Index-Stratified Types

Rohan Jacob-Rao
Digital Asset, Sydney, Australia
rohanjr@digitalasset.com

Brigitte Pientka
McGill University, Montreal, Canada
bpientka@cs.mcgill.ca

David Thibodeau
McGill University, Montreal, Canada
david.thibodeau@mail.mcgill.ca

Abstract

We present Tores, a core language for encoding metatheoretic proofs. The novel features we introduce are well-founded Mendler-style (co)recursion over indexed data types and a form of recursion over objects in the index language to build new types. The latter, which we call *index-stratified types*, are analogue to the concept of large elimination in dependently typed languages. These features combined allow us to encode sophisticated case studies such as normalization for lambda calculi and normalization by evaluation. We prove the soundness of Tores as a programming and proof language via the key theorems of subject reduction and termination.

2012 ACM Subject Classification Theory of computation → Type theory
Theory of computation → Logic and verification

Keywords and phrases Indexed types, (co)recursive types, logical relations

Digital Object Identifier 10.4230/LIPIcs.FSCD.2018.19

Related Version A long version of this paper with the full technical appendix is available at `https://arxiv.org/abs/1805.00401`.

Funding This research was funded by the Natural Science and Engineering Research Council Canada (NSERC).

1 Introduction

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. Dually, well-founded corecursion corresponds to coinductive reasoning. However, concentrating only on well-founded (co)recursive definitions is not sufficient to support the encoding of meta-theoretic proofs. There are two missing ingredients: 1) To express fine-grained properties we often rely on first-order logic which is analogous to *indexed types* in programming languages. 2) Many common notions cannot be directly characterized by well-founded (co)recursive definitions. An example is Girard's notion of reducibility for functions: a term M is reducible at type $A \to B$ if, for all terms N that are reducible at type A, we have that $M\ N$ is reducible at type B. This definition is well-founded because it is by structural recursion on the type indices (A and B), so we want to admit such definitions.

Our contribution in this paper is a core language called Tores that features indexed types and (co)inductive reasoning via well-founded (co)recursion. The primary forms of types are

indexed (co)recursive types, over which we support reasoning via Mendler-style (co)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 (co)recursive types are more flexible, allowing (co)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.

How to justify definitions that are recursively defined on a given index in addition to well-founded (co)recursive definitions has been explored in proof theory (see for example Tiu [2012], Baelde and Nadathur [2012]). While this line of work is more general, it is also more complex and further from standard programming practice. In dependent type theories, large eliminations achieve the same. Our approach, grounded in the Curry-Howard isomorphism, provides a complementary perspective on this problem where we balance expressiveness and ease of programming with a compact metatheory. We believe this may be an advantage when considering more sophisticated index languages and reasoning techniques.

The combination of indexed (co)recursive types and stratified types is already used 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 (co)inductive proofs.

2 Index Language for Tores

The design of Tores is parametric over an index language. Following Thibodeau et al. [2016] we stay as abstract as possible and state the general conditions the index language must satisfy. Whenever we require inspection of the particular index language, namely the structure of stratified types and induction terms, we will draw attention to it.

To illustrate the required structure for a concrete index language, we use natural numbers. In practice, however, we can consider other index languages such as those of strings, types [Cheney and Hinze, 2003, Xi et al., 2003], or (contextual) LF [Pientka, 2008, 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.

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

Tores relies on typing for index terms which we give for natural numbers in Fig. 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 and describe well-formed contexts. These definitions are generic.

$\boxed{\vdash \Delta\ \mathsf{ictx}}$ Index context Δ is well-formed $\quad$ $\boxed{\Delta \vdash U\ \mathsf{itype}}$ Index type U is well-kinded

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

$\boxed{\Delta \vdash M : U}$ 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 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.

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 an index term and $\overrightarrow{M}$ is a spine, then $M_0, \overrightarrow{M}$ is a spine. Similarly if $u_0{:}U_0$ is an index type assignment and $(\overrightarrow{u{:}U})$ is a type spine, then $(u_0{:}U_0, \overrightarrow{u{:}U})$ is a type spine. Spines are convenient for setting up the types and terms of Tores. Unlike index substitutions Θ which are built from right to left, spines are built from left to right.

Throughout our development we use both a single index substitution operation $M[N/u]$ and a simultaneous substitution operation $M[\Theta]$. For composition of simultaneous substitutions we write $\Theta_1[\Theta_2]$.

▶ 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 $M[\Theta_1][\Theta_2] = M[\Theta_1[\Theta_2]]$.

2.2 Unification and Matching

Type checking of Tores relies on 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. To illustrate, we show the unification rules for natural numbers. We write id_i for the identity substitution that maps index variables from Δ_i to themselves.

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

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

$$\frac{u \notin \mathrm{FV}(M) \quad \Delta = \Delta_1, u{:}U, \Delta_2 \quad \Delta' = \Delta_1, \Delta_2[M/u]}{\Delta \vdash u \doteqdot M \searrow (\Delta' \mid \mathsf{id}_1, M/u, \mathsf{id}_2)} \quad \text{(same for } M \doteqdot u\text{)}$$

$$\frac{}{\Delta \vdash 0 \doteqdot \mathsf{suc}\, M \searrow \#} \qquad \frac{}{\Delta \vdash \mathsf{suc}\, M \doteqdot 0 \searrow \#} \qquad \frac{u \in \mathrm{FV}(M) \quad M \neq u}{\Delta \vdash u \doteqdot M \searrow \#} \quad \text{(same for } M \doteqdot u\text{)}$$

▶ 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.

Finally, our operational semantics relies on 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 describe it using the judgment $\Delta \vdash M \doteq N \searrow (\Delta' \mid \Theta)$. The notion of matching also lifts to the level of index substitutions. We omit the full specifications here and instead state the required properties.

▶ 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 Specification of Tores

We now describe Tores, a programming language designed to express (co)inductive proofs and programs using Mendler-style (co)recursion. It also features *index-stratified* types, which allow definitions of types via well-founded recursion over indices.

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. If the spine of type declarations is empty then $(\cdot); T_1 \to T_2$ degenerates to the simple function space. We can also quantify existentially over an index using the type $\Sigma u{:}U.\, T$, and have a type for index equality

$M = N$. These two types are useful for expressing equality constraints on indices. We model (co)recursive and stratified types as type constructors of kind $\Pi\overrightarrow{u{:}U}.*$. These 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. For the index type nat, 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 $T_s[M/u, T_{\mathtt{Rec}}\,M/X]$. For richer index languages such as Contextual LF we can generate an appropriate recursion scheme following Pientka and Abel [2015].

Kinds	K	$::= * \mid \Pi u{:}U.\,K$
Types	T	$::= 1 \mid T_1 \times T_2 \mid T_1 + T_2 \mid (\overrightarrow{u{:}U});\, T_1 \to T_2 \mid \Sigma u{:}U.\,T \mid M = N$ $\mid T\,M \mid \Lambda\, u.\,T \mid X \mid \mu X{:}K.\,T \mid \nu X{:}K.\,T \mid T_{\mathtt{Rec}}$
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	$::= \cdot \mid \Gamma, x{:}T$

▶ **Example 1.** We illustrate indexed recursive types and stratified types using vectors, i.e. lists indexed by their length, with elements of type A. Vectors are of kind $\Pi n{:}\mathsf{nat}.*$. We omit the kind annotation for better readability in the subsequent type definitions. One way to define vectors is with an indexed recursive type, an explicit equality and an existential type: $\mathsf{Vec}_\mu \equiv \mu V.\,\Lambda\, n.\, n = 0 + \Sigma m{:}\mathsf{nat}.\, n = \mathsf{suc}\, m \times (A \times V\, m)$.

Alternatively, they can be defined as a stratified type: $\mathsf{Vec}_S \equiv \mathtt{Rec}\,(0 \mapsto 1 \mid \mathsf{suc}\, m,\, V \mapsto A \times V)$. In this case equality reasoning is implicit. While we have a choice how to define vectors, some types are only possible to encode using one form or the other.

▶ **Example 2.** A type that must be stratified is the encoding of reducibility for simply typed lambda terms. This example is explored in detail by Cave and Pientka [2013]; our work gives it theoretical justification.

Here the index objects are the simple types, $\mathtt{unit}$ and $\mathtt{arr}\ A\ B$ of index type $\mathtt{tp}$, as well as lambda terms $()$, $\mathtt{lam}\ x.M$ and $\mathtt{app}\ M\ N$ of index type $\mathtt{tm}$. We can define reducibility as a stratified type of kind $\Pi a{:}\mathtt{tp}.\Pi m{:}\mathtt{tm}.*$. This relies on an indexed recursive type $\mathtt{Halt}$ (omitted here) that describes when a term m steps to a value.

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

▶ **Example 3.** To illustrate a corecursive type, we define an indexed stream of bits following Thibodeau et al. [2016]. The index here guarantees that we are reading exactly m bits. Once $m = 0$, we read a new message consisting of the length of the message n together with a stream indexed by n. In contrast to the recursive type definition for vectors, here the equality constraints guard the observations we can make about a stream.

$$\begin{aligned}\mathsf{Stream} \equiv \nu\mathrm{Str}.\,\Lambda\, m.\quad &(\cdot);\, m = 0 && \to \Sigma n{:}\mathsf{nat}.\,\mathrm{Str}\ n\\ &\times (n{:}\mathsf{nat});\, m = \mathsf{suc}\, n && \to \mathrm{Str}\ n\\ &\times (n{:}\mathsf{nat});\, m = \mathsf{suc}\, n && \to \mathrm{Bit}\end{aligned}$$

3.2 Terms

Tores contains many common constructs found in functional programming languages, such as unit, pairs and case expressions. We focus on the less standard constructs: indexed functions, equality witnesses, well-founded recursion and index induction.

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

Since we combine the dependent and simple function types in $(\overrightarrow{u{:}U});\ T_1 \to T_2$, we similarly combine abstraction over index variables $\vec{u}$ and a term variable x in our function term $\lambda\, \vec{u}, x.\, t$. The corresponding application form is $t\, \vec{M}\, s$. 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 `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.

Recursive types are introduced by the "fold" syntax $\mathtt{in}_\mu$, and stratified types are introduced by $\mathtt{in}_l$ terms. Here l ranges over constructors in the index language such as 0 and suc. The important difference is how we eliminate recursive and stratified types. We can analyze data defined by a recursive type using Mendler-style recursion $\mathtt{rec}\, f.\, t$. This gives a powerful means of recursion 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. While this induction principle is specific to natural numbers, it can also be derived for other index domains, in particular contextual LF (see Pientka and Abel [2015]).

▶ **Example 4.** Recall that vectors can be defined using the indexed recursive type Vec_μ or the stratified type Vec_S. Which definition we choose impacts how we write programs that analyze vectors. We show the difference using a recursive function that copies a vector.

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

To analyze the recursively defined vector, we use recursion and case analysis of the input vector to reconstruct the output vector. If 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.

To contrast we show the program using induction on natural numbers and unfolding the stratified type definition of Vec_S. Note that the first argument is the natural number index n paired with a unit term argument, since index abstraction is always combined with term abstraction. The program analyzes n and in the suc case unfolds the input vector before reconstructing it using the result of the recursive call. In this version of *copy* the equality

constraints are handled silently by the type checker.

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

▶ **Example 5.** Note that Tores does not have an explicit notion of falsehood. This is because it is definable using existing constructs: we can define the empty type as a recursive type $\bot \equiv \mu X{:}*.\,X$, and a contradiction term $\mathtt{abort} \equiv \mathtt{rec}\, f.\, f : \bot \to C$, for any type C. Our termination result with the logical relation in Section 4.3 shows 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 Typing Rules

We define a bidirectional type system in Fig. 2. We focus here on equality, recursive and stratified types.

The introduction for an index equality type is simply `refl`, which is checked via equality in the index domain. Both equality elimination forms rely on unification in the index domain (see Section 2.2). Specifically, the $\mathtt{eq\_abort}\, s$ term checks against any type because the unification must fail, establishing a contradiction. For the term $\mathtt{eq}\, s\, \mathtt{with}\, (\Delta'.\Theta \ \mapsto\ t)$, unification must result in the MGU which by Req. 4 is α-equivalent to the supplied unifier $(\Delta' \mid \Theta)$. We then check the body t using the new index context Δ' and Θ applied to the contexts Ξ and Γ and the goal type T.

This treatment of equality elimination is similar to the use of refinement substitutions for dependent pattern matching [Pientka and Dunfield, 2008, Cave and Pientka, 2012], and is 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* (which may be infinite!), instead of a single most general unifier. We believe our requirement for a unique MGU is a practical choice for type checking.

Indexed recursive and stratified types are both introduced by injections ($\mathtt{in}_\mu$ and $\mathtt{in}_l$), though their elimination forms are different. Stratified types are eliminated (unfolded) in reverse to the corresponding fold rules. For recursive types on the other hand, the naive unfold rules lead to nontermination, so we use a Mendler-style recursion form $\mathtt{rec}\, f.\, t$, generalizing the 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 premise of the rule. Here X represents types exactly one constructor smaller than the recursive type, so the use of f is guaranteed to be well-founded.

▶ **Theorem 6.** *Type checking of terms is decidable.*

Proof. Since the typing rules are syntax directed, it is straight-forward to extract a type checking algorithm. Note that the algorithm relies on decidability of judgments in the index language, namely index type checking (Req. 1), equality (Req. 2) and unification (Req. 4). ◀

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\Delta;\Xi;\Gamma \vdash t \Rightarrow (\nu X{:}K.\, \Lambda\, \vec{u}.\, T)\, \vec{M}}{\Delta;\Xi;\Gamma \vdash \mathtt{out}_\nu\, t \Rightarrow T[\overrightarrow{M/u}; \nu X{:}K.\, \Lambda\, \vec{u}.\, T/X]}$$

Figure 2 Typing rules for Tores

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 term.

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

Values consist of unit, pairs, injections, reflexivity, and closures. Typing for values and environments, which is used to state the subject reduction theorem, are given in the appendix.

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 (lambdas, recursion and induction) uniformly. Similarly, we factor out the application of $\mathtt{out}_\nu$ to a closure c in an additional judgment written $c \cdot_{\mathtt{out}_\nu} \Downarrow w$. This simplifies the type interpretation used to prove termination.

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

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

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

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

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

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

$$\frac{}{(\mathtt{corec}\, f.\, t)[\theta;\sigma] \Downarrow (\mathtt{corec}\, f.\, t)[\theta;\sigma]} \qquad \frac{t[\theta;\sigma] \Downarrow c \quad c \cdot_{\mathtt{out}_\nu} \Downarrow w}{(\mathtt{out}_\nu\, t)[\theta;\sigma] \Downarrow w}$$

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

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

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

$$\frac{}{(\mathtt{corec}\, f.\, t)[\theta;\sigma] \cdot \vec{N}\, v \Downarrow (\mathtt{corec}\, f.\, t)[\theta;\sigma] \cdot \vec{N}\ v}$$

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

$\boxed{c \cdot_{\mathtt{out}_\nu} \Downarrow w}$ Closure c applied to observation $\mathtt{out}_\nu$ evaluates to w

$$\frac{t[\theta;\sigma, (\mathtt{corec}\, f.\, t)[\theta;\sigma]/f] \Downarrow c \quad c \cdot \vec{N}\, v \Downarrow w}{((\mathtt{corec}\, f.\, t)[\theta;\sigma] \cdot \vec{N}\ v) \cdot_{\mathtt{out}_\nu} \Downarrow w}$$

Figure 3 Big-step evaluation rules

We only explain the evaluation rule for equality elimination $\texttt{eq}\, s\, \texttt{with}\, (\Delta.\Theta \mapsto t)$. We first evaluate the equality witness s under environments $\theta; \sigma$ to the value $\texttt{refl}$. 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. This relies on sound and complete index substitution matching (see Section 2.2) 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.

Notably absent is an evaluation rule for $\texttt{eq\_abort}\, t$. This term is used in a branch of a case split that we know statically to be impossible. Such branches are never reached at run time, so there is no need for an evaluation rule. For example, consider a type-safe "head" function, which receives a nonempty vector as input. As we write each branch of a case split explicitly, the empty list case must use $\texttt{eq\_abort}\, t$, but is never executed. We now state subject reduction for Tores.

▶ **Theorem 7** (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* $w : T[\theta, \overrightarrow{N/u}]$.
3. *If* $c \cdot_{out_\nu} \Downarrow w$ *where* $c : (\nu X{:}K.\, \Lambda\, \vec{u}.\, T)\ \vec{M}$ *then* $w : T[\overrightarrow{M/u}; (\nu X{:}K.\, \Lambda\, \vec{u}.\, T)/X]$.

4 Termination Proof

We now describe our main technical result: termination of evaluation. Our proof uses the logical predicate technique of Tait [1967] and Girard [1972]. We interpret each language construct (index types, kinds, types, etc.) into a semantic model of sets and functions.

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. Hence our interpretation for index types U must take into account an environment θ containing instantiations for index variables u. Such an index environment θ is simply a grounding substitution $\vdash \theta : \Delta$.

▶ **Definition 8** (Interpretation of index types $[\![U]\!]$ and index spines $[\![\overrightarrow{u{:}U}]\!]$).

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

The interpretation of an index type U under environment θ is the set of closed terms of type $U[\theta]$. The interpretation lifts to index spines $(\overrightarrow{u{:}U})$. With these definitions, the following lemma follows from the substitution principles of index terms (Req. 3).

▶ **Lemma 9** (Interpretation of index substitution).

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

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

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

A key observation in our metatheory is that each $[\![K]\!](\theta)$ forms a *complete lattice*. In the base case, $[\![*]\!](\theta) = \mathcal{P}(\Omega)$ is a complete lattice under the subset ordering, with meet and join given by intersection and union respectively. For a kind $K = \Pi u{:}U.\, K'$, we induce a lattice structure on $[\![K]\!](\theta)$ 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. Here we rely on the existence of arbitrary meets, as we take the meet over an impredicatively defined subset of $\mathcal{L}$.

▶ **Definition 11** (Mendler-style pre-fixed and post-fixed points). 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

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

and $\boldsymbol{\nu}_{\mathcal{L}} : (\mathcal{L} \to \mathcal{L}) \to \mathcal{L}$ by

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

We will mostly omit the subscript denoting the underlying lattice $\mathcal{L}$ of the order $\leq$ and pre-fixed and post-fixed points, $\boldsymbol{\mu}$ and $\boldsymbol{\nu}$.

Note that a usual treatment of recursive types would define the least pre-fixed point of a monotone operator as $\bigwedge\{\mathcal{C} \in \mathcal{L} \mid \mathcal{F}(\mathcal{C}) \leq \mathcal{C}\}$ and the greatest post-fixed point of a monotone operator as $\bigvee\{\mathcal{C} \in \mathcal{L} \mid \mathcal{C} \leq \mathcal{F}(\mathcal{C})\}$, using the Knaster-Tarski theorem. However, our unconventional definition (following Jacob-Rao et al. [2016]) more closely models Mendler-style (co)recursion and does not require $\mathcal{F}$ to be monotone (thereby avoiding a positivity restriction on recursive types).

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 12** (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 $\mathtt{in}^* : [\![K]\!](\theta) \to [\![K]\!](\theta)$ 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.

Dually we define $\mathtt{out}^*_\nu : [\![K]\!](\theta) \to [\![K]\!](\theta)$. If $\mathcal{V} \in [\![*]\!](\theta) = \mathcal{P}(\Omega)$ then $\mathtt{out}^*_\nu\,\mathcal{V} = \mathtt{out}_\nu\,\mathcal{V} = \{c \mid c \cdot_{\mathtt{out}_\nu} \Downarrow w \in \mathcal{V}\}$. If $\mathcal{C} \in [\![\Pi u{:}U.\,K']\!](\theta)$ then $(\mathtt{out}^*_\nu\,\mathcal{C})(M) = \mathtt{out}^*_\nu\,(\mathcal{C}(M))$ for all $M \in [\![U]\!](\theta)$.

Finally, we 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 13** (Interpretation of type variable contexts $[\![\Xi]\!]$).

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

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 14** (Interpretation of types and constructors).

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

where

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

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 $[\![\mathsf{nat}]\!] = \mathbb{N}$.

Last, we give the interpretation for typing contexts Γ, describing well-formed term-level environments σ.

▶ **Definition 15** (Interpretation of typing contexts).

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

4.4 Proof

We now sketch our proof using some key lemmas. The following two lemmas concern the fixed point operators $\boldsymbol{\mu}$ and $\boldsymbol{\nu}$, and are key for reasoning about (co)recursive types and Mendler-style (co)recursion. These lemmas generalize those of Jacob-Rao et al. [2016] from the simply typed setting.

▶ **Lemma 16** (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. 11. Then $\mathcal{F}(\boldsymbol{\mu}\mathcal{F}) \leq \boldsymbol{\mu}\mathcal{F}$.*

▶ **Lemma 17** (Function space from pre-fixed and post-fixed points). *Let $\mathcal{L} = \vec{\mathcal{U}} \to \mathcal{P}(\Omega)$ and $\mathcal{B} \in \mathcal{L}$ and $\mathcal{F} : \mathcal{L} \to \mathcal{L}$.*

1. *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 $c \in \vec{\mathcal{U}}, \boldsymbol{\mu}\mathcal{F} \to \mathcal{B}$.*
2. *If $\forall \mathcal{X} \in \mathcal{L}.\, c \in \vec{\mathcal{U}}, \mathcal{B} \to \mathcal{X} \implies c \in \vec{\mathcal{U}}, \mathcal{B} \to \mathcal{F}\mathcal{X}$, then $c \in \vec{\mathcal{U}}, \mathcal{B} \to \boldsymbol{\nu}\mathcal{F}$.*

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 18** (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 19** (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 $\boldsymbol{in}_\mu\, [\![S[\overrightarrow{M/u}; R/X]]\!](\theta;\eta) \subseteq [\![R\, \vec{M}]\!](\theta;\eta)$.

▶ **Lemma 20** (Backward closure).
Let t be a term, θ and σ environments, and $\mathcal{A}, \mathcal{B} \in \vec{\mathcal{U}} \to \mathcal{P}(\Omega)$.

1. *If $t[\theta; \sigma, (\mathtt{rec}\, f.\, t)[\theta;\sigma]/f] \Downarrow c' \in \vec{\mathcal{U}}, \mathcal{A} \to \mathcal{B}$, then $(\mathtt{rec}\, f.\, t)[\theta;\sigma] \in \vec{\mathcal{U}}, \boldsymbol{in}^*_\mu\, \mathcal{A} \to \mathcal{B}$.*
2. *If $t[\theta; \sigma, (\mathtt{corec}\, f.\, t)[\theta;\sigma]/f] \Downarrow c' \in \vec{\mathcal{U}}, \mathcal{A} \to \mathcal{B}$, then $(\mathtt{corec}\, f.\, t)[\theta;\sigma] \in \vec{\mathcal{U}}, \mathcal{A} \to \boldsymbol{out}^*_\nu\, \mathcal{B}$.*

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 21** (Stratified types equivalent to unfolding).
Let $T_{\mathsf{Rec}} \equiv \mathsf{Rec}_K\,(0 \mapsto T_z \mid \mathsf{suc}\,n,\ X \mapsto T_s)$ *where* $K = \Pi n\colon \mathsf{nat}\,.\,\Pi\overrightarrow{u{:}U}.*$ *and* $\Delta;\Xi \vdash T_{\mathsf{Rec}} \Rightarrow K$*, 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. $[\![T_{\mathsf{Rec}}\,0\,\vec{M}]\!](\theta;\eta) = \boldsymbol{in}_0\,([\![T_z\,\vec{M}]\!](\theta;\eta))$ *and*
2. $[\![T_{\mathsf{Rec}}\,(\mathsf{suc}\,N)\,\vec{M}]\!](\theta;\eta) = \boldsymbol{in}_{\mathsf{suc}}\,([\![T_s[N/n;(T_{\mathsf{Rec}}\,N)/X]\,\vec{M}]\!](\theta;\eta))$.

Finally we state the main termination theorem.

▶ **Theorem 22** (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* $v \in [\![T]\!](\theta;\eta)$.

5 Related Work

Our work draws inspiration from two different areas: dependent type theory and proof theory.

Dependent type theories often support large eliminations that are definitions of dependent types by primitive recursion. They reduce the same way as term-level recursion. In general, our work shows how to gain the power of large eliminations in an indexed type system by simulating reduction on the level of types by unfolding stratified types in their typing rules.

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 which implies consistency of the logics. Their cut elimination proof extends Girard's proof technique of reducibility candidates, similar to ours. 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. 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. Although their notion of stratification of recursive definitions is slightly more general than ours, 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.

From a programmer's view, the proof theoretic foundations give rise to programs written using iterators; our use of Mendler-style (co)recursion is arguably closer to standard programming practice. Mendler-style recursion schemes for term-indexed languages have been investigated by Ahn [2014]. Ahn 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.

6 Conclusion and Future Work

We presented a core language Tores extending an indexed type system with (co)recursive types and stratified types. We argued that Tores provides a sound and powerful foundation for programming (co)inductive proofs, in particular those involving logical relations. This power comes from the (co)induction principles on recursive types given by Mendler-style (co)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).

An important question to investigate in the future is how to compile a practical language that supports (co)pattern matching into the core language we propose in Tores. Similarly it would also be interesting to explore how our treatment of indexed recursive and stratified types could help (or hinder) proof search. Such issues are important to solve in order to create a productive user experience for dependently typed programming and proving.

Acknowledgements

We thank Andrew Cave for the idea of stratified types and for guiding the initial development.

References

Andreas Abel and Ralph Matthes. Fixed points of type constructors and primitive recursion. In *18th Int'l Workshop on Computer Science Logic (CSL'04)*, Lecture Notes in Computer Science (LNCS 3210), pages 190–204. Springer, 2004.

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.

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

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

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

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

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

Gilles Dowek and Benjamin Werner. Proof normalization modulo. *J. Symb. Log.*, 68(4): 1289–1316, 2003.

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

J. Y Girard. *Interprétation fonctionnelle et elimination des coupures de l'arithmétique d'ordre supérieur*. These d'état, Université de Paris 7, 1972.

Rohan Jacob-Rao, Andrew Cave, and Brigitte Pientka. Mechanizing proofs about Mendler-style recursion. In *11th Int'l Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'16)*, pages 1 – 9. ACM, 2016.

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.

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

Alberto Momigliano and Alwen Fernanto Tiu. Induction and co-induction in sequent calculus. In *Types for Proofs and Programs (TYPES'03)*, Lecture Notes in Computer Science (LNCS 3085), 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.

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

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

Brigitte Pientka and Joshua Dunfield. Programming with proofs and explicit contexts. In *35th ACM Symp. on Principles and Practice of Declarative Programming (PPDP'08)*, pages 163–173. ACM, 2008.

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

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 *21st ACM Int'l Conf. on Functional Programming (ICFP'16)*, pages 351–363. ACM, 2016.

Alwen Tiu. Stratification in logics of definitions. In *6th Int'l Joint Conf. 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.

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

A Appendix

A.1 Kinding

We employ a bidirectional kinding system to show when kinds can be inferred and when kinding annotations are necessary. The judgments use two contexts: an index context Δ and a type variable context Ξ. Note that in general the kinds assigned to type variables in Ξ may depend on index variables in Δ.

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

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

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

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

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

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

$$\frac{\Delta;\Xi, X{:}K \vdash T \Leftarrow K}{\Delta;\Xi \vdash \mu X{:}K.\, T \Rightarrow K} \qquad \frac{\Delta;\Xi, X{:}K \vdash T \Leftarrow K}{\Delta;\Xi \vdash \nu X{:}K.\, T \Rightarrow K}$$

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

Figure 4 Kinding rules for Tores

A.2 Value Typing

Values are the results of evaluation. Note that values are closed, and hence their typing judgment does not require a context. However, closures do contain terms (typed with the main typing judgment) and environments (typed against the contexts Δ and Γ).

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

$$\frac{\cdot \vdash \theta : \Delta \quad \sigma : \Gamma[\theta] \quad \Delta; \cdot; \Gamma \vdash g \Leftarrow T}{(g)[\theta;\sigma] : T[\theta]} \qquad \frac{}{\langle\rangle : 1} \qquad \frac{\cdot \vdash M = N}{\mathtt{refl} : M = N} \qquad \frac{v_1 : T_1 \quad v_2 : T_2}{\langle v_1, v_2\rangle : T_1 \times T_2}$$

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

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

$$\frac{\cdot \vdash \theta : \Delta \quad \sigma : \Gamma[\theta] \quad \cdot \vdash \vec{N} : \vec{U} \quad v : S[\theta, \overrightarrow{N/u}] \quad \Delta; \cdot, X{:}K; \Gamma, f{:}((\overrightarrow{u{:}U});\, S \to X\vec{u}) \vdash t \Leftarrow (\overrightarrow{u{:}U});\, S \to T[\overrightarrow{u/u'}]}{(\mathtt{corec}\, f.\, t)[\theta;\sigma] \cdot \vec{N}\ v : (\nu X{:}K.\, \Lambda\, \overrightarrow{u'}.\, T)[\theta]\ \vec{N}}$$

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

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

Figure 5 Value and environment typing