searrow \#}{\Delta;\Xi;\Gamma \vdash \mathtt{eq\_abort}\, s \Leftarrow T}$$

$$\frac{\Delta;\Xi;\Gamma \vdash s \Rightarrow M = N \quad \Delta \vdash M \doteq N \searrow (\Delta' \mid \Theta) \quad \Delta';\Xi[\Theta];\Gamma[\Theta] \vdash t \Leftarrow T[\Theta]}{\Delta;\Xi;\Gamma \vdash \mathtt{eq}\, s\, \mathtt{with}\, (\Delta'.\Theta \mapsto t) \Leftarrow T}$$

$$\frac{\Delta;\Xi;\Gamma \vdash t \Leftarrow T[\overrightarrow{M/u}; \mu X{:}K.\, \Lambda\,\vec{u}.\, T/X]}{\Delta;\Xi;\Gamma \vdash \mathtt{in}_\mu\, t \Leftarrow (\mu X{:}K.\, \Lambda\,\vec{u}.\, T)\,\vec{M}} \qquad \frac{\Delta;\Xi, X{:}K;\Gamma, f{:}((\overrightarrow{u{:}U});\ X\vec{u} \to T) \vdash t \Leftarrow (\overrightarrow{u{:}U});\ S[\overrightarrow{u/v}] \to T}{\Delta;\Xi;\Gamma \vdash \mathtt{rec}\, f.\, t \Leftarrow (\overrightarrow{u{:}U});\ (\mu X{:}K.\, \Lambda\,\overrightarrow{v}.\, S)\,\vec{u} \to T}$$

$$\frac{\Delta;\Xi;\Gamma \vdash t_0 \Leftarrow T[0/u] \quad \Delta, u{:}\,\mathsf{nat};\Xi;\Gamma, f{:}T \vdash t_s \Leftarrow T[\mathsf{suc}\, u/u]}{\Delta;\Xi;\Gamma \vdash \mathtt{ind}\, t_0\, (u,\ f.\, t_s) \Leftarrow (u{:}\,\mathsf{nat});\ 1 \to T}$$

$$\frac{\Delta;\Xi;\Gamma \vdash t \Leftarrow T_0\,\vec{M}}{\Delta;\Xi;\Gamma \vdash \mathtt{in}_0\, t \Leftarrow T_{\mathtt{Rec}}\, 0\,\vec{M}} \qquad \frac{\Delta;\Xi;\Gamma \vdash t \Leftarrow T_s[N/u; (T_{\mathtt{Rec}}\, N)/X]\,\vec{M}}{\Delta;\Xi;\Gamma \vdash \mathtt{in}_{\mathsf{suc}}\, t \Leftarrow T_{\mathtt{Rec}}\, (\mathsf{suc}\, N)\,\vec{M}}$$

$\boxed{\Delta;\Xi;\Gamma \vdash t \Rightarrow T}$ Term t synthesizes output type T

$$\frac{x{:}T \in \Gamma}{\Delta;\Xi;\Gamma \vdash x \Rightarrow T} \qquad \frac{\Delta;\Xi;\Gamma \vdash t \Rightarrow (\overrightarrow{u{:}U});\ S \to T \quad \Delta \vdash \vec{M} : (\overrightarrow{u{:}U}) \quad \Delta;\Xi;\Gamma \vdash s \Leftarrow S[\overrightarrow{M/u}]}{\Delta;\Xi;\Gamma \vdash t\,\vec{M}\, s \Rightarrow T[\overrightarrow{M/u}]}$$

$$\frac{\Delta;\Xi;\Gamma \vdash t \Rightarrow T_{\mathtt{Rec}}\, 0\,\vec{M}}{\Delta;\Xi;\Gamma \vdash \mathtt{out}_0\, t \Rightarrow T_0\,\vec{M}} \qquad \frac{\Delta;\Xi;\Gamma \vdash t \Rightarrow T_{\mathtt{Rec}}\, (\mathsf{suc}\, N)\,\vec{M}}{\Delta;\Xi;\Gamma \vdash \mathtt{out}_{\mathsf{suc}}\, t \Rightarrow T_s[N/u; (T_{\mathtt{Rec}}\, N)/X]\,\vec{M}} \qquad \frac{\Delta;\Xi;\Gamma \vdash t \Leftarrow T}{\Delta;\Xi;\Gamma \vdash t{:}T \Rightarrow T}$$

Figure 3.4: Typing rules for Tores

must result in the MGU which by Req. 4 is α-equivalent to the supplied unifier. We then check the body t with the unifier $(\Delta' \mid \Theta)$ in full effect: we use the new index context Δ' and the substitution Θ applied to the contexts Ξ and Γ as well as the goal type T. Note that in general our kinds may be dependent on indices, which is why we apply Θ to the kinds in Ξ.

Our treatment of equality elimination is similar to the use of refinement substitutions to handle dependent pattern matching [Pientka and Dunfield, 2008, Cave and Pientka, 2012]. It is also inspired by equality elimination in proof theory [Tiu and Momigliano, 2012, McDowell and Miller, 2002, Schroeder-Heister, 1993]. In the latter line of work, type checking involves trying all unifiers from a *complete set of unifiers*, instead

of a single most general unifier. As a complete set of unifiers may be infinite in general, we believe our requirement for a unique MGU is a practical one for type checking.

Both indexed recursive types and stratified types are introduced using injections ($\mathtt{in}_\mu$ and $\mathtt{in}_l$). However, as mentioned earlier, they use different elimination forms. For recursive types, we employ a Mendler-style typing rule for recursion $\mathtt{rec}\, f.\, t$. We generalize Mendler's original formulation [Mendler, 1988] to an indexed type system. The idea is to constrain the type of the function variable f so that it can only be applied to structurally smaller data. This is achieved by declaring f of type $(\overrightarrow{u{:}U})$; $X\ \vec{u} \to T$ in the corresponding typing rule. Here X can only be used to construct types exactly one constructor smaller than the recursive type, so the use of f is guaranteed to be well-founded.

In the induction rule we check $\mathtt{ind}\, t_0\, (u,\, f.\, t_s)$ against $(u{:}\,\mathsf{nat});\ 1 \to T$ by checking t_0 against $T[0/u]$ and checking t_s against $T[\mathsf{suc}\, u/u]$ given a function f of type T. Morally f stands for a function that can be applied to data constrained by the predecessor u. The unfolding rules for stratified types mirror the corresponding folding rules.

Finally, we can synthesize the type of the application $t\ \vec{M}\ s$ by first synthesizing the type $(\overrightarrow{u{:}U})$; $S \to T$ for t, and then checking the arguments $\vec{M}$ and s against their corresponding types $(\overrightarrow{u{:}U})$ and $S[\overrightarrow{M/u}]$.

Theorem 14. *Type checking of terms is decidable.*

Proof. Observe that the typing rules are syntax directed, except for the very first rule converting from checking mode to synthesis mode. The bidirectional system therefore specifies the following algorithm. When possible, apply the rule corresponding to the term syntax. If no rule applies during checking, apply the conversion rule and continue in synthesis mode. If no rule applies during synthesis then the algorithm fails. Moreover if one of the index judgments fails then the algorithm fails.

This algorithm terminates for two main reasons. Firstly, each syntax-directed rule relies on checking or inference for syntactically smaller subterms. The conversion rule only moves from checking to synthesis so it cannot be applied in a cycle. Secondly, the judgments involving index terms are decidable due to the requirements in Section 3.2. Specifically, the following are decidable: type checking index terms (Req. 1) and spines (Lemma 9), index equality (Req. 2) and index unification (Req. 4). The former two judgments are used in checking against existential types and synthesizing types for applications, and the latter two are used in type checking the equality introduction and elimination rules. □

3.3.4 Operational Semantics

We define a big-step operational semantics using environments, which provide closed values for the free variables that may occur in a terms. The approach is similar to our operational semantics in Section 2.2.3, except here we require both an index environment θ and term environment σ. These environments will again be useful in our semantics in Section 3.4.3.

$$
\begin{array}{lll}
\text{Term environments} & \sigma & ::= \cdot \mid \sigma, v/x \\
\text{Function values} & g & ::= \lambda\,\vec{u}, x.\, t \mid \mathtt{rec}\, f.\, t \mid \mathtt{ind}\, t_0\, (u,\ f.\, t_s) \\
\text{Closures} & c & ::= (g)[\theta; \sigma] \\
\text{Values} & v & ::= c \mid \langle\rangle \mid \langle v_1, v_2\rangle \mid \mathtt{in}_i\, v \mid \mathtt{pack}\,(M, v) \mid \mathtt{refl} \mid \mathtt{in}_\mu\, v \mid \mathtt{in}_l\, v
\end{array}
$$

Values consist of unit, pairs, injections, reflexivity, and closures. Note that our closures also include those arising from induction terms. We show the typing for values and environments in Fig. 3.5, which is used to state the subject reduction theorem.

$\boxed{v : T}$ Value v has type T

$$
\frac{\cdot \vdash \theta : \Delta \quad \sigma : \Gamma[\theta] \quad \Delta; \cdot; \Gamma \vdash g \Leftarrow T}{(g)[\theta; \sigma] : T[\theta]} \qquad \frac{}{\langle\rangle : 1} \qquad \frac{\cdot \vdash M = N}{\mathtt{refl} : M = N}
$$

$$
\frac{v_1 : T_1 \quad v_2 : T_2}{\langle v_1, v_2\rangle : T_1 \times T_2} \qquad \frac{v : T_i}{\mathtt{in}_i\, v : T_1 + T_2} \qquad \frac{\cdot \vdash M : U \quad v : T[M/u]}{\mathtt{pack}\,(M, v) : \Sigma u{:}U.\, T} \qquad \frac{v : T[\overrightarrow{M/u}; \mu X{:}K.\, \Lambda\,\vec{u}.\, T/X]}{\mathtt{in}_\mu\, v : (\mu X{:}K.\, \Lambda\,\vec{u}.\, T)\,\vec{M}}
$$

$$
\frac{v : T_0\,\vec{M}}{\mathtt{in}_0\, v : T_{\mathtt{Rec}}\, 0\,\vec{M}} \qquad \frac{v : T_s[N/u; T_{\mathtt{Rec}}\, N/X]\,\vec{M}}{\mathtt{in}_{\mathsf{suc}}\, v : T_{\mathtt{Rec}}\,(\mathsf{suc}\, N)\vec{M}}
$$

$\boxed{\sigma : \Gamma}$ Environment σ has domain Γ

$$
\frac{}{\cdot : \cdot} \qquad \frac{\sigma : \Gamma \qquad v : T}{(\sigma, v/x) : \Gamma, x{:}T}
$$

Figure 3.5: Value and environment typing

The main evaluation judgment, $t[\theta; \sigma] \Downarrow v$, describes the evaluation of a term t under environments $\theta; \sigma$ to a value v. Here, t stands for a term in an index context Δ and term variable context Γ. The index environment θ provides closed index objects for all the index variables in Δ, while σ provides closed values for all the variables declared in Γ, i.e. $\vdash \theta : \Delta$ and $\sigma : \Gamma[\theta]$. For convenience, we factor out the application of a closure c to values $\vec{N}$ and v resulting in a value w using a second judgment, written $c \cdot \vec{N}\, v \Downarrow w$. This allows us to treat application of functions (lambda, recursion or induction) uniformly.

The evaluation rules for pairs, splits, injections, case expressions, pack and unpack terms are mostly straight-forward. Values evaluate to themselves. To evaluate a variable x in the environment $\theta; \sigma$, we look up the value x is bound to in σ. To evaluate an application $t\,\vec{M}\,s$ under environments θ and σ, we evaluate t in the environment $\theta; \sigma$ to a closure c, evaluate s to a value v, and then compute the result of applying the closure c to the index terms $\overrightarrow{M[\theta]}$ and the value v. If c stands for a function, we simply extend both environments and continue to evaluate the body. If c stands for a recursive function $\mathtt{rec}\, f.\, t$, we continue evaluating the body t in the extended environment $\sigma, (\mathtt{rec}\, f.\, t)[\theta; \sigma]/f$. If c stands for an induction on an index object and c is applied to 0, we evaluate the the first branch of the induction term. If c is applied

$\boxed{t[\theta;\sigma] \Downarrow v}$ Term t under environments θ and σ evaluates to v

$$\frac{\sigma(x) = v}{x[\theta;\sigma] \Downarrow v} \qquad \frac{}{\langle\rangle[\theta;\sigma] \Downarrow \langle\rangle} \qquad \frac{t_1[\theta;\sigma] \Downarrow v_1 \quad t_2[\theta;\sigma] \Downarrow v_2}{\langle t_1, t_2\rangle[\theta;\sigma] \Downarrow \langle v_1, v_2\rangle} \qquad \frac{t[\theta;\sigma] \Downarrow \langle v_1, v_2\rangle \quad s[\theta;\sigma, v_1/x_1, v_2/x_2] \Downarrow v}{(\mathtt{split}\, t\, \mathtt{as}\, \langle x_1, x_2\rangle\, \mathtt{in}\, s)[\theta;\sigma] \Downarrow v}$$

$$\frac{t[\theta;\sigma] \Downarrow v}{(\mathtt{in}_i\, t)[\theta;\sigma] \Downarrow \mathtt{in}_i\, v} \qquad \frac{t[\theta;\sigma] \Downarrow \mathtt{in}_i\, v' \quad t_i[\theta;\sigma, v'/x_i] \Downarrow v}{(\mathtt{case}\, t\, \mathtt{of}\, \mathtt{in}_1\ x_1 \mapsto t_1 \mid \mathtt{in}_2\ x_2 \mapsto t_2)[\theta;\sigma] \Downarrow v} \qquad \frac{t[\theta;\sigma] \Downarrow v}{(t{:}T)[\theta;\sigma] \Downarrow v}$$

$$\frac{t[\theta;\sigma] \Downarrow v}{(\mathtt{pack}\,(M, t))[\theta;\sigma] \Downarrow \mathtt{pack}\,(M[\theta], v)} \qquad \frac{t[\theta;\sigma] \Downarrow \mathtt{pack}\,(N, v') \quad s[\theta, N/u;\sigma, v'/x] \Downarrow v}{(\mathtt{unpack}\, t\, \mathtt{as}\, (u, x)\, \mathtt{in}\, s)[\theta;\sigma] \Downarrow v}$$

$$\frac{}{\mathtt{refl}[\theta;\sigma] \Downarrow \mathtt{refl}} \qquad \frac{s[\theta;\sigma] \Downarrow \mathtt{refl} \quad \Delta \vdash \Theta \doteq \theta \searrow (\cdot \mid \theta') \quad t[\theta';\sigma] \Downarrow v}{(\mathtt{eq}\, s\, \mathtt{with}\, (\Delta.\Theta \mapsto t))[\theta;\sigma] \Downarrow v} \qquad \frac{t[\theta;\sigma] \Downarrow v}{(\mathtt{in}_l\, t)[\theta;\sigma] \Downarrow \mathtt{in}_l\, v}$$

$$\frac{}{(\lambda\, \vec{u}, x.\, t)[\theta;\sigma] \Downarrow (\lambda\, \vec{u}, x.\, t)[\theta;\sigma]} \qquad \frac{}{(\mathtt{rec}\, f.\, t)[\theta;\sigma] \Downarrow (\mathtt{rec}\, f.\, t)[\theta;\sigma]} \qquad \frac{t[\theta;\sigma] \Downarrow \mathtt{in}_l\, v}{(\mathtt{out}_l\, t)[\theta;\sigma] \Downarrow v}$$

$$\frac{}{(\mathtt{ind}\, t_0\, (u,\ f.\, t_s))[\theta;\sigma] \Downarrow (\mathtt{ind}\, t_0\, (u,\ f.\, t_s))[\theta;\sigma]} \qquad \frac{t[\theta;\sigma] \Downarrow c \quad s[\theta;\sigma] \Downarrow v \quad c \cdot \overrightarrow{M[\theta]}\, v \Downarrow w}{(t\, \vec{M}\, s)[\theta;\sigma] \Downarrow w}$$

$\boxed{c \cdot \vec{N}\, v \Downarrow w}$ Closure c applied to values $\vec{N}$ and v evaluates to w

$$\frac{t[\theta, \overrightarrow{N/u};\sigma, v/x] \Downarrow w}{(\lambda\, \vec{u}, x.\, t)[\theta;\sigma] \cdot \vec{N}\, v \Downarrow w} \qquad \frac{t[\theta;\sigma, (\mathtt{rec}\, f.\, t)[\theta;\sigma]/f] \Downarrow c \quad c \cdot \vec{N}\, v \Downarrow w}{(\mathtt{rec}\, f.\, t)[\theta;\sigma] \cdot \vec{N}\, (\mathtt{in}_\mu\, v) \Downarrow w}$$

$$\frac{t_0[\theta;\sigma] \Downarrow w}{(\mathtt{ind}\, t_0\, (u,\ f.\, t_s))[\theta;\sigma] \cdot 0\, \langle\rangle \Downarrow w} \qquad \frac{(\mathtt{ind}\, t_0\, (u,\ f.\, t_s))[\theta;\sigma] \cdot N\, \langle\rangle \Downarrow v \quad t_s[\theta, N/u;\sigma, v/f] \Downarrow w}{(\mathtt{ind}\, t_0\, (u,\ f.\, t_s))[\theta;\sigma] \cdot (\mathtt{suc}\, N)\, \langle\rangle \Downarrow w}$$

Figure 3.6: Big-step evaluation rules

to $\mathtt{suc}\, N$, we recursively compute the result of c applied to N before evaluating the body of the successor branch.

Lastly we explain evaluation of the equality elimination term $\mathtt{eq}\, s\, \mathtt{with}\, (\Delta.\Theta \mapsto t)$. We first evaluate the equality witness s under environments $\theta;\sigma$ to the value $\mathtt{refl}$. This ensures that θ respects the index equality $M = N$ witnessed by s. From type checking we know that $\Delta \vdash M[\Theta] = N[\Theta]$: the key is how we extend Θ at run-time to produce a new index environment θ' that is consistent with θ. As mentioned in Section 3.2.4, this relies on sound and complete index substitution matching to generate θ' such that $\cdot \vdash \theta' : \Delta$ and $\cdot \vdash \Theta[\theta'] = \theta$. We can then evaluate the body t under the new index environment θ' and the same term environment σ to produce a value v.

With this, we can prove subject reduction for Tores.

Theorem 15 (Subject Reduction)**.**

1. *If* $t[\theta;\sigma] \Downarrow v$ *where* $\Delta;\cdot;\Gamma \vdash t \Leftarrow T$ *or* $\Delta;\cdot;\Gamma \vdash t \Rightarrow T$*, and* $\vdash \theta : \Delta$ *and* $\sigma : \Gamma[\theta]$*, then* $v : T[\theta]$*.*
2. *If* $g[\theta;\sigma] \cdot \vec{N}\, v \Downarrow w$ *where* $\Delta;\cdot;\Gamma \vdash g \Leftarrow (\overrightarrow{u{:}U});\ S \to T$ *and* $\vdash \theta : \Delta$ *and* $\sigma : \Gamma[\theta]$ *and* $\vdash \vec{N} : (\overrightarrow{u{:}U})[\theta]$ *and* $v : S[\theta, \overrightarrow{N/u}]$*, then* $w : T[\theta, \overrightarrow{N/u}]$*.*

Proof. By mutual induction on the evaluation judgments. We include a few key cases here.

Case $\dfrac{s[\theta;\sigma] \Downarrow \mathtt{refl} \qquad \Delta' \vdash \Theta \doteq \theta \searrow (\cdot \mid \theta') \qquad t[\theta';\sigma] \Downarrow v}{(\mathtt{eq}\, s\, \mathtt{with}\, (\Delta'.\Theta \mapsto t))[\theta;\sigma] \Downarrow v}$

$\Delta; \cdot; \Gamma \vdash s \Rightarrow M = N$ and $\Delta'; \cdot; \Gamma[\Theta] \vdash t \Leftarrow T[\Theta]$	by inversion of typing
$\mathtt{refl} : (M = N)[\theta]$	by I.H.
$\mathtt{refl} : M[\theta] = N[\theta]$	by type substitution
$\vdash M[\theta] = N[\theta]$	by inversion of value typing
$\vdash \theta' : \Delta'$ and $\vdash \Theta[\theta'] = \theta$	by soundness of matching (Req. 5.2)
$T[\Theta][\theta'] = T[\Theta[\theta']] = T[\theta]$	by associativity of type substitution
$\Gamma[\Theta][\theta'] = \Gamma[\Theta[\theta']] = \Gamma[\theta]$	similarly for contexts
$\sigma : \Gamma[\theta]$	by assumption
$\sigma : \Gamma[\Theta][\theta']$	by context equality
$v : T[\Theta][\theta']$	by I.H.
$v : T[\theta]$	by type equality

Case $\dfrac{t[\theta;\sigma] \Downarrow c \qquad s[\theta;\sigma] \Downarrow v \qquad c \cdot \overrightarrow{M[\theta]}\, v \Downarrow w}{(t\, \vec{M}\, s)[\theta;\sigma] \Downarrow w}$

$\vdash \theta : \Delta$ and $\sigma : \Gamma[\theta]$	by assumption
$\Delta; \cdot; \Gamma \vdash t\, \vec{M}\, s \Rightarrow T[\overrightarrow{M/u}]$	by assumption
$\Delta; \cdot; \Gamma \vdash t \Rightarrow (\overrightarrow{u{:}U});\ S \to T$	
$\Delta \vdash \vec{M} : (\overrightarrow{u{:}U})$	
$\Delta; \cdot; \Gamma \vdash s \Leftarrow S[\overrightarrow{M/u}]$	by inversion of typing
$\vdash \overrightarrow{M[\theta]} : (\overrightarrow{u{:}U})[\theta]$	extending Req. 3.2 to index spines
$c : ((\overrightarrow{u{:}U});\ S \to T)[\theta]$	by I.H.
$v : S[\overrightarrow{M/u}][\theta]$	by I.H.
$v : S[\theta, \overrightarrow{M[\theta]/u}]$	by associativity of type substitution
$c = g[\theta';\sigma']$ where g is either $\lambda\, \vec{u}, x.\, t$, $\mathtt{rec}\, f.\, t$, or $\mathtt{ind}\, t_0\, (u,\ f.\, t_s)$	by grammar of closures
$\Delta'; \cdot; \Gamma' \vdash g \Leftarrow G$ and $g[\theta';\sigma'] : G[\theta']$	by closure typing
$G[\theta'] = ((\overrightarrow{u{:}U});\ S \to T)[\theta]$	by previous lines
$G = (\overrightarrow{u{:}U'});\ S' \to T'$ where $\overrightarrow{U'[\theta']} = \overrightarrow{U[\theta]}$	
and $S'[\theta', \overrightarrow{u/u}] = S[\theta, \overrightarrow{u/u}]$ and $T'[\theta', \overrightarrow{u/u}] = T[\theta, \overrightarrow{u/u}]$	by equality of types
$v : S'[\theta', \overrightarrow{M[\theta]/u}]$	by type equality
$\vdash \overrightarrow{M[\theta]} : (\overrightarrow{u{:}U'[\theta']})$	by type equality
$w : T'[\theta', \overrightarrow{M[\theta]/u}]$	by I.H. 2
$w : T[\theta, \overrightarrow{M[\theta]/u}]$	by type equality

Case $\dfrac{t[\theta, \overrightarrow{N/u}; \sigma, v/x] \Downarrow w}{(\lambda\, \vec{u}, x.\, t)[\theta; \sigma] \cdot \vec{N}\, v \Downarrow w}$

$\Delta; \cdot; \Gamma \vdash \lambda\, \vec{u}, x.\, t \Leftarrow (\overrightarrow{u{:}U});\ S \to T$ by assumption
$\Delta, \overrightarrow{u{:}U}; \cdot; \Gamma, x{:}S \vdash t \Leftarrow T$ by inversion of typing
$\vdash \theta : \Delta$ by assumption
$\vdash \theta, \overrightarrow{N/u} : \Delta, \overrightarrow{u{:}U}$ by substitution typing
$\sigma : \Gamma[\theta]$ by assumption
$\sigma : \Gamma[\theta, \overrightarrow{N/u}]$ by weakening since $\vec{u}$ do not occur in Γ
$\sigma, v/x : \Gamma[\theta, \overrightarrow{N/u}], x : S[\theta, \overrightarrow{N/u}]$ by environment typing
$\sigma, v/x : (\Gamma, x : S)[\theta, \overrightarrow{N/u}]$ by def. of context substitution
$w : T[\theta, \overrightarrow{N/u}]$ by I.H.

Case $\dfrac{t[\theta; \sigma, (\mathtt{rec}\, f.\, t)[\theta; \sigma]/f] \Downarrow c \qquad c \cdot \vec{N}\, v' \Downarrow w}{(\mathtt{rec}\, f.\, t)[\theta; \sigma] \cdot \vec{N}\, (\mathtt{in}_\mu\, v') \Downarrow w}$

$\Delta; \cdot; \Gamma \vdash \mathtt{rec}\, f.\, t \Leftarrow (\overrightarrow{u{:}U});\ (\mu X{:}K.\, \Lambda\, \overrightarrow{v}.\, S)\, \vec{u} \to T$ by assumption
$\Delta; X{:}K; \Gamma, f{:}((\overrightarrow{u{:}U});\ X\vec{u} \to T) \vdash t \Leftarrow (\overrightarrow{u{:}U});\ S[\overrightarrow{u/v}] \to T$ by inversion of typing
$\Delta; \cdot; \Gamma, f{:}((\overrightarrow{u{:}U});\ (\mu X{:}K.\, \Lambda\, \overrightarrow{v}.\, S)\ \vec{u} \to T) \vdash t \Leftarrow (\overrightarrow{u{:}U});\ S[\overrightarrow{u/v}; \mu X{:}K.\, \Lambda\, \overrightarrow{v}.\, S/X] \to T$
by substitution property of type variables
$\mathtt{in}_\mu\, v' : ((\mu X{:}K.\, \Lambda\, \overrightarrow{v}.\, S)\ \vec{u}))[\theta, \overrightarrow{N/u}]$ by assumption
$v' : S[\theta, \overrightarrow{N/u};\ \mu X{:}K.\, \Lambda\, \overrightarrow{v}.\, S/X]$ by value typing
$\vdash \theta : \Delta$ by assumption
$\sigma, (\mathtt{rec}\, f.\, t)[\theta; \sigma]/f : \Gamma[\theta], f{:}((\overrightarrow{u{:}U});\ (\mu X{:}K.\, \Lambda\, \overrightarrow{v}.\, S)\ \vec{u} \to T)[\theta]$ by environment typing
$\sigma, (\mathtt{rec}\, f.\, t)[\theta; \sigma]/f : (\Gamma, f{:}((\overrightarrow{u{:}U});\ (\mu X{:}K.\, \Lambda\, \overrightarrow{v}.\, S)\ \vec{u} \to T))[\theta]$ by def. of context substitution
$c : ((\overrightarrow{u{:}U});\ S[\overrightarrow{u/v}; \mu X{:}K.\, \Lambda\, \overrightarrow{v}.\, S/X] \to T)[\theta]$ by I.H.
$c = g[\theta'; \sigma']$ where $\Delta'; \cdot; \Gamma' \vdash g \Leftarrow G$ and $g[\theta'; \sigma'] : G[\theta']$
and $\vdash \theta' : \Delta'$ and $\sigma' : \Gamma'[\theta']$ by closure grammar and typing
$G[\theta'] = ((\overrightarrow{u{:}U});\ S \to T)[\theta]$ by previous lines
$G = (\overrightarrow{u{:}U'});\ S' \to T'$ where $\overrightarrow{U'[\theta']} = \overrightarrow{U[\theta]}$
and $S'[\theta', \overrightarrow{u/u}] = S[\theta, \overrightarrow{u/u}\ ;\ \mu X{:}K.\, \Lambda\, \overrightarrow{v}.\, S/X]$ and $T'[\theta', \overrightarrow{u/u}] = T[\theta, \overrightarrow{u/u}]$ by type equality
$\Delta'; \cdot; \Gamma' \vdash g \Leftarrow (\overrightarrow{u{:}U'});\ S' \to T'$ by previous lines
$\vdash \vec{N} : (\overrightarrow{u : U[\theta]})$ by assumption
$\vdash \vec{N} : (\overrightarrow{u : U'[\theta']})$ by type equality
$v' : S'[\theta', \overrightarrow{N/u}]$ by type equality
$w : T'[\theta', \overrightarrow{N/u}]$ by I.H.
$w : T[\theta, \overrightarrow{N/u}]$ by type equality
□

3.4 Termination Proof

We now describe our main technical result: termination of evaluation. Our proof of termination uses the technique of logical relations, following the style of Tait [1967] and Girard [1972]. As in Section 2.3.1, we use a unary logical relation, also called a logical predicate, where we interpret constructs at each level of the language as objects in a semantic model. Note that in this section we keep the notation lighter by using the same interpretation brackets $[\![-]\!]$ for every level of the interpretation (index types, kinds, types, etc.).

3.4.1 Interpretation of Index Language

We start with the interpretations for index types and spines. In general, our index language may be dependently typed, as it is if we choose Contextual LF for example. Hence our interpretation for index types U must take into account an environment θ containing instantiations for index variables u. An index environment θ for an index context Δ is simply a grounding substitution $\vdash \theta : \Delta$.

Definition 16 (Interpretation of index types $[\![U]\!]$)**.** $[\![U]\!](\theta) = \{M \mid \cdot \vdash M : U[\theta]\}$

The interpretation of an index type U under environment θ is the set of closed terms of type $U[\theta]$. Unlike for term-level types, we do not restrict our interpretation to contain only normal forms. This is because for index languages, any reducible expressions are reduced either during equality checking or, in the case of Contextual LF, during hereditary substitutions [Nanevski et al., 2008]. We leave those details abstract, requiring only the properties in Section 3.2 such as decidable equality of index terms and type-preserving substitution. The interpretation lifts to index spines $(\overrightarrow{u{:}U})$ in a straight-forward manner.

Definition 17 (Interpretation of index spines)**.**

$$\begin{array}{lcl} [\![(\cdot)]\!](\theta) & = & \{\cdot\} \\ [\![(u_0{:}U_0, \overrightarrow{u{:}U})]\!](\theta) & = & \{M_0, \vec{M} \mid M_0 \in [\![U_0]\!](\theta), \vec{M} \in [\![(\overrightarrow{u{:}U})]\!](\theta, M_0/u_0)\} \end{array}$$

With these definitions, the following lemma follows from the substitution principles of index terms (Req. 3).

Lemma 18 (Interpretation of index substitution)**.**

18.1. If $\Delta \vdash M : U$ *and* $\vdash \theta : \Delta$ *then* $M[\theta] \in [\![U]\!](\theta)$.

18.2. If $\Delta \vdash \vec{M} : (\overrightarrow{u{:}U})$ *and* $\vdash \theta : \Delta$ *then* $\overrightarrow{M[\theta]} \in [\![(\overrightarrow{u{:}U})]\!](\theta)$.

3.4.2 Lattice Interpretation of Kinds

We now describe the lattice structure that underlies the interpretation of kinds in our language. The idea is that types are interpreted as sets of term-level values and type constructors as functions taking indices to sets of values. We call the set of all term-level values Ω and write its power set as $\mathcal{P}(\Omega)$. The interpretation is defined inductively on the structure of kinds.

Definition 19 (Interpretation of kinds $[\![K]\!]$)**.**

$$\begin{aligned}[\![*]\!](\theta) &= \mathcal{P}(\Omega)\\ [\![\Pi u{:}U.\, K]\!](\theta) &= \{\mathcal{C} \mid \forall M \in [\![U]\!](\theta).\, \mathcal{C}(M) \in [\![K]\!](\theta, M/u)\}\end{aligned}$$

A key observation in our metatheory is that each $[\![K]\!](\theta)$ forms a *complete lattice*. Recall that a lattice is a partially ordered set with additional meet $\bigwedge$ and join $\bigvee$ operations. A lattice is complete if every subset has a meet (greatest lower bound) and a join (least upper bound). In our case, $[\![*]\!](\theta) = \mathcal{P}(\Omega)$ is a complete lattice under the subset ordering, with meet and join given by intersection and union respectively. Further, we can induce a complete lattice structure on interpretations of kinds $K = \Pi u{:}U.\, K'$ by lifting the lattice operations pointwise. Precisely, we define

$$\mathcal{A} \leq_{[\![K]\!](\theta)} \mathcal{B} \quad \text{iff} \quad \forall M \in [\![U]\!](\theta).\, \mathcal{A}(M) \leq_{[\![K']\!](\theta, M/u)} \mathcal{B}(M).$$

The meet and join operations can similarly be lifted pointwise.

This structure is important because it allows us to define pre-fixed points for operators on the lattice, which is central to our interpretation of recursive types. The following definition lifts the set-theoretic fixed point operator from Section 2.3 to the lattice $\mathcal{L}$ of set-valued functions. Here we rely on the existence of arbitrary meets, as we take the meet over an impredicatively defined subset of $\mathcal{L}$.

Definition 20 (Mendler-style pre-fixed point)**.** Suppose $\mathcal{L}$ is a complete lattice and $\mathcal{F} : \mathcal{L} \to \mathcal{L}$. Define $\boldsymbol{\mu}_{\mathcal{L}} : (\mathcal{L} \to \mathcal{L}) \to \mathcal{L}$ by

$$\boldsymbol{\mu}_{\mathcal{L}}\mathcal{F} = \bigwedge\{\mathcal{C} \in \mathcal{L} \mid \forall \mathcal{X} \in \mathcal{L}.\, \mathcal{X} \leq_{\mathcal{L}} \mathcal{C} \implies \mathcal{F}(\mathcal{X}) \leq_{\mathcal{L}} \mathcal{C}\}.$$

Note that we will mostly omit the subscript denoting the underlying lattice $\mathcal{L}$ of the order $\leq$ and pre-fixed point $\boldsymbol{\mu}$.

As discussed in Section 2.3.1, a usual treatment of recursive types would define the least pre-fixed point of a monotone operator as $\bigwedge\{\mathcal{C} \in \mathcal{L} \mid \mathcal{F}(\mathcal{C}) \leq \mathcal{C}\}$, using the Knaster-Tarski theorem. However, our unconventional definition more closely models Mendler-style recursion and does not require $\mathcal{F}$ to be monotone (thereby avoiding a positivity restriction on recursive types).

3.4.3 Interpretation of Types

In order to interpret the types of our language, it is helpful to define semantic versions of some syntactic constructs. We first define a semantic form of our indexed function type $(\overrightarrow{u{:}U}); T_1 \to T_2$, which helps us formulate the interaction of function types with fixed points and recursion.

Definition 21 (Semantic function space)**.** For a spine interpretation $\vec{\mathcal{U}}$ and functions $\mathcal{A}, \mathcal{B} : \vec{\mathcal{U}} \to \mathcal{P}(\Omega)$, define

$$\vec{\mathcal{U}},\, \mathcal{A} \to \mathcal{B} = \{c \mid \forall \vec{M} \in \vec{\mathcal{U}}.\, \forall v \in \mathcal{A}(\vec{M}).\, c \cdot \vec{M}\, v \Downarrow w \in \mathcal{B}(\vec{M})\}.$$

It will also be convenient to lift term-level `in` tags to the level of sets and functions in the lattice $[\![K]\!](\theta)$. We define the lifted tags $\texttt{in}^* : [\![K]\!](\theta) \to [\![K]\!](\theta)$ inductively on K. If $\mathcal{V} \in [\![*]\!](\theta) = \mathcal{P}(\Omega)$ then

$\mathtt{in}^*\,\mathcal{V} = \mathtt{in}\,\mathcal{V} = \{\mathtt{in}\,v \mid v \in \mathcal{V}\}$. If $\mathcal{C} \in [\![\Pi u{:}U.\,K']\!](\theta)$ then $(\mathtt{in}^*\,\mathcal{C})(M) = \mathtt{in}^*\,(\mathcal{C}(M))$ for all $M \in [\![U]\!](\theta)$. Essentially, the $\mathtt{in}^*$ function attaches a tag to every element in the set produced after the index arguments are received.

Finally, we need to define the interpretation of type variable contexts Ξ. These describe semantic environments η mapping each type variable to an object in its respective kind interpretation. Such environments are necessary to interpret type expressions with free type variables.

Definition 22 (Interpretation of type variable contexts $[\![\Xi]\!]$)**.**

$$\begin{array}{lcl} [\![\cdot]\!](\theta) & = & \{\cdot\} \\ [\![\Xi, X{:}K]\!](\theta) & = & \{\eta, \mathcal{X}/X \mid \eta \in [\![\Xi]\!](\theta), \mathcal{X} \in [\![K]\!](\theta)\} \end{array}$$

This definition corresponds the $\mathcal{D}$ function in Section 2.3.1. In this setting, however, we constrain type variable mappings to their respective kind interpretations, unlike the simply typed case where mappings can be to arbitrary sets of values.

We are now able to define the interpretation of types T under environments θ and η. This is done inductively on the structure of T.

Definition 23 (Interpretation of types and constructors)**.**

$$\begin{array}{lcl}
[\![1]\!](\theta;\eta) & = & \{\langle\rangle\} \\
[\![T_1 \times T_2]\!](\theta;\eta) & = & \{\langle v_1, v_2\rangle \mid v_1 \in [\![T_1]\!](\theta;\eta), v_2 \in [\![T_2]\!](\theta;\eta)\} \\
[\![T_1 + T_2]\!](\theta;\eta) & = & \mathtt{in}_1\,[\![T_1]\!](\theta;\eta) \bigcup \mathtt{in}_2\,[\![T_2]\!](\theta;\eta) \\
[\![(\overrightarrow{u{:}U});\, T_1 \to T_2]\!](\theta;\eta) & = & [\![(\overrightarrow{u{:}U})]\!](\theta),\ \mathcal{T}_1 \to \mathcal{T}_2 \\
 & & \text{where } \mathcal{T}_i(\vec{M}) = [\![T_i]\!](\theta, \overrightarrow{M/u};\eta) \text{ for } i \in \{1,2\} \\
[\![\Sigma u{:}U.\,T]\!](\theta;\eta) & = & \{\mathtt{pack}\,(M, v) \mid M \in [\![U]\!](\theta), v \in [\![T]\!](\theta, M/u;\eta)\} \\
[\![T\,M]\!](\theta;\eta) & = & [\![T]\!](\theta;\eta)(M[\theta]) \\
[\![M = N]\!](\theta;\eta) & = & \{\mathtt{refl} \mid \vdash M[\theta] = N[\theta]\} \\
[\![X]\!](\theta;\eta) & = & \eta(X) \\
[\![\Lambda\, u.\,T]\!](\theta;\eta) & = & (M \mapsto [\![T]\!](\theta, M/u;\eta)) \\
[\![\mu X{:}K.\,T]\!](\theta;\eta) & = & \boldsymbol{\mu}_{[\![K]\!](\theta)}(\mathcal{X} \mapsto \mathtt{in}^*_\mu\,([\![T]\!](\theta;\eta, \mathcal{X}/X))) \\
[\![\mathtt{Rec}_K\,(0 \mapsto T_z \mid \mathsf{suc}\,u,\ X \mapsto T_s)]\!](\theta;\eta) & = & \mathbf{Rec}_{[\![K]\!](\theta)}\,(\mathtt{in}^*_0\,[\![T_z]\!](\theta;\eta)) \\
 & & \quad (N \mapsto \mathcal{X} \mapsto \mathtt{in}^*_{\mathsf{suc}}\,[\![T_s]\!](\theta, N/u;\eta, \mathcal{X}/X))
\end{array}$$

where

$$\begin{array}{llllll}
\mathbf{Rec}_{\mathcal{L}}: & \mathcal{L} \to & (\mathbb{N} \to \mathcal{L} \to \mathcal{L}) \to & \mathbb{N} & \to & \mathcal{L} \\
\mathbf{Rec}_{\mathcal{L}} & \mathcal{C} & \mathcal{F} & 0 & = & \mathcal{C} \\
\mathbf{Rec}_{\mathcal{L}} & \mathcal{C} & \mathcal{F} & (\mathsf{suc}\,N) & = & \mathcal{F}\,N\,(\mathbf{Rec}_{\mathcal{L}}\,\mathcal{C}\,\mathcal{F}\,N)
\end{array}$$

The interpretation of the indexed function type $[\![(\overrightarrow{u{:}U});\, T_1 \to T_2]\!](\theta;\eta)$ contains closures which, when applied to values in the appropriate input sets, evaluate to values in the appropriate output set. The interpretation of the equality type $[\![M = N]\!](\theta;\eta)$ is the set $\{\mathtt{refl}\}$ if $\vdash M[\theta] = N[\theta]$ and the empty set

otherwise. The interpretation of a recursive type is the pre-fixed point of the function obtained from the underlying type expression. Finally, interpretation of a stratified type built from `Rec` relies on an analogous semantic operator **Rec**. It is defined by primitive recursion on the index argument, returning the first argument in the base case and calling itself recursively in the step case. Note that the definition of **Rec** is specific to the index type it recurses over. We only use the index language of natural numbers here, so the appropriate set of index values is $[\![\mathsf{nat}]\!] = \mathbb{N}$.

The last form of interpretation we need is for typing contexts Γ, describing well-formed term-level environments σ.

Definition 24 (Interpretation of typing contexts)**.**

$$[\![\cdot]\!](\theta;\eta) = \{\cdot\}$$
$$[\![\Gamma, x{:}T]\!](\theta;\eta) = \{\sigma, v/x \mid \sigma \in [\![\Gamma]\!](\theta;\eta), v \in [\![T]\!](\theta;\eta)\}$$

3.4.4 Proof

We now sketch our proof using some key lemmas. The following two lemmas concern the pre-fixed point operator $\boldsymbol{\mu}$ and generalize Lemmas 4 and 5 to the setting of a complete lattice. These are again key for reasoning about recursive types and Mendler-style recursion.

Lemma 25 (Soundness of pre-fixed point)**.** *Suppose $\mathcal{L}$ is a complete lattice, $\mathcal{F} : \mathcal{L} \to \mathcal{L}$ and $\boldsymbol{\mu}$ is as in Def. 20. Then $\mathcal{F}(\boldsymbol{\mu}\mathcal{F}) \leq \boldsymbol{\mu}\mathcal{F}$.*

Proof. Recall that $\boldsymbol{\mu}\mathcal{F} = \bigwedge \mathcal{S}$ where $\mathcal{S} = \{\mathcal{C} \in \mathcal{L} \mid \forall \mathcal{X} \in \mathcal{L}.\ \mathcal{X} \leq \mathcal{C} \implies \mathcal{F}(\mathcal{X}) \leq \mathcal{C}\}$. To show $\mathcal{F}(\boldsymbol{\mu}\mathcal{F}) \leq \bigwedge \mathcal{S}$, it suffices to show $\mathcal{F}(\boldsymbol{\mu}\mathcal{F}) \leq \mathcal{C}$ for every $\mathcal{C} \in \mathcal{S}$. By definition of the meet, $\boldsymbol{\mu}\mathcal{F} \leq \mathcal{C}$ for every $\mathcal{C} \in \mathcal{S}$. But by definition of $\mathcal{S}$, this implies that $\mathcal{F}(\boldsymbol{\mu}\mathcal{F}) \leq \mathcal{C}$ as required. (If this argument appears to be circular, that's because it is! It cleverly exploits our impredicative definition of $\boldsymbol{\mu}$.) □

Lemma 26 (Function space from a pre-fixed point)**.** *Let $\mathcal{L} = \vec{\mathcal{U}} \to \mathcal{P}(\Omega)$ and $\mathcal{B} \in \mathcal{L}$ and $\mathcal{F} : \mathcal{L} \to \mathcal{L}$. If $\forall \mathcal{X} \in \mathcal{L}.\ c \in \vec{\mathcal{U}},\ \mathcal{X} \to \mathcal{B} \implies c \in \vec{\mathcal{U}},\ \mathcal{F}\mathcal{X} \to \mathcal{B}$, then $c \in \vec{\mathcal{U}},\ \boldsymbol{\mu}\mathcal{F} \to \mathcal{B}$.*

Proof. We will reframe the lemma statement using a new piece of notation. For a closure $c \in \Omega$ and $\mathcal{B} \in \vec{\mathcal{U}} \to \mathcal{P}(\Omega) = \mathcal{L}$, define $\mathcal{E}_c(\mathcal{B}) \in \mathcal{L}$ by $\mathcal{E}_c(\mathcal{B})(\vec{M}) = \{v \in \Omega \mid c \cdot \vec{M}\ v \Downarrow w \in \mathcal{B}(\vec{M})\}$. One can see that $c \in \vec{\mathcal{U}},\ \mathcal{A} \to \mathcal{B} \iff \mathcal{A} \leq \mathcal{E}_c(\mathcal{B})$ (using the ordering on $\mathcal{L}$). We can now rewrite the lemma as the following: if $\forall \mathcal{X} \in \mathcal{L}.\ \mathcal{X} \leq \mathcal{E}_c(\mathcal{B}) \implies \mathcal{F}\mathcal{X} \leq \mathcal{E}_c(\mathcal{B})$ then $\boldsymbol{\mu}\mathcal{F} \leq \mathcal{E}_c(\mathcal{B})$.

To prove this, assume the premise. Recall that $\boldsymbol{\mu}\mathcal{F} = \bigwedge \mathcal{S}$ where $\mathcal{S} = \{\mathcal{C} \in \mathcal{L} \mid \forall \mathcal{X} \in \mathcal{L}.\ \mathcal{X} \leq \mathcal{C} \implies \mathcal{F}(\mathcal{X}) \leq \mathcal{C}\}$. By definition of the meet, $\boldsymbol{\mu}\mathcal{F} \leq \mathcal{C}$ for every $\mathcal{C} \in \mathcal{S}$. Therefore it suffices to show that there is just one $\mathcal{C} \in \mathcal{S}$ for which $\mathcal{C} \leq \mathcal{E}_c(\mathcal{B})$. However, our assumption is exactly that $\mathcal{E}_c(\mathcal{B}) \in \mathcal{S}$, and clearly $\mathcal{E}_c(\mathcal{B}) \leq \mathcal{E}_c(\mathcal{B})$, so we are done. (This proof again makes use of impredicativity in the definition of $\boldsymbol{\mu}$.) □

Another key result we rely on is that type-level substitutions associate with our semantic interpretations. Note that single index (and spine) substitutions on types are handled as special cases of the result for simultaneous index substitutions. We omit the definitions of type substitutions for brevity.

Lemma 27 (Type-level substitution associates with interpretation)**.**
Suppose $\Delta;\Xi \vdash T \Leftarrow K$ *or* $\Delta;\Xi \vdash T \Rightarrow K$*, and* $\vdash \theta : \Delta'$ *and* $\eta \in [\![\Xi']\!](\theta)$*.*

1. *If* $\Delta' \vdash \Theta : \Delta$ *and* $\Xi' = \Xi[\Theta]$ *then* $[\![\Xi']\!](\theta) = [\![\Xi]\!](\Theta[\theta])$ *and* $[\![T[\Theta]]\!](\theta;\eta) = [\![T]\!](\Theta[\theta];\eta)$*.*
2. *If* $\Delta = \Delta'$ *and* $\Xi = \Xi', X{:}K$ *and* $\Delta';\Xi' \vdash S \Leftarrow K$ *or* $\Delta';\Xi' \vdash S \Rightarrow K$*, then* $[\![T[S/X]]\!](\theta;\eta) = [\![T]\!](\theta;\eta, [\![S]\!](\theta;\eta)/X)$*.*

Proof. By induction on the structure of T. □

The next two lemmas concern recursive types and terms respectively.

Lemma 28 (Recursive type contains unfolding)**.**
Let $R = \mu X{:}K.\,\Lambda\,\vec{u}.\,S$ *where* $K = \Pi\overrightarrow{u{:}U}.\,*$ *and* $\Delta;\Xi \vdash R \Rightarrow K$*, and* $\Delta \vdash \vec{M} : (\overrightarrow{u{:}U})$ *and* $\vdash \theta : \Delta$ *and* $\eta \in [\![\Xi]\!](\theta)$*. Then* $\mathit{in}_\mu\,[\![S[\overrightarrow{M/u}; R/X]]\!](\theta;\eta) \subseteq [\![R\,\vec{M}]\!](\theta;\eta)$*.*

Proof. Let $\mathcal{L} = [\![K]\!](\theta)$.
Define $\mathcal{F} : \mathcal{L} \to \mathcal{L}$ by $\mathcal{F}(\mathcal{X}) = \vec{N} \mapsto \mathtt{in}_\mu\,[\![S]\!](\theta, \overrightarrow{N/u}; \eta, \mathcal{X}/X)$.
Then $[\![R]\!](\theta;\eta)$
$= \boldsymbol{\mu}(\mathcal{X} \mapsto \mathtt{in}^*_\mu\,[\![\Lambda\,\vec{u}.\,S]\!](\theta;\eta,\mathcal{X}/X))$ by $[\![\mu X.\,T]\!]$ def.
$= \boldsymbol{\mu}(\mathcal{X} \mapsto \vec{N} \mapsto \mathtt{in}_\mu\,[\![S]\!](\theta, \overrightarrow{N/u}; \eta, \mathcal{X}/X))$ by $[\![\Lambda\,\vec{u}.\,T]\!]$ def.
$= \boldsymbol{\mu}\mathcal{F}$ by $\mathcal{F}$ def.
$\mathcal{F}([\![R]\!](\theta;\eta)) \leq_{\mathcal{L}} [\![R]\!](\theta;\eta)$ by Lemma 25
Now $\mathtt{in}_\mu\,[\![S[\overrightarrow{M/u}; R/X]]\!](\theta;\eta)$
$= \mathtt{in}_\mu\,[\![S]\!]((\mathsf{id}_\Delta, \overrightarrow{M/u})[\theta]; \eta, [\![R]\!](\theta;\eta)/X)$ by Lemma 27
$= \mathtt{in}_\mu\,[\![S]\!](\theta, \overrightarrow{M[\theta]/u}; \eta, [\![R]\!](\theta;\eta)/X)$ by Def. 8
$= \mathcal{F}([\![R]\!](\theta;\eta))(\overrightarrow{M[\theta]})$ by $\mathcal{F}$ def.
$\subseteq [\![R]\!](\theta;\eta)(\overrightarrow{M[\theta]})$ since $\mathcal{F}([\![R]\!](\theta;\eta)) \leq_{\mathcal{L}} [\![R]\!](\theta;\eta)$
$= [\![R\,\vec{M}]\!](\theta;\eta)$ by $[\![R\,\vec{M}]\!]$ def.
□

Lemma 29 (Backward closure)**.** *Let* t *be a term,* θ *and* σ *environments, and* $\mathcal{A}, \mathcal{B} \in \vec{\mathcal{U}} \to \mathcal{P}(\Omega)$*.*
If $t[\theta;\sigma, (\mathtt{rec}\,f.\,t)[\theta;\sigma]/f] \Downarrow c' \in \vec{\mathcal{U}},\ \mathcal{A} \to \mathcal{B}$*, then* $(\mathtt{rec}\,f.\,t)[\theta;\sigma] \in \vec{\mathcal{U}},\ \mathit{in}^*_\mu\,\mathcal{A} \to \mathcal{B}$*.*

Proof. Let $c = (\mathtt{rec}\,f.\,t)[\theta;\sigma]$.
Suppose $\vec{M} \in \vec{\mathcal{U}}$ and $v' \in (\mathtt{in}^*_\mu\,\mathcal{A})(\vec{M})$.
$v \in \mathtt{in}_\mu\,\mathcal{A}(\vec{M})$ by $\mathtt{in}^*$ def.
$v' = \mathtt{in}_\mu\,v$ where $v \in \mathcal{A}(\vec{M})$
$t[\theta;\sigma,c/f] \Downarrow c' \in \vec{\mathcal{U}},\ \mathcal{A} \to \mathcal{B}$ assumption of lemma
$c' \cdot \vec{M}\,v \Downarrow w \in \mathcal{B}(\vec{M})$ by $\vec{\mathcal{U}},\ \mathcal{A} \to \mathcal{B}$ def.
$c \cdot \vec{M}\,(\mathtt{in}_\mu\,v) \Downarrow w \in \mathcal{B}(\vec{M})$ by `e-app-rec`
$c \cdot \vec{M}\,v' \Downarrow w \in \mathcal{B}(\vec{M})$ since $v' = \mathtt{in}_\mu\,v$

$c \in \vec{\mathcal{U}},\ \mathtt{in}^*_\mu\, \mathcal{A} \to \mathcal{B}$ by $\vec{\mathcal{U}},\ \mathcal{A} \to \mathcal{B}$ def.

□

Our final lemma concerns the semantic equivalence of an applied stratified type with its unfolding. Note that here we only state and prove the lemma for an index language of natural numbers. For a different index language, one would need to reverify this lemma for the corresponding stratified type. This should be straight-forward once the semantic **Rec** operator is chosen to reflect the inductive structure of the index language.

Lemma 30 (Stratified types equivalent to unfolding)**.**
Let $T_{\mathtt{Rec}} \equiv \mathtt{Rec}_K\,(0 \mapsto T_z \mid \mathsf{suc}\, n,\, X \mapsto T_s)$ *where* $K = \Pi n\colon \mathsf{nat}\,.\,\Pi \overrightarrow{u{:}U}.\,*$ *and* $\Delta;\Xi \vdash T_{\mathtt{Rec}} \Rightarrow K$*, and* $\Delta \vdash \vec{M} : (\overrightarrow{u{:}U})$ *and* $\Delta \vdash N : \mathsf{nat}$ *and* $\vdash \theta : \Delta$ *and* $\eta \in [\![\Xi]\!](\theta)$*. Then*

1. $[\![T_{\mathtt{Rec}}\, 0\, \vec{M}]\!](\theta;\eta) = \mathit{in}_0\,([\![T_z\, \vec{M}]\!](\theta;\eta))$ *and*
2. $[\![T_{\mathtt{Rec}}\,(\mathsf{suc}\, N)\, \vec{M}]\!](\theta;\eta) = \mathit{in}_{\mathsf{suc}}\,([\![T_s[N/n; (T_{\mathtt{Rec}}\, N)/X]\, \vec{M}]\!](\theta;\eta))$.

Proof. Let $\mathcal{C} = \mathtt{in}^*_0\,[\![T_z]\!](\theta;\eta)$ and $\mathcal{F} = (N \mapsto \mathcal{X} \mapsto \mathtt{in}^*_{\mathsf{suc}}\,[\![T_s]\!](\theta, N/n; \eta, \mathcal{X}/X))$.

1. $[\![T_{\mathtt{Rec}}\, 0\, \vec{M}]\!](\theta;\eta)$
   $= [\![T_{\mathtt{Rec}}]\!](\theta;\eta)(0)(\overrightarrow{M[\theta]})$ by $[\![T\, \vec{M}]\!]$ def.
   $= \mathbf{Rec}\ \mathcal{C}\ \mathcal{F}\ 0\ \overrightarrow{M[\theta]}$ by $[\![T_{\mathtt{Rec}}]\!]$ def.
   $= \mathcal{C}\ \overrightarrow{M[\theta]}$ by **Rec** def
   $= (\mathtt{in}^*_0\,[\![T_z]\!](\theta;\eta))(\overrightarrow{M[\theta]})$ by $\mathcal{C}$ def.
   $= \mathtt{in}_0\,([\![T_z]\!](\theta;\eta)(\overrightarrow{M[\theta]}))$ by $\mathtt{in}^*$ def.
   $= \mathtt{in}_0\,([\![T_z\, \vec{M}]\!](\theta;\eta))$ by $[\![T\, \vec{M}]\!]$ def.

2. $[\![T_{\mathtt{Rec}}\,(\mathsf{suc}\, N)\, \vec{M}]\!](\theta;\eta)$
   $= [\![T_{\mathtt{Rec}}]\!](\theta;\eta)(\mathsf{suc}\, N[\theta])(\overrightarrow{M[\theta]})$ by $[\![T\, \vec{M}]\!]$ def.
   $= \mathbf{Rec}\ \mathcal{C}\ \mathcal{F}\ (\mathsf{suc}\, N[\theta])\ \overrightarrow{M[\theta]}$ by $[\![T_{\mathtt{Rec}}]\!]$ def.
   $= \mathcal{F}\, N[\theta]\ (\mathbf{Rec}\ \mathcal{C}\ \mathcal{F}\, N[\theta])\ \overrightarrow{M[\theta]}$ by **Rec** def.
   $= (\mathtt{in}^*_{\mathsf{suc}}\,[\![T_s]\!](\theta, N[\theta]/n; \eta, (\mathbf{Rec}\ \mathcal{C}\ \mathcal{F}\, N[\theta])/X))(\overrightarrow{M[\theta]})$ by $\mathcal{F}$ def.
   $= \mathtt{in}_{\mathsf{suc}}\,([\![T_s]\!](\theta, N[\theta]/n; \eta, (\mathbf{Rec}\ \mathcal{C}\ \mathcal{F}\, N[\theta])/X)(\overrightarrow{M[\theta]}))$ by $\mathtt{in}^*$ def.
   $= \mathtt{in}_{\mathsf{suc}}\,([\![T_s]\!]((\mathsf{id}_\Delta, N/n)[\theta]; \eta, [\![T_{\mathtt{Rec}}\, N]\!](\theta;\eta)/X)(\overrightarrow{M[\theta]}))$ by Def. 8 and $[\![T_{\mathtt{Rec}}]\!]$ def.
   $= \mathtt{in}_{\mathsf{suc}}\,([\![T_s[N/n; (T_{\mathtt{Rec}}\, N)/X]]\!](\theta;\eta)(\overrightarrow{M[\theta]}))$ by Lemma 27
   $= \mathtt{in}_{\mathsf{suc}}\,([\![T_s[N/n; (T_{\mathtt{Rec}}\, N)/X]\, \vec{M}]\!](\theta;\eta))$ by $[\![T\, \vec{M}]\!]$ def.

□

Finally we state and prove the main termination theorem.

Theorem 31 (Termination of evaluation)**.** *If* $\Delta; \Xi; \Gamma \vdash t \Leftarrow T$ *or* $\Delta; \Xi; \Gamma \vdash t \Rightarrow T$*, and* $\vdash \theta : \Delta$ *and* $\eta \in [\![\Xi]\!](\theta)$ *and* $\sigma \in [\![\Gamma]\!](\theta; \eta)$*, then* $t[\theta; \sigma] \Downarrow v$ *for some* $v \in [\![T]\!](\theta; \eta)$*.*

Proof. The proof is by induction on the typing derivation. Technically this is a mutual induction on the dual judgments of type checking and synthesis. In each case we introduce the assumptions $\vdash \theta : \Delta$ and $\eta \in [\![\Xi]\!](\theta)$ and $\sigma \in [\![\Gamma]\!](\theta; \eta)$, where Δ, Ξ and Γ appear in the conclusion of the relevant typing rule. We will slightly abuse notation to introduce an existentially quantified variable in the judgment $t[\theta; \sigma] \Downarrow v \in \mathcal{V}$, to mean that $\exists v.\, t[\theta; \sigma] \Downarrow v \wedge v \in \mathcal{V}$. Note that the cases involving stratified types and induction over indices are specific to the particular index language (and assume an induction principle over closed index types). The rest of the proof is generic, only assuming the properties in Section 3.2.

Case:

$$\frac{}{\Delta; \Xi; \Gamma \vdash \langle\rangle \Leftarrow 1}\ \texttt{t-unit}$$

$\langle\rangle[\theta; \sigma] \Downarrow \langle\rangle$ by `e-unit`
$\langle\rangle \in [\![1]\!](\theta; \eta)$ by $[\![1]\!]$ def.

Case:

$$\frac{x{:}T \in \Gamma}{\Delta; \Xi; \Gamma \vdash x \Rightarrow T}\ \texttt{t-var}$$

$\sigma \in [\![\Gamma]\!](\theta; \eta)$ assumption of Thm 31
$x{:}T \in \Gamma$ premise of `t-var`
$\sigma(x) = v \in [\![T]\!](\theta; \eta)$ by Def. 24
$x[\theta; \sigma] \Downarrow v$ by `e-var`

Case:

$$\frac{\Delta, \overrightarrow{u{:}U}; \Xi; \Gamma, x{:}R \vdash s \Leftarrow S}{\Delta; \Xi; \Gamma \vdash \lambda\, \vec{u}, x.\, s \Leftarrow (\overrightarrow{u{:}U});\ R \to S}\ \texttt{t-lam}$$

Let c be the closure $(\lambda\, \vec{u}, x.\, s)[\theta; \sigma]$.
$(\lambda\, \vec{u}, x.\, s)[\theta; \sigma] \Downarrow c$ by `e-lam`
Suffices to show $c \in [\![(\overrightarrow{u{:}U});\ R \to S]\!](\theta; \eta)$, i.e.
$\forall \vec{M} \in [\![(\overrightarrow{u{:}U})]\!](\theta).\ \forall v \in [\![R]\!](\theta'; \eta).\ c \cdot \vec{M}\, v \Downarrow w \in [\![S]\!](\theta'; \eta)$, where $\theta' = \theta, \overrightarrow{M/u}$.
Suppose $\vec{M} \in [\![(\overrightarrow{u{:}U})]\!](\theta)$ and $v \in [\![R]\!](\theta'; \eta)$ where $\theta' = \theta, \overrightarrow{M/u}$.
$\vdash \theta : \Delta$ assumption of Thm 31
$\vdash \theta' : \Delta, \overrightarrow{u{:}U}$ by index substitution typing
Let $\sigma' = \sigma, v/x$.
$\sigma \in [\![\Gamma]\!](\theta; \eta)$ assumption of Thm 31
$\sigma \in [\![\Gamma]\!](\theta'; \eta)$ since $\vec{u} \notin \mathrm{FV}(\Gamma)$

$\sigma' \in [\![\Gamma, x{:}R]\!](\theta'; \eta)$ by Def. 24
$s[\theta'; \sigma'] \Downarrow w \in [\![S]\!](\theta'; \eta)$ by I.H. with θ', η and σ'
$c \cdot \vec{M}\, v \Downarrow w$ by `e-app-lam`

Case:

$$\frac{\Delta; \Xi; \Gamma \vdash q \Rightarrow (\overrightarrow{u{:}U});\ R \to S \quad \Delta \vdash \vec{M} : (\overrightarrow{u{:}U}) \quad \Delta; \Xi; \Gamma \vdash r \Leftarrow R[\overrightarrow{M/u}]}{\Delta; \Xi; \Gamma \vdash q\, \vec{M}\, r \Rightarrow S[\overrightarrow{M/u}]}\ \texttt{t-app}$$

$q[\theta; \sigma] \Downarrow c \in [\![(\overrightarrow{u{:}U});\ R \to S]\!](\theta; \eta)$ by I.H.
$c \in [\![(\overrightarrow{u{:}U})]\!](\theta)$**,** $(\vec{N} \mapsto [\![R]\!](\theta, \overrightarrow{M/u}; \eta)) \to (\vec{N} \mapsto [\![S]\!](\theta, \overrightarrow{M/u}; \eta))$ by $[\![(\overrightarrow{u{:}U});\ R \to S]\!](\theta; \eta)$ def.
$\overrightarrow{M[\theta]} \in [\![(\overrightarrow{u{:}U})]\!](\theta)$ by Lemma 18.2
$r[\theta; \sigma] \Downarrow v \in [\![R[\overrightarrow{M/u}]]\!](\theta; \eta)$ by I.H.
$v \in [\![R]\!]((\mathsf{id}_\Delta, \overrightarrow{M/u})[\theta]; \eta)$ by Lemma 27
$v \in [\![R]\!](\theta, \overrightarrow{M[\theta]/u}; \eta)$ by Def. 8
$c \cdot \overrightarrow{M[\theta]}\, v \Downarrow w \in [\![S]\!](\theta, \overrightarrow{M[\theta]/u}; \eta)$ by Def. 21
$w \in [\![S]\!]((\mathsf{id}_\Delta, \overrightarrow{M/u})[\theta]; \eta)$ by Def. 8
$w \in [\![S[\overrightarrow{M/u}]]\!](\theta; \eta)$ by Lemma 27
$(q\, \vec{M}\, r)[\theta; \sigma] \Downarrow w$ by `e-app`

Case:

$$\frac{\Delta; \Xi; \Gamma \vdash t_1 \Leftarrow T_1 \quad \Delta; \Xi; \Gamma \vdash t_2 \Leftarrow T_2}{\Delta; \Xi; \Gamma \vdash \langle t_1, t_2 \rangle \Leftarrow T_1 \times T_2}\ \texttt{t-pair}$$

$t_i[\theta; \sigma] \Downarrow v_i \in [\![T_i]\!](\theta; \eta)$ for $i \in \{1, 2\}$ by I.H.
$\langle t_1, t_2 \rangle[\theta; \sigma] \Downarrow \langle v_1, v_2 \rangle$ by `e-pair`
$\langle v_1, v_2 \rangle \in [\![T_1 \times T_2]\!](\theta; \eta)$ by $[\![T_1 \times T_2]\!]$ def.

Case:

$$\frac{\Delta; \Xi; \Gamma \vdash p \Rightarrow T_1 \times T_2 \quad \Delta; \Xi; \Gamma, x_1{:}T_1, x_2{:}T_2 \vdash s \Leftarrow T}{\Delta; \Xi; \Gamma \vdash \mathtt{split}\, p\, \mathtt{as}\, \langle x_1, x_2 \rangle\, \mathtt{in}\, s \Leftarrow T}\ \texttt{t-split}$$

$p[\theta; \sigma] \Downarrow w \in [\![T_1 \times T_2]\!](\theta; \eta)$ by I.H.
$w = \langle v_1, v_2 \rangle$ where $v_i \in [\![T_i]\!](\theta; \eta)$ for $i \in \{1, 2\}$ by $[\![T_1 \times T_2]\!]$ def.
$\sigma, v_1/x_1, v_2/x_2 \in [\![\Gamma, x_1{:}T_1, x_2{:}T_2]\!](\theta; \eta)$ by Def. 24
$s[\theta; \sigma, v_1/x_1, v_2/x_2] \Downarrow v \in [\![T]\!](\theta; \eta)$ by I.H.
$t[\theta; \sigma] \Downarrow v$ by `e-split`

Case:

$$\frac{\Delta;\Xi;\Gamma \vdash t_i \Leftarrow T_i}{\Delta;\Xi;\Gamma \vdash \mathtt{in}_i\, t_i \Leftarrow T_1 + T_2}\ \mathtt{t\text{-}in}_i \qquad \text{for } i \in \{1,2\}$$

This is really two cases. Fix $i \in \{1,2\}$.

$t_i[\theta;\sigma] \Downarrow v_i \in [\![T_i]\!](\theta;\eta)$	by I.H.
$(\mathtt{in}_i\, t_i)[\theta;\sigma] \Downarrow \mathtt{in}_i\, v_i$	by $\mathtt{e\text{-}in}_i$.
$\mathtt{in}_i\, v_i \in \mathtt{in}_i\, [\![T_i]\!](\theta;\eta) \subseteq [\![T_1 + T_2]\!](\theta;\eta)$	by $[\![T_1 + T_2]\!]$ def.

Case:

$$\frac{\Delta;\Xi;\Gamma \vdash s \Rightarrow T_1 + T_2 \quad \Delta;\Xi;\Gamma, x_1{:}T_1 \vdash t_1 \Leftarrow T \quad \Delta;\Xi;\Gamma, x_2{:}T_2 \vdash t_2 \Leftarrow T}{\Delta;\Xi;\Gamma \vdash (\mathtt{case}\, s\, \mathtt{of}\, \mathtt{in}_1\, x_1 \mapsto t_1 \mid \mathtt{in}_2\, x_2 \mapsto t_2) \Leftarrow T}\ \mathtt{t\text{-}case}$$

$s[\theta;\sigma] \Downarrow w \in [\![T_1 + T_2]\!](\theta;\eta)$	by I.H.
$w \in \mathtt{in}_1\, [\![T_1]\!](\theta;\eta)$ or $w \in \mathtt{in}_2\, [\![T_2]\!](\theta;\eta)$	by $[\![T_1 + T_2]\!]$ def.
$w = \mathtt{in}_i\, v_i$ where $v_i \in [\![T_i]\!](\theta;\eta)$, for some $i \in \{1,2\}$.	
$\sigma \in [\![\Gamma]\!](\theta;\eta)$	by assumption of Thm 31
$\sigma, v_i/x_i \in [\![\Gamma, x_i{:}T_i]\!](\theta;\eta)$	by Def. 24
$t_i[\theta;\sigma, v_i/x_i] \Downarrow v \in [\![T]\!](\theta;\eta)$	by I.H. on t_i
$t[\theta;\sigma] \Downarrow v$	by $\mathtt{e\text{-}case\text{-}in}_i$

Case:

$$\frac{\Delta \vdash M : U \quad \Delta;\Xi;\Gamma \vdash s \Leftarrow S[M/u]}{\Delta;\Xi;\Gamma \vdash \mathtt{pack}\,(M,s) \Leftarrow \Sigma u{:}U.\, S}\ \mathtt{t\text{-}pack}$$

$s[\theta;\sigma] \Downarrow w \in [\![S[M/u]]\!](\theta;\eta)$	by I.H.
$w \in [\![S]\!]((\mathsf{id}_\Delta, M/u)[\theta];\eta)$	by Lemma 27
$w \in [\![S]\!](\theta, M[\theta]/u;\eta)$	by Def. 8
$(\mathtt{pack}\,(M,s))[\theta;\sigma] \Downarrow \mathtt{pack}\,(M[\theta], w)$	by $\mathtt{e\text{-}pack}$
$M[\theta] \in [\![U]\!](\theta)$	by Lemma 18.1
$\mathtt{pack}\,(M[\theta], w) \in [\![\Sigma u{:}U.\, S]\!](\theta;\eta)$	by $[\![\Sigma u{:}U.\, S]\!]$ def.

Case:

$$\frac{\Delta;\Xi;\Gamma \vdash p \Rightarrow \Sigma u{:}U.\, S \quad \Delta, u{:}U;\Xi;\Gamma, x{:}S \vdash q \Leftarrow T}{\Delta;\Xi;\Gamma \vdash \mathtt{unpack}\, p\, \mathtt{as}\, (u,x)\, \mathtt{in}\, q \Leftarrow T}\ \mathtt{t\text{-}unpack}$$

$p[\theta;\sigma] \Downarrow w' \in [\![\Sigma u{:}U.\, S]\!](\theta;\eta)$	by I.H.
$w' = \mathtt{pack}\,(M,w)$ where $M \in [\![U]\!](\theta)$ and $w \in [\![S]\!](\theta, M/u;\eta)$	by $[\![\Sigma u{:}U.\, S]\!]$ def.
Let $\theta' = \theta, M/u$.	
$\vdash \theta' : \Delta, u{:}U$	by index substitution typing
$\sigma \in [\![\Gamma]\!](\theta;\eta)$	assumption of Thm 31

$\sigma \in [\![\Gamma]\!](\theta'; \eta)$ since $u \notin \mathrm{FV}(\Gamma)$
Let $\sigma' = \sigma, w/x$.
$\sigma' \in [\![\Gamma, x{:}S]\!](\theta'; \eta)$ by Def. 24
$q[\theta'; \sigma'] \Downarrow v \in [\![T]\!](\theta'; \eta)$ by I.H.
$(\mathtt{unpack}\, p\, \mathtt{as}\, (u, x)\, \mathtt{in}\, q)[\theta; \sigma] \Downarrow v$ by `e-unpack`
$v \in [\![T]\!](\theta; \eta)$ since $u \notin \mathrm{FV}(T)$

Case:

$$\frac{\Delta \vdash M = N}{\Delta; \Xi; \Gamma \vdash \mathtt{refl} \Leftarrow M = N}\ \mathtt{t\text{-}refl}$$

$\vdash M[\theta] = N[\theta]$ by Req. 3.3
$\mathtt{refl} \in [\![M = N]\!](\theta; \eta)$ by $[\![M = N]\!]$ def.
$\mathtt{refl}[\theta; \sigma] \Downarrow \mathtt{refl}$ by `e-refl`

Case:

$$\frac{\Delta; \Xi; \Gamma \vdash q \Rightarrow M = N \quad \Delta \vdash M \doteqdot N \searrow (\Delta' \mid \Theta) \quad \Delta'; \Xi[\Theta]; \Gamma[\Theta] \vdash s \Leftarrow T[\Theta]}{\Delta; \Xi; \Gamma \vdash \mathtt{eq}\, q\, \mathtt{with}\, (\Delta'.\Theta \mapsto s) \Leftarrow T}\ \mathtt{t\text{-}eq}$$

$q[\theta; \sigma] \Downarrow w \in [\![M = N]\!](\theta; \eta)$ by I.H.
$w = \mathtt{refl}$ and $\vdash M[\theta] = N[\theta]$ by $[\![M = N]\!]$ def.
$\Delta \vdash M \doteqdot N \searrow (\Delta' \mid \Theta)$ premise of `t-eq`
$\vdash \theta : \Delta$ assumption of Thm 31
$\Delta' \vdash \Theta \doteq \theta \searrow (\cdot \mid \theta')$ by Req. 6
$[\![\Xi[\Theta]]\!](\theta') = [\![\Xi]\!](\Theta[\theta'])$ and $[\![T[\Theta]]\!](\theta'; \eta) = [\![T]\!](\Theta[\theta']; \eta)$ by Lemma 27
$[\![\Xi[\Theta]]\!](\theta') = [\![\Xi]\!](\theta])$ and $[\![T[\Theta]]\!](\theta'; \eta) = [\![T]\!](\theta; \eta)$ by Thm 5.2
Extending Lemma 27 from types T to typing contexts Γ,
$[\![\Gamma[\Theta]]\!](\theta'; \eta) = [\![\Gamma]\!](\Theta[\theta']; \eta) = [\![\Gamma]\!](\theta; \eta)$.
$\eta \in [\![\Xi[\Theta]]\!](\theta')$ since $\eta \in [\![\Xi]\!](\theta)$
$\sigma \in [\![\Gamma[\Theta]]\!](\theta'; \eta)$ since $\sigma \in [\![\Gamma]\!](\theta; \eta)$
$s[\theta'; \sigma] \Downarrow v \in [\![T[\Theta]]\!](\theta'; \eta)$ by I.H.
$v \in [\![T]\!](\theta; \eta)$
$(\mathtt{eq}\, q\, \mathtt{with}\, (\Delta'.\Theta \mapsto s))[\theta; \sigma] \Downarrow v$ by `e-eq`

Case:

$$\frac{\Delta; \Xi; \Gamma \vdash s \Rightarrow M = N \quad \Delta \vdash M \doteqdot N \searrow \#}{\Delta; \Xi; \Gamma \vdash \mathtt{eq\_abort}\, s \Leftarrow T}\ \mathtt{t\text{-}eqfalse}$$

$s[\theta; \sigma] \Downarrow w \in [\![M = N]\!](\theta; \eta)$ by I.H.
$w = \mathtt{refl}$ and $\vdash M[\theta] = N[\theta]$ by $[\![M = N]\!]$ def.

θ unifies M and N in Δ — by def. of a unifier

$\Delta \vdash M \doteqdot N \searrow \#$ — premise of `t-eqfalse`

There is no unifier for M and N — consequence of Req. 4

Contradiction: derive $(\texttt{eq\_abort}\, s)[\theta;\sigma] \Downarrow v \in [\![T]\!](\theta;\eta)$

Case:

$$\frac{\Delta;\Xi;\Gamma \vdash s \Leftarrow S[\overrightarrow{M/u}; \mu X{:}K.\, \Lambda\, \vec{u}.\, S/X]}{\Delta;\Xi;\Gamma \vdash \texttt{in}_\mu\, s \Leftarrow (\mu X{:}K.\, \Lambda\, \vec{u}.\, S)\, \vec{M}}\ \texttt{t-in}_\mu$$

Let $R = \mu X{:}K.\, \Lambda\, \vec{u}.\, S$.

$s[\theta;\sigma] \Downarrow w \in [\![S[\overrightarrow{M/u}; R/X]]\!](\theta;\eta)$ — by I.H.

$(\texttt{in}_\mu\, s)[\theta;\sigma] \Downarrow \texttt{in}_\mu\, w$ — by $\texttt{e-in}_\mu$

$\texttt{in}_\mu\, w \in [\![R\, \vec{M}]\!](\theta;\eta)$ — by Lemma 28

Case:

$$\frac{\Delta;\Xi, X{:}K;\Gamma, f{:}(\overrightarrow{u{:}U});\ X\, \vec{u} \to S \vdash s \Leftarrow (\overrightarrow{u{:}U});\ R[\overrightarrow{u/u'}] \to S \quad \vec{u} \notin \mathrm{FV}(R)}{\Delta;\Xi;\Gamma \vdash \texttt{rec}\, f.\, s \Leftarrow (\overrightarrow{u{:}U});\ (\mu X{:}K.\, \Lambda\, \overrightarrow{u'}.\, R)\, \vec{u} \to S}\ \texttt{t-rec}$$

Let $c = (\texttt{rec}\, f.\, s)[\theta;\sigma]$ and $C = (\mu X{:}K.\, \Lambda\, \overrightarrow{u'}.\, R)\, \vec{u}$. By `e-rec`, $(\texttt{rec}\, f.\, s)[\theta;\sigma] \Downarrow c$. We need to show that $c \in [\![(\overrightarrow{u{:}U});\ C \to S]\!](\theta;\eta)$,

Let $\vec{\mathcal{U}} = [\![(\overrightarrow{u{:}U})]\!](\theta) = [\![(\overrightarrow{u'{:}U})]\!](\theta)$ (both $\vec{u}$ and $\overrightarrow{u'}$ do not appear in θ). From the kinding rules we know that $K = \Pi\overrightarrow{u'{:}U}.\, *$ so $[\![K]\!](\theta) = \vec{\mathcal{U}} \to \mathcal{P}(\Omega)$. Let $\mathcal{L} = [\![K]\!](\theta)$ and define $\mathcal{F} \in \mathcal{L} \to \mathcal{L}$ by $\mathcal{F}(\mathcal{X}) = \texttt{in}^*_\mu\, [\![\Lambda\, \overrightarrow{u'}.\, R]\!](\theta;\eta,\mathcal{X}/X)$.

For $\vec{M} \in \vec{\mathcal{U}}$,

$$\begin{aligned}
[\![C]\!](\theta, \overrightarrow{M/u};\eta) &= [\![\mu X{:}K.\, \Lambda\, \overrightarrow{u'}.\, R]\!](\theta, \overrightarrow{M/u};\eta)(\vec{u}[\theta, \overrightarrow{M/u}]) && \text{by } [\![T]\!] \text{ def} \\
&= [\![\mu X{:}K.\, \Lambda\, \overrightarrow{u'}.\, R]\!](\theta;\eta)(\vec{M}) \\
&\qquad \text{dropping mappings for } \vec{u} \text{ since } \vec{u} \notin \mathrm{FV}(R) \\
&= \boldsymbol{\mu}(\mathcal{X} \mapsto \texttt{in}^*_\mu\, [\![\Lambda\, \overrightarrow{u'}.\, R]\!](\theta;\eta,\mathcal{X}/X))(\vec{M}) && \text{by } [\![T]\!] \text{ def} \\
&= (\boldsymbol{\mu}\mathcal{F})(\vec{M}) && \text{by } \mathcal{F} \text{ def.}
\end{aligned}$$

Define $\mathcal{B} \in \mathcal{L}$ by $\mathcal{B}(\vec{M}) = [\![S]\!](\theta, \overrightarrow{M/u};\eta)$. Then

$$[\![(\overrightarrow{u{:}U});\ C \to S]\!](\theta;\eta) = \vec{\mathcal{U}},\ \boldsymbol{\mu}\mathcal{F} \to \mathcal{B} \qquad \text{by } [\![T]\!] \text{ def.}$$

We want to show $c \in \vec{\mathcal{U}},\ \boldsymbol{\mu}\mathcal{F} \to \mathcal{B}$. We will instead prove the sufficient condition given in Lemma 26. To this end, suppose $\mathcal{X} \in \vec{\mathcal{U}} \to \mathcal{P}(\Omega) = \mathcal{L}$ and $c \in \vec{\mathcal{U}},\ \mathcal{X} \to \mathcal{B}$. The goal is now to show $c \in \vec{\mathcal{U}},\ \mathcal{F}(\mathcal{X}) \to \mathcal{B}$.

Define $\mathcal{A} \in \mathcal{L}$ by $\mathcal{A} = [\![\Lambda\, \overrightarrow{u'}.\, R]\!](\theta; \eta, \mathcal{X}/X)$, so $\mathcal{F}(\mathcal{X}) = \mathtt{in}_\mu^*\, \mathcal{A}$. By Lemma 29, it suffices to show that

$$s[\theta; \sigma, c/f] \Downarrow c' \in \vec{\mathcal{U}},\ \mathcal{A} \to \mathcal{B}.$$

We need to interpret the types $(\overrightarrow{u{:}U});\ X\, \vec{u} \to S$ and $(\overrightarrow{u{:}U});\ R[\overrightarrow{u/u'}] \to S$ appearing in the premise of `t-rec`. Note that these types are well-kinded under the contexts $\Delta; \Xi, X{:}K$. Since $\mathcal{X} \in \mathcal{L} = [\![K]\!](\theta)$, we interpret them under the environments θ and $\eta, \mathcal{X}/X \in [\![\Xi, X{:}K]\!](\theta)$.

$$\begin{aligned}
&[\![(\overrightarrow{u{:}U});\ X\, \vec{u} \to S]\!](\theta; \eta, \mathcal{X}/X) \\
&= \vec{\mathcal{U}},\ (\vec{M} \mapsto [\![X\, \vec{u}]\!](\theta, \overrightarrow{M/u}; \eta, \mathcal{X}/X)) \to (\vec{M} \mapsto [\![S]\!](\theta, \overrightarrow{M/u}; \eta, \mathcal{X}/X)) \\
&= \vec{\mathcal{U}},\ (\vec{M} \mapsto \mathcal{X}(\vec{M})) \to (\vec{M} \mapsto [\![S]\!](\theta, \overrightarrow{M/u}; \eta)) \\
&\qquad \text{dropping a mapping for } X \text{ since } X \notin \mathrm{FV}(S) \\
&= \vec{\mathcal{U}},\ \mathcal{X} \to \mathcal{B}
\end{aligned}$$

$$\begin{aligned}
&[\![(\overrightarrow{u{:}U});\ R[\overrightarrow{u/u'}] \to S]\!](\theta; \eta, \mathcal{X}/X) \\
&= \vec{\mathcal{U}},\ (\vec{M} \mapsto [\![R[\overrightarrow{u/u'}]]\!](\theta, \overrightarrow{M/u}; \eta, \mathcal{X}/X)) \to (\vec{M} \mapsto [\![S]\!](\theta, \overrightarrow{M/u}; \eta, \mathcal{X}/X)) \\
&= \vec{\mathcal{U}},\ (\vec{M} \mapsto [\![R]\!]((\mathsf{id}_\Delta, \overrightarrow{u/u'})[\theta, \overrightarrow{M/u'}]; \eta, \mathcal{X}/X)) \to (\vec{M} \mapsto [\![S]\!](\theta, \overrightarrow{M/u}; \eta)) \\
&\qquad \text{by Lemma 27 and again dropping a mapping for } X \\
&= \vec{\mathcal{U}},\ (\vec{M} \mapsto [\![R]\!](\theta, \overrightarrow{M/u'}; \eta, \mathcal{X}/X)) \to (\vec{M} \mapsto [\![S]\!](\theta, \overrightarrow{M/u}; \eta)) \\
&= \vec{\mathcal{U}},\ \mathcal{A} \to \mathcal{B}.
\end{aligned}$$

Our assumption from Lemma 26 is that $c \in \vec{\mathcal{U}},\ \mathcal{X} \to \mathcal{B}$. Moreover, since $X \notin \mathrm{FV}(\Gamma)$, $[\![\Gamma]\!](\theta; \eta, \mathcal{X}/X) = [\![\Gamma]\!](\theta; \eta) \ni \sigma$. Hence $\sigma, c/f \in [\![\Gamma, f{:}(\overrightarrow{u{:}U});\ X\, \vec{u} \to S]\!](\theta; \eta, \mathcal{X}/X)$. Now we can apply the induction hypothesis with $\eta' = \eta, \mathcal{X}/X$ and $\sigma' = \sigma, c/f$ to learn that $s[\theta; \sigma'] \Downarrow c'$ where $c' \in [\![(\overrightarrow{u{:}U});\ R[\overrightarrow{u/u'}] \to S]\!](\theta; \eta') = \vec{\mathcal{U}},\ \mathcal{A} \to \mathcal{B}$.

Case:

$$\frac{\Delta; \Xi; \Gamma \vdash s \Leftarrow T_z\, \vec{M}}{\Delta; \Xi; \Gamma \vdash \mathtt{in}_0\, s \Leftarrow T_{\mathtt{Rec}}\, 0\, \vec{M}}\ \mathtt{t\text{-}in_0}$$

$s[\theta; \sigma] \Downarrow w \in [\![T_z\, \vec{M}]\!](\theta; \eta)$	by I.H.
$(\mathtt{in}_0\, s)[\theta; \sigma] \Downarrow \mathtt{in}_0\, w$	by `e-in`$_0$
$\mathtt{in}_0\, w \in \mathtt{in}_0\, [\![T_z\, \vec{M}]\!](\theta; \eta)$	
$\mathtt{in}_0\, w \in [\![T_{\mathtt{Rec}}\, 0\, \vec{M}]\!](\theta; \eta)$	by Lemma 30

Case:

$$\frac{\Delta; \Xi; \Gamma \vdash s \Leftarrow T_s[N/u; (T_{\mathtt{Rec}}\, N)/X]\, \vec{M}}{\Delta; \Xi; \Gamma \vdash \mathtt{in}_{\mathsf{suc}}\, s \Leftarrow T_{\mathtt{Rec}}\, (\mathsf{suc}\, N)\, \vec{M}}\ \mathtt{t\text{-}in}_{\mathsf{suc}}$$

$s[\theta; \sigma] \Downarrow w \in [\![T_s[N/u; (T_{\mathtt{Rec}}\, N)/X]\, \vec{M}]\!](\theta; \eta)$ by I.H.
$(\mathtt{in}_{\mathsf{suc}}\, s)[\theta; \sigma] \Downarrow \mathtt{in}_{\mathsf{suc}}\, w$ by $\mathtt{e\text{-}in}_{\mathsf{suc}}$
$\mathtt{in}_{\mathsf{suc}}\, w \in \mathtt{in}_{\mathsf{suc}}\, ([\![T_s[N/u; (T_{\mathtt{Rec}}\, N)/X]\, \vec{M}]\!](\theta; \eta))$
$\mathtt{in}_{\mathsf{suc}}\, w \in [\![T_{\mathtt{Rec}}\, (\mathsf{suc}\, N)\, \vec{M}]\!](\theta; \eta)$ by Lemma 30

Case:

$$\frac{\Delta; \Xi; \Gamma \vdash s \Rightarrow T_{\mathtt{Rec}}\, 0\, \vec{M}}{\Delta; \Xi; \Gamma \vdash \mathtt{out}_0\, s \Rightarrow T_z\, \vec{M}}\ \mathtt{t\text{-}out}_0$$

$s[\theta; \sigma] \Downarrow w \in [\![T_{\mathtt{Rec}}\, 0\, \vec{M}]\!](\theta; \eta)$ by I.H.
$w = \mathtt{in}_0\, v$ for some $v \in ([\![T_z\, \vec{M}]\!](\theta; \eta))$ by Lemma 30
$(\mathtt{out}_0\, s)[\theta; \sigma] \Downarrow v$ by $\mathtt{e\text{-}out}_0$

Case:

$$\frac{\Delta; \Xi; \Gamma \vdash s \Rightarrow T_{\mathtt{Rec}}\, (\mathsf{suc}\, N)\, \vec{M}}{\Delta; \Xi; \Gamma \vdash \mathtt{out}_{\mathsf{suc}}\, s \Rightarrow T_s[N/u; (T_{\mathtt{Rec}}\, N)/X]\, \vec{M}}\ \mathtt{t\text{-}out}_{\mathsf{suc}}$$

$s[\theta; \sigma] \Downarrow w \in [\![T_{\mathtt{Rec}}\, (\mathsf{suc}\, N)\, \vec{M}]\!](\theta; \eta)$ by I.H.
$w = \mathtt{in}_{\mathsf{suc}}\, v$ for some $v \in [\![T_s[N/u; (T_{\mathtt{Rec}}\, N)/X]\, \vec{M}]\!](\theta; \eta)$ by Lemma 30
$(\mathtt{out}_{\mathsf{suc}}\, s)[\theta; \sigma] \Downarrow v$ by $\mathtt{e\text{-}out}_{\mathsf{suc}}$

Case:

$$\frac{\Delta; \Xi; \Gamma \vdash t_z \Leftarrow S[0/u] \quad \Delta, u{:}\,\mathsf{nat}; \Xi; \Gamma, x{:}S \vdash t_s \Leftarrow S[\mathsf{suc}\, u/u]}{\Delta; \Xi; \Gamma \vdash \mathtt{ind}\, t_z\, (u,\, x.\, t_s) \Leftarrow (u{:}\,\mathsf{nat});\, 1 \to S}\ \mathtt{t\text{-}ind}$$

Let c be the closure $(\mathtt{ind}\, t_z\, (u,\, x.\, t_s))[\theta; \sigma]$.
$(\lambda\, \vec{u}, x.\, s)[\theta; \sigma] \Downarrow c$ by $\mathtt{e\text{-}ind}$
Suffices to show $c \in [\![(u{:}\,\mathsf{nat});\, 1 \to S]\!](\theta; \eta)$, i.e.
$\forall N \in [\![\mathsf{nat}]\!].\ c \cdot N\, \langle\rangle \Downarrow w \in [\![S]\!](\theta, N/u; \eta)$.
Proceed by induction on N.
Base case: $N = 0$.
$t_z[\theta; \sigma] \Downarrow w \in [\![S[0/u]]\!](\theta; \eta)$ by I.H.
$w \in [\![S]\!]((\mathsf{id}_\Delta, 0/u)[\theta]; \eta)$ by Lemma 27
$w \in [\![S]\!](\theta, 0/u; \eta)$ by Def. 8
$c \cdot 0\, \langle\rangle \Downarrow w$ by $\mathtt{e\text{-}app\text{-}ind}_0$
Step case: $N = \mathsf{suc}\, N'$ for some $N' \in [\![\mathsf{nat}]\!]$.

$c \cdot N' \langle\rangle \Downarrow v \in [\![S]\!](\theta, N'/u; \eta)$	by inner I.H.
Let $\theta' = \theta, N'/u$, so $\vdash \theta' : \Delta, u{:}\,\mathsf{nat}$.	
$\sigma \in [\![\Gamma]\!](\theta'; \eta)$	since $u \notin \mathrm{FV}(\Gamma)$
$\sigma, v/x \in [\![\Gamma, x{:}S]\!](\theta'; \eta)$	by Def. 24
$t_s[\theta'; \sigma, v/x] \Downarrow w \in [\![S[\mathsf{suc}\, u/u]]\!](\theta'; \eta)$	by I.H.
$w \in [\![S]\!]((\mathsf{id}_\Delta, \mathsf{suc}\, u/u)[\theta']; \eta)$	by Lemma 27
$w \in [\![S]\!](\theta', (\mathsf{suc}\, u)[\theta']/u; \eta)$	by Def. 8
$w \in [\![S]\!](\theta', \mathsf{suc}\, N'/u; \eta)$	by Def. 7
$w \in [\![S]\!](\theta, N/u; \eta)$	overwriting u in θ'
$c \cdot N \langle\rangle \Downarrow w$	by $\texttt{e-app-ind}_{\mathsf{suc}}$

□

3.5 Example: Encoding Logical Relations

Our main applications for the combination of inductive and stratified definitions are proofs involving logical relations. In this section we show how Tores can be used to implement a simple yet important use of a logical relation: Tait's normalization argument for the simply typed lambda calculus. We show how to mechanize the proof in Beluga, whose theory closely resembles Tores. This section is a sample of a case study by Cave and Pientka [2013], with the full mechanization available online[1].

Note that the proof is also similar to our termination proof in Section 2.3, differing in the following ways:

- The language is simpler, with only unit and function types.
- We use a small-step reduction relation instead of a big-step semantics.
- The logical relation describes normalizing terms instead of values.
- We lead to an encoding in Beluga using a mix of LF, inductive and stratified definitions.

Let us start by introducing our notation. The grammar of lambda terms is straightforward: we have variables, lambda abstractions and applications, and the constant () of type unit. Lambda abstractions and the unit constant are considered values.

$$\begin{array}{lll} \text{Terms } M, N & ::= & x \mid \mathsf{lam}\ x.M \mid \mathsf{app}\ M\ N \mid () \\ \text{Types } A, B & ::= & \mathsf{unit} \mid A \to B \end{array}$$

Next we define well-typed terms, using the judgment $M : A$. The typing rules follow Gentzen's natural deduction style, where the context of assumptions is implicit. This foreshadows its encoding in the logical framework LF [Harper et al., 1993].

[1] https://github.com/Beluga-lang/Beluga/blob/master/examples/logrel/weak-norm-total-mix-lf-inductive.bel

$\boxed{M : A}$ Term M has type A

$$\frac{\begin{array}{c}\overline{x : A}\\ \vdots \\ M : B\end{array}}{\mathsf{lam}\ x.M : A \to B}\ \text{lam} \qquad \frac{M : A \to B \quad N : A}{\mathsf{app}\ M\ N : B}\ \text{app} \qquad \frac{}{() : \mathsf{unit}}\ \text{unit}$$

Note that in the typing rule for lambda abstractions, we must show that the body M has type B assuming the fresh variable x has type A.

We now define our reduction relation, written $M \Rightarrow N$. This is a *weak head reduction* because we do not reduce in the body of a lambda abstraction. Therefore the only cases of reduction are for applications. The rules say that we reduce the first term in an application until we reach a lambda abstraction, and then we can substitute the second term of the application into the body of the lambda.

$\boxed{M \Rightarrow N}$ Term M steps to N

$$\frac{}{\mathsf{app}\ (\mathsf{lam}\ x.M)\ N \Rightarrow [N/x]M}\ \text{Beta} \qquad \frac{M \Rightarrow M'}{\mathsf{app}\ M\ N \Rightarrow \mathsf{app}\ M'\ N}\ \text{StepApp}$$

Formally we will also require the reflexive transitive closure of the reduction relation above. This allows us to state that a term M halts if it takes zero or more steps to a value.

We can now define a logical relation which describes normalizing terms. This unary logical relation can be viewed as defining sets, called *reducible* sets. We say that a term is reducible at the unit type precisely when it halts. A term M is reducible at the arrow type $A \to B$ if it halts and for every N reducible at type A, the application of M to N is reducible at type B.

$$\begin{array}{lcl}\mathcal{R}_{\mathsf{unit}} & = & \{M \mid M \text{ halts}\} \\ \mathcal{R}_{A \to B} & = & \{M \mid M \text{ halts and } \forall N{:}A.\ \text{if } N \in \mathcal{R}_A \text{ then } (\mathsf{app}\ M\ N) \in \mathcal{R}_B\}\end{array}$$

The above definition is well-founded because it is by induction on the structure of the type.

Now, we can prove that all well-typed terms must halt using two proof steps. First, we show that each well-typed term is reducible at its type. Then we show that every reducible term halts. The second part is trivial since our definition of reducibility includes halting. The first part requires an inductive proof on well-typed terms.

In order to mechanize such a proof, we must decide how to define well-typed terms, the reduction relation, and reducible sets. We describe one choice of encodings here.

When we encode types and terms, we must model variables and their scope. We will use the logical framework LF and model binding in our object language by reusing the binding of LF. This representation, known as *higher-order abstract syntax (HOAS)*, allows us to uniformly represent binding structures by mapping them to the LF function space and inherit α-renaming and substitution operations from LF. We show the encoding of types and terms in LF below.

```
LF tp : type =
| unit : tp
| arr  : tp → tp → tp;

LF tm : tp → type =
| lam : (tm A → tm B) → tm (arr A B)
| app : tm (arr A B) → tm A → tm B
| u   : tm unit;
```

We previously defined the grammar of types and terms prior to the typing relation, but in our encoding it is more concise and convenient to define *intrinsically typed* terms using the LF type family `tm: tp → type`. Using BELUGA's concrete syntax, free variables such as A and B are quantified implicitly at the outside. There is no case for variables, as they are treated implicitly via higher-order abstract syntax. Note that the LF function space is a weak, *representational* function space: it does not allow case analysis or recursion, and hence only contains genuine lambda terms.

Next, we encode the reduction relation $M \Rightarrow N$ as an inductive definition. Note that we could again encode it as an LF type family, but in this case we want to highlight our indexed recursive types.

```
inductive Step : [tm A] → [tm A]→ type =
  | Beta   : Step [app (lam λx.M) N] [M[N]]
  | StepApp: Step [M] [M'] → Step [app M N] [app M' N];
```

Here the inductive type `Step` is indexed by closed LF objects of type `tm A`. In BELUGA, we wrap index objects and types inside `[ ]` to distinguish them from computations and computation-level types. The constructor `Beta` represents the β-reduction rule, where replace the variable `x` in the body of `lam λx.M` with the term `N`. Note that we are utilizing LF substitution, written `M[N]`.[2]

Finally, we encode the definition of reducible sets $\mathcal{R}_A$. Recall that our definition is well-founded as it is inductive in the type A. This justifies the clause for $\mathcal{R}_{A\to B}$, despite the negative occurrence of $\mathcal{R}$. We therefore encode $\mathcal{R}_A$ as a type `Reduce` that is *stratified* by the index representing the object-level type A.

```
stratified Reduce : ΠA:[tp]. [tm A] → type =
| I  : Halts [M] → Reduce [unit] [M]
| Arr: Halts [M] → (ΠN:[tm A]. Reduce [A] [N] → Reduce [B] [app M N])
                 → Reduce [arr A B] [M];
```

Note that we cannot perform induction directly over reducible sets; rather they are unfolded in proofs according to the type index. In our case, when we prove reducibility by induction over well-typed terms, we learn in some cases that the type is an arrow type, which allows the reducibility definition to be unfolded.

[2]By default, the term `M` depends on all the bound variables in whose context it occurs; in particular, it may depend on `x`. This differs from the usual HOAS notation.

3.6 Conclusion

We presented a core language Tores extending an indexed type system with recursive types and stratified types. We argued that Tores provides a sound and powerful foundation for programming inductive proofs, in particular those involving logical relations. This power comes from the induction principles on recursive types given by Mendler-style recursion as well as the flexibility of recursive definitions given by stratified types. Type checking in Tores is decidable and types are preserved during evaluation. The soundness of our language is guaranteed by our logical predicate semantics and termination proof. We believe that Tores balances well the proof-theoretic power with a simple metatheory (especially when compared with full dependent types).

Chapter 4

Related and Future Work

4.1 Mendler-style Recursion

Our treatment of induction over recursive types is due to Mendler [1991]. He describes the addition of least and greatest fixed points (inductive and coinductive types) to a second-order lambda calculus and proves strong normalization. He gives types to (co-)recursion using suitably generic polymorphic types. We on the other hand present Mendler-style recursion independently of polymorphism.

In terms of expressivity, the simple form of Mendler recursion describes a class of programs that are called *iterative* [Matthes, 1999]. This is in fact a smaller class than that of primitive recursive programs. To recover all primitive recursive programs, Mendler [1987] gave a slight generalization to his typing rule for recursion. This allows the subterm of a recursive type to be used not only as the argument to a recursive call but also as a member of the recursive type itself. In particular, this permits a constant-time implementation of the predecessor function, which is impossible with pure Mendler iteration. The extension to primitive recursion requires a simple notion of subtyping so that the recursive type variable X is a subtype of the recursive type $\mu X.\, T$. Gimenez [1998] generalizes this idea further to type an even larger class of terminating programs given by *course-of-value recursion*. This leads to the general framework of *sized types*, which Gimenez described for the Calculus of Constructions. Abel [2010] gives an approachable account of this progression of terminating recursion schemes, which we have summarized here.

One point of difference to related work is in our semantics of Mendler recursion. In our interpretation of recursive types, we use the following pre-fixed point definition (showing the simply typed version here): $\boldsymbol{\mu}\mathcal{F} = \bigcap\{\mathcal{C} \subseteq \Omega \mid \forall\mathcal{X}.\ \mathcal{X} \subseteq \mathcal{C} \implies \mathcal{F}(\mathcal{X}) \subseteq \mathcal{C}\}$. Our pre-fixed point condition on $\mathcal{C}$ is inspired by Mendler's typing rule for recursion. The connection is made clearer by observing the generalization of Mendler recursion to higher kinds [Abel et al., 2005]. The similarity in structure between our fixed point property and the Mendler typing rule makes for a more elegant termination proof.

Other interpretations of recursive types, such as by Abel [2010], build the least fixed point iteratively

using *ordinal induction*. This formulation uses a set union of the form

$$\boldsymbol{\mu}\mathcal{F} = \bigcup_{i \in \omega_1} \mathcal{F}^i(\phi),$$

where ω_1 is the least uncountable ordinal. In fact, Mendler's proof of strong normalization also uses ordinal induction in the case of least fixed points. We prefer our impredicative formulation because it can be encoded directly in second-order logic. This is particularly advantageous for our mechanization in Section 2.4, as we do not require proofs of ordinal properties.

4.2 Dependent Types

There are many systems closely related to Tores in the world of dependent types. The modern proof assistants Coq [Bertot and Castéran, 2004] and Agda [Norell, 2007], for example, are based on the foundational languages of the Calculus of Constructions [Coquand and Huet, 1988] and Martin-Löf type theory [Martin-Löf, 1973] respectively. In those languages, indices appearing in types are not restricted to a designated index domain, as they are in Tores: instead, they may be programs or types from the same language. In this sense, programs and types are intertwined in dependently typed languages.

This approach leads to a more powerful theory but complicates the metatheory. For one, consider type checking in a dependently typed language. Because programs may appear in types (and in general types may be defined by functions), one must normalize such types during the type checking phase. Thus the decidability of type checking is tied to the normalizability of terms. In contrast, our presentation using indexed types provides decidability of type checking given the decidability of index term equality and unification, which we ensure in advance. In fact, we specified a bidirectional type checking algorithm, which allows type inference given sufficient annotations (usually only at top-level declarations).

Now consider the normalization argument for a dependently typed language (from which decidability of type checking usually follows). The proof is again complicated by the mixing of terms and types. One cannot semantically interpret types independently of terms, because terms may appear in types, and so one must talk about the normalization of types and programs simultaneously. The issue is not at all disastrous: one can see for example an elegant extension of the logical predicate (or *saturated sets*) method to handle the Calculus of Constructions with inductive types by Geuvers [1995]. We on the other hand avoid this issue entirely in the study of Tores, because the only terms that may appear in types are from an index language with no real form of computation.

The next important point of comparison is regarding treatments of recursive or inductive types. Dependently typed languages such as Coq and Agda employ a *strict positivity* restriction on the formation of recursive types: recursive type variables are not allowed to occur in the domain of a (dependent) function type. This ensures termination in the presence of recursive types. The more interesting question is how induction principles are added to theories with inductive types. This has been addressed by Paulin-Mohring [1993] for the Calculus of Constructions (underlying Coq) and by Dybjer [1994] for Martin-Löf type theory (underlying Agda). Both give generic elimination schemes for inductive types based on the structure of the

type. We on the other hand take the approach of Mendler-style recursion which gives a generic induction principle for recursive types without enforcing a positivity restriction.

Another point of comparison is between the index-stratified types in Tores and so-called *large eliminations* in dependently typed languages. Large eliminations, using the terminology of Altenkirch and Werner, are definitions of dependent types by primitive recursion. For example, a large elimination on a natural number is of the form $\texttt{Rec}\ t\ T_0\ (X.\, T_{\mathsf{suc}})$ (very similar to our stratified type), analogous to a "small" elimination which would simply be the usual program-level primitive recursion. In a dependent type theory, this large elimination reduces in the same way as the small elimination, depending on the value of the natural number expression t.

Large eliminations are important for increasing the expressive power of dependent type theories, in particular allowing one to prove that constructors of inductive types are disjoint (e.g. $0 \neq 1$). Jan Smith [Smith, 1989] gave an account of large eliminations (calling them *propositional functions*) as an extension of Martin-Löf type theory. Werner [1992] was able to prove strong normalization for a language (dependently typed System F) with large eliminations over natural numbers. His proof is similar to Geuvers' proof for the Calculus of Constructions, though it requires a semantic interpretation of the large elimination form. Naturally, Werner's solution involves a semantic operator which computes interpretations by primitive recursion in the metalanguage. His interpretation also needs to consider the normalizability of the natural number argument to the large elimination. Our interpretation of stratified types has a similar structure, but is simplified by the fact that the natural number argument comes from the index language, and is therefore trivially in a normal form.

On a broader note, our development of Tores shows how to gain the power of large eliminations in an indexed type system. Since we have no reduction on the level of types, we instead simulate this reduction by unfolding stratified types in the typing rules. Again this feature of typing does not rely on the normalization of programs.

4.3 Proof Theory

In the world of proof theory, our core language corresponds to a first-order logic with equality, inductive and stratified (recursive) definitions. Momigliano and Tiu [2004], Tiu and Momigliano [2012] give comprehensive treatments of logics with induction and co-induction as well as first-class equality. They present their logics in a sequent calculus style and prove cut elimination (i.e. that the cut rule is admissible) which implies consistency of the logics. Their cut elimination proof extends Girard's proof technique of reducibility candidates, similar to ours. Unlike our work, however, theirs handles co-induction. Note that they require strict positivity of inductive definitions, i.e. the head of a definition (analogous to the recursive type variable) is not allowed to occur to the left of an implication.

Tiu [2012] also develops a first-order logic with stratified definitions similar to our stratified types. His notion of stratification comes from defining the "level" of a formula, which measures its size. A recursive definition is then called stratified if the level does not increase from the head of the definition to the body.

This is a more general formulation than our notion of stratification for types: we require the type to be stratified exactly according to the structure of an index term, instead of a more general decreasing measure. However, we could potentially replicate such a measure by suitably extending our index language.

Another approach to supporting recursive definitions in proof theory is via a rewriting relation, as in the Deduction Modulo system [Dowek et al., 2003]. The idea of this system is to generalize a given first-order logic to account for a congruence relation defined by a set of rewrite rules. This rewriting could include recursive definitions in the same sense as Tiu. Dowek and Werner [2003] show that such logics under congruences can be proven normalizing given general conditions on the congruence. Specifically, they define the notion of a *pre-model* of a congruence, whose existence is sufficient to prove cut elimination of the logic.

Baelde and Nadathur [2012] extend this work in the following way. First, they present a first-order logic with inductive and co-inductive definitions, together with a general form of equality. They show strong normalization for this logic using a reducibility candidate argument. Crucially, their proof is in terms of a pre-model which anticipates the addition of recursive definitions via a rewrite relation. Then they give conditions on the rewrite rules, essentially requiring that each definition follows a well-founded order on its arguments. Under these conditions, they are able to construct a pre-model for the relation, proving normalization as a result. Again their notion of stratification of recursive definitions is slightly more general than ours. However, our treatment is perhaps more direct as the rewriting of types takes place within our typing rules, and our semantic model accounts for stratified types directly.

4.4 Indexed Types

Mendler-style recursion schemes for term-indexed languages have been investigated by Ahn [2014]. He describes an extension of System F_ω with erasable term indices, called F_i. He combines this with fixed points of type operators, as in the Fix_ω language by Abel and Matthes [2004], to produce the core language Fix_i. In Fix_i, one can embed Mendler-style recursion over term-indexed data types by Church-style encodings.

Fundamentally, our use of indices is more liberal than in Ahn's core languages. In F_i, term indices are drawn from the same term language as programs. They are treated polymorphically, in analogue with polymorphic type indices, i.e. they must have closed types and cannot be analyzed at runtime. Our approach is to separate the language of index terms from the language of programs. In Tores, the indices that appear in types can be handled and analyzed at runtime, may be dependently typed and have types with free variables. This flexibility is crucial for writing inductive proofs over LF specifications as we do in Beluga.

Another difference from Ahn's work is our treatment of Mendler-style recursion. Ahn is able to embed a variety of Mendler-style recursion schemes via Church encodings, taking advantage of polymorphism and type-level functions inherited from System F_ω. Our work does not include polymorphism and general type-level functions as we concentrate on a small core language for inductive reasoning. For this purpose, Tores includes recursive types with a Mendler-style elimination form. We believe this treatment is a more direct interpretation of Mendler recursion for indexed recursive types.

Ahn's core languages do not support stratified types, but they may be possible to encode using type-level

abstractions over term indices. We leave this investigation to future work.

4.5 General-purpose Languages

One can draw a comparison between our work and certain typed functional languages designed for general-purpose programming. The prominent example we have in mind is Haskell [Peyton Jones, 2003]. Haskell now has advanced type system features which allow it to replicate aspects of indexed types which we emphasize in Tores. Although programs in Haskell are not guaranteed to be terminating, and hence it is does not truly serve as a proof assistant, we can still draw parallels as a programming language.

Haskell (and in fact also OCaml) supports indices in types by way of *generalized algebraic data types (GADTs)* [Xi et al., 2003, Peyton Jones et al., 2006]. GADTs allow one to define families of types indexed by other types. One can simulate an index term language by defining uninhabited dummy types to act as index terms. Although something of a workaround, this can be done safely and conveniently using data type promotion [Yorgey et al., 2012]. This allows a limited form of dependently typed programming in Haskell.

Going further, one can even simulate large eliminations or stratified types using the concept of *type families* [Chakravarty et al., 2005]. Type families establish equivalences between types, essentially allowing functions at the type level. Such a function can be recursive on the dummy types mentioned above (which act as our index terms), thus resembling a large elimination or stratified type.

Note that Haskell does not allow direct analysis or induction over indices. This is because indices are really types which are erased before run-time. The design choice of type-erasure is important for run-time performance, but sacrifices flexibility in how we can use indices.

4.6 Intensional Type Analysis

Finally, we point out a relationship to work on intensional/flexible type analysis, which aims to support runtime analysis of types. The original system by Harper and Morrisett [1995] includes a `Typerec` operator that defines types by recursion on a type argument. The LX system by Crary and Weirich [1999] also has a rich type constructor language including means for defining types by primitive (in fact, Mendler-style) recursion. Unlike in those systems, our `Rec` operator defines types by recursion on index terms instead of types. Fundamentally, the goals of intensional type analysis are vastly different from our own. We focus on providing a type-theoretic foundation for programming inductive proofs, whereas the above authors are concerned with efficient compilation of polymorphic programs.

4.7 Future Work

There are a few directions in which we could take our work on Tores. Firstly, polymorphism is notably missing from the language, though it should be straight-forward to add. This feature would enable richer

second-order encodings and propositions to our currently first-order framework. Secondly, it may be worthwhile to strengthen the induction principles that we support for recursive types. As described in Section 4.1, our basic Mendler recursion scheme allows only iterative programs and proofs. We could extend it to cover primitive recursive or course-of-value recursive programs, if this extra power turns out to be needed in practical proofs.

More interestingly, we would like Tores to support coinductive proofs via Mendler-style *corecursion*. This involves adding another form of recursive type representing a coinductive type, or greatest fixed point, of the form $\nu X{:}K.\, T$. The typing rules are dual to those for Mendler-style inductive types (essentially reversing function arrows in the corecursion rule):

$$\frac{\Delta; \Gamma \vdash t \Leftarrow (\nu X{:}K.\, \Lambda\, \vec{u}.\, T)\, \vec{M}}{\Delta; \Gamma \vdash \mathtt{out}_\nu\, t \Leftarrow T[\nu X{:}K.\, \Lambda\, \vec{u}.\, T/X, \overrightarrow{M/u}]} \qquad \frac{\Delta, X{:}K; \Gamma, f{:}(\overrightarrow{u{:}U});\, T \to X\, \vec{u} \vdash t \Leftarrow (\overrightarrow{u{:}U});\, T \to S[\overrightarrow{u/v}]}{\Delta; \Gamma \vdash \mathtt{corec}\, f.\, t \Leftarrow (\overrightarrow{u{:}U});\, T \to (\nu X{:}K.\, \Lambda\, \vec{v}.\, S)\, \vec{u}}$$

The semantic interpretation of coinductive types is similarly dual to that of inductive types, relying on a semantic greatest fixed point operator. We would also need to modify our operational semantics to account for corecursion. Such an extension of Tores could serve as a core language into which we can compile proofs using coinduction and copatterns as described by Thibodeau et al. [2016].

There are still questions of how to compile a practical surface language into the core language we propose in Tores. We have not explained for example how to encode features such as mutually recursive definitions (including mixing of inductive and coinductive definitions), reconstruction of index arguments omitted by the programmer, and generation of unifying substitutions in equality elimination terms. Such issues are important to solve in order to create a productive user experience for dependently typed programming and proving.

Finally, it would be interesting to explore how our treatment of indexed recursive and stratified types could help (or hinder) proof search. Proof search is a fundamental technique to ease the development and maintenance of proofs, by automatically generating parts of proof terms. Like Baelde and Nadathur [2012], we are curious to see how our treatment of recursive definitions can be handled by search techniques, especially those derived from *focusing* [Andreoli, 1992]. It seems that the grounds for exploration with Tores are vast and fertile.

Bibliography

Andreas Abel. Termination checking with types. *RAIRO - Theoretical Informatics and Applications*, 38(4): 277–319, 3 2010.

Andreas Abel and Ralph Matthes. Fixed points of type constructors and primitive recursion. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *18th International Workshop on Computer Science Logic (CSL'04)*, volume 3210 of *Lecture Notes in Computer Science*, pages 190–204. Springer, 2004.

Andreas Abel, R. Matthes, and T. Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theoretical Computer Science*, 333:3–66, 2005. ISSN 03043975.

Robin Adams. *Formalized Metatheory with Terms Represented by an Indexed Family of Types*, pages 1–16. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-31429-5. doi: 10.1007/11617990_1. URL `http://dx.doi.org/10.1007/11617990_1`.

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.

Thorsten Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In Marc Bezem and Jan Friso Groote, editors, *International Conference on Typed Lambda Calculi and Applications (TLCA '93)*, volume 664 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 1993. ISBN 3-540-56517-5.

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

David Baelde. Least and greatest fixed points in linear logic. *ACM Transactions on Computational Logic*, 13(1):2:1–2:44, 2012.

David Baelde and Gopalan Nadathur. Combining deduction modulo and logics of fixed-point definitions. In *27th Annual IEEE Symposium on Logic in Computer Science (LICS'12)*, pages 105–114. IEEE, 2012.

Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly typed term representations in coq. *J. Autom. Reasoning*, 49(2):141–159, 2012. doi: 10.1007/s10817-011-9219-0. URL `http://dx.doi.org/10.1007/s10817-011-9219-0`.

Stefano Berardi. Girard normalization proof in LEGO. In *Proceedings of the First Workshop on Logical Frameworks*, pages 67–78, 1990.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

Andrew Cave and Brigitte Pientka. Programming with binders and indexed data-types. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 413–424. ACM Press, 2012.

Andrew Cave and Brigitte Pientka. First-class substitutions in contextual type theory. In *8th ACM SIGPLAN International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'13)*, pages 15–24. ACM Press, 2013.

Andrew Cave and Brigitte Pientka. A case study on logical relations using contextual types. In I. Cervesato and K.Chaudhuri, editors, *10th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'15)*, pages 18–33. Electronic Proceedings in Theoretical Computer Science (EPTCS), 2015.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 241–253, New York, NY, USA, 2005. ACM. ISBN 1-59593-064-7. doi: 10.1145/1086365.1086397. URL `http://doi.acm.org/10.1145/1086365.1086397`.

Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, pages 1–46, 2011. ISSN 0168-7433. 10.1007/s10817-011-9225-2.

James Cheney and Ralf Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.

Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, February 1988. ISSN 0890-5401. doi: 10.1016/0890-5401(88)90005-3. URL `http://dx.doi.org/10.1016/0890-5401(88)90005-3`.

Karl Crary and Stephanie Weirich. Flexible type analysis. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, ICFP '99, pages 233–248, New York, NY, USA, 1999. ACM. ISBN 1-58113-111-9. doi: 10.1145/317636.317906. URL `http://doi.acm.org/10.1145/317636.317906`.

Gilles Dowek and Benjamin Werner. Proof normalization modulo. *J. Symb. Log.*, 68(4):1289–1316, 2003. doi: 10.2178/jsl/1067620188. URL `http://dx.doi.org/10.2178/jsl/1067620188`.

Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72, 2003.

Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994. ISSN 1433-299X. doi: 10.1007/BF01211308. URL `http://dx.doi.org/10.1007/BF01211308`.

Herman Geuvers. A short and flexible proof of strong normalization for the calculus of constructions. In *Selected Papers from the International Workshop on Types for Proofs and Programs*, TYPES '94, pages 14–38, London, UK, UK, 1995. Springer-Verlag. ISBN 3-540-60579-7. URL `http://dl.acm.org/citation.cfm?id=646535.695848`.

Eduardo Gimenez. Structural recursive definitions in type theory. In *International Colloquium on Automata, Languages and Programming (ICALP'98)*, Lecture Notes in Computer Science (LNCS) 1443, pages 397–408. Springer, 1998. URL `citeseer.nj.nec.com/gimenez98structural.html`.

J. Y Girard. *Interprtation fonctionnelle et elimination des coupures de l'arithmtique d'ordre suprieur*. These d'tat, Universit de Paris 7, 1972.

Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 130–141, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. doi: 10.1145/199448.199475. URL `http://doi.acm.org/10.1145/199448.199475`.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479 – 490. Academic Press, 1980.

Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988. ISBN 0131103709.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629596. URL `http://doi.acm.org/10.1145/1629575.1629596`.

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

Per Martin-Löf. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, Amsterdam, 1973. North-Holland.

Ralph Matthes. *Extensions of system F by iteration and primitive recursion on monotone inductive types.* Herbert Utz Verlag, 1999.

Raymond C. McDowell and Dale A. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic*, 3(1):80–136, 2002. ISSN 1529-3785.

N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *Symposium on Logic in Computer Science (LICS'87)*, pages 30–36. IEEE Computer Society, 1987.

Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1):159 – 172, 1991. ISSN 0168-0072. doi: http://dx.doi.org/10.1016/0168-0072(91)90069-X. URL `http://www.sciencedirect.com/science/article/pii/016800729190069X`.

Nax Paul Francis Mendler. *Inductive Definition in Type Theory.* PhD thesis, Cornell University, Ithaca, NY, USA, 1988. AAI8804634.

Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1997. ISBN 0262631814.

Alberto Momigliano and Alwen Fernanto Tiu. Types for proofs and programs (types'03). In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Induction and Co-induction in Sequent Calculus*, volume 3085 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 2004.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):1–49, 2008.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science.* Springer, 2002.

Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007. Technical Report 33D.

Christine Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In Marc Bezem and Jan Friso Groote, editors, *International Conference on Typed Lambda Calculi and Applications(TLCA '93)*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993.

Simon Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report.* 2003.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *11th ACM SIGPLAN Int'l Conference on Functional Programming (ICFP '06)*, pages 50–61, September 2006.

Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 371–382. ACM Press, 2008.

Brigitte Pientka and Andreas Abel. Structural recursion over contextual objects. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*, pages 273–287. Leibniz International Proceedings in Informatics (LIPIcs) of Schloss Dagstuhl, 2015.

Brigitte Pientka and Andrew Cave. Inductive Beluga:Programming Proofs (System Description). In Amy P. Felty and Aart Middeldorp, editors, *25th International Conference on Automated Deduction (CADE-25)*, Lecture Notes in Computer Science (LNCS 9195), pages 272–281. Springer, 2015.

Brigitte Pientka and Joshua Dunfield. Programming with proofs and explicit contexts. In *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 163–173. ACM Press, 2008.

Andrew Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186 (2):165–193, November 2003. ISSN 0890-5401.

Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de bruijn terms and parallel substitutions. In Christian Urban and Xingyuan Zhang, editors, *6th International Conference of Interactive Theorem Proving (ITP)*, Lecture Notes in Computer Science (9236), pages 359–374. Springer, Aug 2015.

Peter Schroeder-Heister. Rules of definitional reflection. In *8th Annual Symposium on Logic in Computer Science (LICS '93)*, pages 222–232. IEEE Computer Society, 1993.

Jan M. Smith. Propositional functions and families of types. In *In Workshop on Programming Logic*, pages 140–159, 1989.

William Tait. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.*, 32(2):198–212, 1967.

David Thibodeau, Andrew Cave, and Brigitte Pientka. Indexed codata. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *21st ACM SIGPLAN International Conference on Functional Programming (ICFP'16)*, pages 351–363. ACM, 2016.

Alwen Tiu. Stratification in logics of definitions. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *6th International Joint Conference on Automated Reasoning (IJCAR'12)*, Lecture Notes in Computer Science (LNCS 7364), pages 544–558. Springer, 2012.

Alwen Tiu and Alberto Momigliano. Cut elimination for a logic with induction and co-induction. *J. Applied Logic*, 10(4):330–367, 2012. doi: 10.1016/j.jal.2012.07.007. URL `http://dx.doi.org/10.1016/j.jal.2012.07.007`.

Christian Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.

Benjamin Werner. A normalization proof for an impredicative type system with large elimination over integers. In *International Workshop on Types for Proofs and Programs (TYPES)*, pages 341–357, 1992.

A.N. Whitehead and B. Russell. *Principia Mathematica*. Number v. 2 in Principia Mathematica. University Press, 1912. URL `https://books.google.com.au/books?id=sbTVAAAAMAAJ`.

Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 249–257. ACM press, 1998.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, 1999.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *30th ACM Symposium on Principles of Programming Languages (POPL'03)*, pages 224–235. ACM Press, 2003. doi: 10.1145/604131.604150.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1120-5. doi: 10.1145/2103786.2103795. URL `http://doi.acm.org/10.1145/2103786.2103795`.

Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187(1-2):147–165, 1997.