System F_i

a higher-order polymorphic λ-calculus with erasable term indices

Ki Yung Ahn Tim Sheard
Portland State University
{kya,sheard}@cs.pdx.edu

Marcelo Fiore Andrew M. Pitts
University of Cambridge
{Marcelo.Fiore,Andrew.Pitts}@cl.cam.ac.uk

Abstract

The purpose of this paper is to introduce a foundational type system, System F_i, for the design of programming languages with first-class term-indexed datatypes – higher-order datatypes whose parameters range over data such as Natural Numbers Lists.

To do this, we have devised a minimal extension of System F_ω that incorporates term indices. While term-indexed datatypes are expressible in rich type theories, like the Implicit Calculus of Constructions (ICC), these systems typically come coupled with orthogonal features such as large eliminations and full type dependency. We argue that there are important pedagogical benefits of isolating the minimal features to support term-indexing. We show that System F_i provides a theory for analysing programs with term-indexed types and also argue that it constitutes a basis for the design of light-weight logically-sound dependent programming languages.

In terms of expressivity, System F_i sits in between System F_ω (the prototypical logical calculus for functional programming) and ICC (a full-featured dependent type theory). Indeed, we relate System F_i to System F_ω and ICC as follows. We establish erasure properties of F_i-types that capture the idea that term indices are discardable in that they are irrelevant for computation. Index erasure projects typing in System F_i to typing in System F_ω; so System F_i inherits the strong-normalisation property from System F_ω. The logical consistency of System F_i is established by embedding it into a subset of ICC.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—data types and structures; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—lambda calculus and related systems

General Terms Languages, Theory

Keywords term-indexed data types, generalized algebraic data types, higher-order polymorphism, type-constructor polymorphism, higher-kinded types, impredicative encoding

1. Introduction

We wish to incorporate dependent types into ordinary programming languages. We are interested in two kinds of dependent types. Full dependency, where the type of a function can depend upon the value of its run-time parameters, and static dependency, where the type of a function can depend only upon static (or compile-time) parameters. Static dependency is sometimes referred to as indexed typing. The first is very expressive, while the second is often much easier to learn and use. Indexed types come in two flavors: type-indexed and term-indexed types. Type indexing includes parametric polymorphism, but it also includes more sophisticated typing such as Generalized Algebraic Datatypes (GADTs). An example of type indexing using GADTs is a type representation:

```
data TypeRep t where
  Int :: TypeRep Int
  Bool:: TypeRep Bool
  Pair:: TypeRep a -> TypeRep b -> TypeRep(a,b)
```

Here, a value of type `TypeRep t` is isomorphic in "shape" with the type `t`. For example `(Pair Int Bool)` is isomorphic in shape with its type `(Int,Bool)`.

On the other hand, term-indexed types include indices that range over data structures, such as Natural Numbers (like `Z`, `(S Z)`) or Lists (like `Nil` or `(Cons Z Nil)`). The classic example of a term index is the second parameter to the length-indexed list type `Vec` (as in `(Vec Int (S Z))`). In languages such as Haskell, which support GADTs with type indexing, term-indices are not first-class; they are "faked" by reflecting data at the type level with uninhabited type constructors. For example,

```
data Zero
data Succ n

data Vector t n where
  Nil  :: Vector a Zero
  Cons :: a -> Vector a n -> Vector a (Succ n)
```

This comes with a number of problems. First, there is no way to say that types such as `(Succ Int)` are ill-formed, and second the costs associated with duplicating the constructor functions of data to be used as term-indices. Nevertheless, "faked" term-indexed GADTs have become extremely popular as a light-weight, type-based mechanism to raise the confidence of users that software systems maintain important properties.

A salient example is Guillemette's thesis [9] encoding the classic paper by Morrisett et al. [15] completely in Haskell. This impressive system embeds a multi-stage compiler, from System F all the way to typed assembly language using indexed datatypes (many of them "faked" term-indices) to show that every stage preserves type information. As such, it provides confidence but no guarantees. Indeed, since in Haskell the non-terminating computation can be assigned any type, it is in principle possible that the type-preservation property is a consequence of a non-terminating computation in the program code.

This drawback is absent in approaches based on strongly normalizing logical calculi; like, for instance, System F_ω, the higher-order polymorphic lambda calculus, which is rich enough to express a wide collection of data structures. Unfortunately, the *term-indexed datatypes* that are necessary to support Guillemette's system are not known to be expressible in System F_ω.

In his CompCert system, Leroy [10] showed that the much richer logical Calculus of Inductive Constructions (CIC), which constitutes the basis of the Coq proof assistant, is expressive enough to guarantee type preservation (and more) between compiler stages. This approach, however, comes at a cost. Programmers must learn to use both dependent types and a new programming paradigm, programming by code extraction.

Some natural questions thus arise: Is there an expressive system supporting term-indexed types, say, sitting somewhere in between F_ω and fully dependent calculi? If only term-indexed types are needed to maintain properties of interest, is there a language one can use? Can one program, rather than extract code? The goal of this paper is to develop the theory necessary to begin answering these and related questions.

Our approach in this direction is to design a new foundational calculus, System F_i, for functional programming languages with term-indexed datatypes. In a nutshell, System F_i is obtained by minimally extending System F_ω with type-indexed kinds. Notably, this yields a logical calculus that is expressive enough to embed non-dependent *term-indexed datatypes* and their eliminators. Our contributions in this development are as follows.

- Identifying the features that are needed for a higher-order polymorphic λ-calculus to embed term-indexed datatypes (§2), in isolation from other features normally associated with such calculi (e.g., general recursion, large elimination, dependent types).
- The design of the calculus, System F_i (§3), and its use to study properties of languages with term-indexed datatypes, by embedding these into the calculus (§4). For instance, one can use System F_i to prove that the Mendler-style eliminators for GADTs of [3] are normalizing.
- Showing that System F_i enjoys a simple erasure property and inherits meta-theoretic results (strong normalization and logical consistency) from well-known calculi (System F_ω and ICC) that enclose System F_i (§5).

2. From System F_ω to System F_i, and back

It is well known that datatypes can be embedded into polymorphic lambda calculi by means of functional encodings (e.g., [1]), such as the Church and Boehm-Berarducci encodings.

In System F, for instance, one can embed *regular datatypes* [4], like homogeneous lists:

Haskell:	`data List a = Nil	Cons a (List a)`
System F:	$\texttt{List}\ A \ \triangleq\ \forall X. X \to (A \to X \to X) \to X$

In such regular datatypes, constructors have algebraic structure that directly translates into polymorphic operations on abstract types as encapsulated by universal quantification.

In the more expressive System F_ω, one can encode more general *type-indexed datatypes* that go beyond the algebraic class. For example, one can embed powerlists with heterogeneous elements in which an element of type `a` is followed by an element of the product type `(a,a)`:

Haskell: `data Powl a = PNil | PCons a (Powl(a,a))`

System F_ω: $\texttt{Powl} \ \triangleq\ \lambda A^{*}.\forall X^{*\to *}.$
$$X A \to (A \to X(A \times A) \to X A) \to X A$$

Note the non-regular occurrence (`Powl(a,a)`) in the definition of (`Powl a`), and the use of universal quantification over higher-order kinds.

What about term-indexed datatypes? What extension to System F_ω is needed to embed these, as well as type-indexed ones? Our answer is System F_i.

In a functional language supporting term-indexed datatypes, we envisage that the classic example of homogeneous length-indexed lists would be defined along the following lines:

```
data Nat = Z | S Nat
data Vec (a:*) {i:Nat} where
  VNil  : Vec a {Z}
  VCons : a -> Vec a {i} -> Vec a {S i}
```

Here the type constructor `Vec` is defined to admit parameterisation by both type and term indices. For instance, the type (`Vec (List Nat) {S (S Z)}`) is that of two-dimensional vectors of lists of natural numbers. By design, our syntax directly reflects the different type and term indexing by indicating the latter in curly brackets. This feature has been directly transferred from System F_i, where it is used as a mechanism for guaranteeing the static nature of term indexing.

The encoding of the vector datatype in System F_i is as follows:

$$\texttt{Vec} \ \triangleq\ \lambda A^{*}.\lambda i^{\texttt{Nat}}.\forall X^{\texttt{Nat}\to *}.$$
$$X\{\texttt{Z}\} \to (\forall j^{\texttt{Nat}}.A \to X\{j\} \to X\{\texttt{S}\ j\}) \to X\{i\}$$

where $\texttt{Nat} \triangleq \forall X^{*}.\, X \to (X \to X) \to X$, $\texttt{Z} \triangleq \lambda x.\,\lambda f.\,x$, and $\texttt{S} \triangleq \lambda n.\,\lambda x.\,\lambda f.\,n\,x\,f$. Without going into the details of the formalism, which are given in the next section, one sees that such a calculus incorporating term-indexing structure needs four additional constructs.

1. Type-indexed kinding ($A \to \kappa$) (as in ($\texttt{Nat}\to *$) in the example above) where the compile-time nature of term-indexing will be reflected in the enforcement that A be a closed type (rule (Ri) in Figure 1).
2. Term-index abstraction $\lambda i^{A}.F$ (as $\lambda i^{\texttt{Nat}}.\cdots$ in the example above) for constructing (or introducing) type-indexed kinds (rule (λi) in Figure 1).
3. Term-index application $F\{s\}$ (as $X\{\texttt{Z}\}$, $X\{j\}$, and $X\{\texttt{S}\ j\}$ in the example above) for destructing (or eliminating) type-indexed kinds, where the compile-time nature of indexing will be reflected in the enforcement that the index be statically typed (rule $(@i)$ in Figure 1) .
4. Term-index polymorphism $\forall i^{A}.B$ (as $\forall j^{\texttt{Nat}}.\cdots$ in the example above) where the compile-time nature of polymorphic term-indexing will be reflected in the enforcement that the variable i be static of closed type A (rule $(\forall Ii)$ in Figure 1).

As exemplified above, System F_i maintains a clear-cut separation between higher-order kinding and term indexing. This adds a level of abstraction to System F_ω and yields types that in addition to structural invariants also keep track of indexing invariants. Being static, all term-index information can be erased. This projects System F_i into System F_ω fixing the latter. For instance, the erasure of the F_i-type `Vec` is the F_ω-type `List`, the erasure of which (when regarded as an F_i-type that is) is in turn itself. Since, as already mentioned, typing in System F_i imposes structural and indexing constraints on terms one expects that the structural projection from System F_i to System F_ω provided by index erasure preserves typing. This is established in §5 and used to deduce the strong normalization of System F_i.

3. System F_i

System F_i is a higher-order polymorphic lambda calculus designed to extend System F_ω by the inclusion of term indices. The syntax and rules of System F_i are described in Figures 1 and 2. The extensions new to System F_i, which are not originally part of System F_ω, are highlighted by grey boxes. Eliding all the grey boxes from Figures 1 and 2, one obtains a Curry-style version of System F_ω with terms and typing contexts separated into two parts, type and term level contexts.

In this section, we first discuss the rational for our design choices (§3.1) and then introduce the new constructs of System F_i (§3.2).

3.1 Design of System F_i

Terms in F_i are Curry style. That is, term level abstractions are unannotated ($\lambda x.t$), and type generalizations ($\forall I$) and type instantiations ($\forall E$) are implicit at term level. A Curry-style calculus generally has an advantage over its Church-style counterpart when reasoning about properties of reduction. For instance, the Church-Rosser property naturally holds for β-, η-, and $\beta\eta$-reduction in the Curry style, but may not hold in the Church style. This is due to the presence of annotations in abstractions [13].[1]

Type constructors, on the other hand, remain Church style in F_i. That is, type level abstractions are annotated by kinds ($\lambda X^\kappa.F$). Choosing type constructors to be Church style makes the kind of a type constructor visually explicit. The choice of style for type constructors is not as crucial as the choice of style for terms, since the syntax and kinding rules at type level are essentially a simply typed lambda calculus. Annotating the type level abstractions with kinds makes kinds explicit in the type syntax. Since F_i is essentially an extension of F_ω with a new formation rule for kinds, making kinds explicit is a pedagogical tool to emphasize the consequences of this new formation rule. As a notational convention, we write A and B, instead of F and G, where A and B to are expected to be types (i.e., nullary type constructors) of kind $*$.

In a language with term indices, terms appear in types (e.g., the length index $(n+m)$ in the type *Vec Nat* $\{n+m\}$). Such terms contain variables. The binding sites of these variables matter. In F_i, we expect such variables to be statically bound. Dynamically bound index variables would require a dependently typed calculus, such as the calculus of constructions. To reflect this design choice, typing contexts are separated into type level contexts (Δ) and term level contexts (Γ). Type level (static) variables (X , i) are bound in Δ and term (dynamic) variables (x) are bound in Γ. Type level variables are either type constructor variables (X) or term variables to be used as indices (i). As a notational convention, we write i, instead of x, when term variables are to be used as indices (i.e., introduced by either index abstraction or index polymorphism).

In contrast to our design choice, System F_ω is most often formalized using a single context, which binds both type variables (X) and term variables (x). In such a formalization, the free type variables in the typing of the term variable must be bound earlier in the context. For example, if X and Y appear free in the type of f, they must appear earlier in the single context (Γ) as below:

$$\Gamma = \ldots, X^*, \ldots, Y^*, \ldots, (f : \forall Z^*.X \to Y \to Z), \ldots$$

In such a formalization, the side condition ($X \notin \Gamma$) in the ($\forall I$) rule of Figure 1 is not necessary, since such a condition is already a part of the well-formedness condition for the context (i.e., Γ, X^κ is well-formed when $X \notin \mathrm{FV}(\Gamma)$). Thus, for F_ω, it is only a matter of taste whether to formalize the system using a single context or two contexts, since they are equivalent formalizations with comparable complexity.

[1] The Church-Rosser property, in its strictest sense (i.e., α-equivalence over terms), generally does not hold in Church-style calculi , but may hold under certain approximations, such as modulo ignoring the annotations in abstractions.

However, in F_i, we separate the context into two parts to distinguish term variables used in types (which we call index variables, or indices, and are bound as Δ, i^A) from the ordinary use of term variables (which are bound as $\Gamma, x : A$). The expectation is that indices should have no effect on reduction at the term level. Although it is imaginable to formalize F_i with a single typing context and distinguish index variables from ordinary term variables using more general concepts (e.g., capability, modality), we think that splitting the typing context into two parts is the simplest solution.

3.2 System F_i compared to System F_ω

We assume readers to be familiar with System F_ω and focus on describing the new constructs of F_i. These appear in grey boxes.

Kinds. The key extension to F_ω is the addition of term-indexed arrow kinds of the form $A \to \kappa$. This allows type constructors to have terms as indices. The rest of the development of F_i flows naturally from this single extension.

Sorting. The formation of indexed arrow kinds is governed by the sorting rule (Ri). The rule (Ri) specifies that an indexed arrow kind $A \to \kappa$ is well-sorted when A has kind $*$ under the empty type level context $(\cdot)$ and κ is well-sorted.

Requiring the use of the empty context avoids dependent kinds (i.e., kinds depending on type level or value level bindings). The type A appearing in the index arrow kind $A \to \kappa$ must be well-kinded under the empty type level context $(\cdot)$. That is, A should to be a closed type of kind $*$, which does not contain any free type variables or index variables. For example, $(\mathit{List}\, X \to *)$ is not a well-sorted kind, while $((\forall X^*.\,\mathit{List}\, X) \to *)$ is a well-sorted kind.

Typing contexts. Typing contexts are split into two parts. Type level contexts (Δ) for type level (static) bindings, and term level contexts (Γ) for term level (dynamic) bindings. A new form of index variable binding (i^A) can appear in type level contexts in addition to the traditional type variable bindings (X^κ). There is only one form of term level binding ($x : A$) that appears in term level contexts.

Well formed typing contexts. A type level context Δ is well-formed if (1) it is either empty, or (2) extended by a type variable binding X^κ whose kind κ is well-sorted under Δ, or (3) extended by an index binding i^A whose type A is well-kinded under the empty type level context at kind $*$. This restriction is similar to the one that occurs in the sorting rule (Ri) for term-indexed arrow kinds (see the paragraph ***Sorting***). The consequence of this is that, in typing contexts and in sorts, A must be closed type (not a type constructor!) without free variables.

A term level context Γ is well-formed under a type level context Δ when it is either empty or extended by a term variable binding $x : A$ whose type A is well-kinded under Δ.

Type constructors and their kinding rules. We extend the type constructor syntax by three constructs, and extend the kinding rules accordingly for these new constructs.

$\lambda i^A.F$ is the type level abstraction over an index (or, index abstraction). Index abstractions introduce indexed arrow kinds by the kinding rule (λi). Note, the use of the new form of context extension, i^A, in the kinding rule (λi).

$F\,\{s\}$ is the type level index application. In contrast to the ordinary type level application $(F\,G)$ where the argument (G) is a type constructor, the argument of an index application $(F\,\{s\})$ is a term (s). We use the curly bracket notation around an index argument in a type to emphasize the transition from ordinary type to term, and to emphasize that s is an index term, which is erasable.

Syntax:

Term Variables	x, i	
Type Constructor Variables	X	
Kinds	κ	$::= \ * \mid \kappa \to \kappa \mid A \to \kappa$
Type Constructors	A, B, F, G	$::= \ X \mid A \to B \mid \lambda X^{\kappa}.F \mid F\,G \mid \forall X^{\kappa}.B \mid \lambda i^{A}.F \mid F\,\{s\} \mid \forall i^{A}.B$
Terms	r, s, t	$::= \ x \mid \lambda x.t \mid r\ s$
Typing Contexts	Δ	$::= \ \cdot \mid \Delta, X^{\kappa} \mid \Delta, i^{A}$
	Γ	$::= \ \cdot \mid \Gamma, x : A$

Well-formed typing contexts:

$$\boxed{\vdash \Delta} \qquad \frac{}{\vdash \cdot} \qquad \frac{\vdash \Delta \quad \vdash \kappa : \Box}{\vdash \Delta, X^{\kappa}}\big(X \notin \mathsf{dom}(\Delta)\big) \qquad \frac{\vdash \Delta \quad \cdot \vdash A : *}{\vdash \Delta, i^{A}}\big(i \notin \mathsf{dom}(\Delta)\big)$$

$$\boxed{\Delta \vdash \Gamma} \qquad \frac{\vdash \Delta}{\Delta \vdash \cdot} \qquad \frac{\Delta \vdash \Gamma \quad \Delta \vdash A : *}{\Delta \vdash \Gamma, x : A}\big(x \notin \mathsf{dom}(\Gamma)\big)$$

Sorting: $\boxed{\vdash \kappa : \Box}$

$$(A)\frac{}{\vdash * : \Box} \qquad (R)\frac{\vdash \kappa : \Box \quad \vdash \kappa' : \Box}{\vdash \kappa \to \kappa' : \Box} \qquad (Ri)\frac{\cdot \vdash A : * \quad \vdash \kappa : \Box}{\vdash A \to \kappa : \Box}$$

Kinding: $\boxed{\Delta \vdash F : \kappa}$

$$(Var)\frac{X^{\kappa} \in \Delta \quad \vdash \Delta}{\Delta \vdash X : \kappa} \qquad (\to)\frac{\Delta \vdash A : * \quad \Delta \vdash B : *}{\Delta \vdash A \to B : *}$$

$$(\lambda)\frac{\vdash \kappa : \Box \quad \Delta, X^{\kappa} \vdash F : \kappa'}{\Delta \vdash \lambda X^{\kappa}.F : \kappa \to \kappa'} \qquad (@)\frac{\Delta \vdash F : \kappa \to \kappa' \quad \Delta \vdash G : \kappa}{\Delta \vdash F\,G : \kappa'} \qquad (\forall)\frac{\vdash \kappa : \Box \quad \Delta, X^{\kappa} \vdash B : *}{\Delta \vdash \forall X^{\kappa}.B : *}$$

$$(\lambda i)\frac{\cdot \vdash A : * \quad \Delta, i^{A} \vdash F : \kappa}{\Delta \vdash \lambda i^{A}.F : A \to \kappa} \qquad (@i)\frac{\Delta \vdash F : A \to \kappa \quad \Delta; \cdot \vdash s : A}{\Delta \vdash F\,\{s\} : \kappa} \qquad (\forall i)\frac{\cdot \vdash A : * \quad \Delta, i^{A} \vdash B : *}{\Delta \vdash \forall i^{A}.B : *}$$

$$(Conv)\frac{\Delta \vdash A : \kappa \quad \Delta \vdash \kappa = \kappa' : \Box}{\Delta \vdash A : \kappa'}$$

Typing: $\boxed{\Delta; \Gamma \vdash t : A}$

$$(:)\frac{(x : A) \in \Gamma \quad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash x : A} \qquad (:i)\frac{i^{A} \in \Delta \quad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash i : A}$$

$$(\to I)\frac{\Delta \vdash A : * \quad \Delta; \Gamma, x : A \vdash t : B}{\Delta; \Gamma \vdash \lambda x.t : A \to B} \qquad (\to E)\frac{\Delta; \Gamma \vdash r : A \to B \quad \Delta; \Gamma \vdash s : A}{\Delta; \Gamma \vdash r\ s : B}$$

$$(\forall I)\frac{\vdash \kappa : \Box \quad \Delta, X^{\kappa}; \Gamma \vdash t : B}{\Delta; \Gamma \vdash t : \forall X^{\kappa}.B}(X \notin \mathrm{FV}(\Gamma)) \qquad (\forall E)\frac{\Delta; \Gamma \vdash t : \forall X^{\kappa}.B \quad \Delta \vdash G : \kappa}{\Delta; \Gamma \vdash t : B[G/X]}$$

$$(\forall Ii)\frac{\cdot \vdash A : * \quad \Delta, i^{A}; \Gamma \vdash t : B}{\Delta; \Gamma \vdash t : \forall i^{A}.B}\begin{pmatrix} i \notin \mathrm{FV}(t), \\ i \notin \mathrm{FV}(\Gamma) \end{pmatrix} \qquad (\forall Ei)\frac{\Delta; \Gamma \vdash t : \forall i^{A}.B \quad \Delta; \cdot \vdash s : A}{\Delta; \Gamma \vdash t : B[s/i]}$$

$$(=)\frac{\Delta; \Gamma \vdash t : A \quad \Delta \vdash A = B : *}{\Delta; \Gamma \vdash t : B}$$

Reduction: $\boxed{t \leadsto t'}$

$$\frac{}{(\lambda x.t)\ s \leadsto t[s/x]} \qquad \frac{t \leadsto t'}{\lambda x.t \leadsto \lambda x.t'} \qquad \frac{r \leadsto r'}{r\ s \leadsto r'\ s} \qquad \frac{s \leadsto s'}{r\ s \leadsto r\ s'}$$

Figure 1. Syntax, Typing rules, and Reduction rules of F_i

Kind equality: $\boxed{\vdash \kappa = \kappa' : \Box}$

$$\frac{}{\vdash * = * : \Box} \qquad \frac{\vdash \kappa_1 = \kappa_1' : \Box \quad \vdash \kappa_2 = \kappa_2' : \Box}{\vdash \kappa_1 \to \kappa_2 = \kappa_1' \to \kappa_2' : \Box} \qquad \frac{\cdot \vdash A = A' : * \quad \vdash \kappa = \kappa' : \Box}{\vdash A \to \kappa = A' \to \kappa' : \Box}$$

$$\frac{\vdash \kappa = \kappa' : \Box}{\vdash \kappa' = \kappa : \Box} \qquad \frac{\vdash \kappa = \kappa' : \Box \quad \vdash \kappa' = \kappa'' : \Box}{\vdash \kappa = \kappa'' : \Box}$$

Type constructor equality: $\boxed{\Delta \vdash F = F' : \kappa}$

$$\frac{\Delta, X^\kappa \vdash F : \kappa' \quad \Delta \vdash G : \kappa}{\Delta \vdash (\lambda X^\kappa.F)\, G = F[G/X] : \kappa'} \qquad \frac{\Delta, i^A \vdash F : \kappa \quad \Delta; \cdot \vdash s : A}{\Delta \vdash (\lambda i^A.F)\, \{s\} = F[s/i] : \kappa}$$

$$\frac{\Delta \vdash X : \kappa}{\Delta \vdash X = X : \kappa} \qquad \frac{\Delta \vdash A = A' : * \quad \Delta \vdash B = B' : *}{\Delta \vdash A \to B = A' \to B' : *}$$

$$\frac{\vdash \kappa : \Box \quad \Delta, X^\kappa \vdash F = F' : \kappa'}{\Delta \vdash \lambda X^\kappa.F = \lambda X^\kappa.F' : \kappa \to \kappa'} \qquad \frac{\Delta \vdash F = F' : \kappa \to \kappa' \quad \Delta \vdash G = G' : \kappa}{\Delta \vdash F\, G = F'\, G' : \kappa'} \qquad \frac{\vdash \kappa : \Box \quad \Delta, X^\kappa \vdash B = B' : *}{\Delta \vdash \forall X^\kappa.B = \forall X^\kappa.B' : *}$$

$$\frac{\cdot \vdash A : * \quad \Delta, i^A \vdash F = F' : \kappa}{\Delta \vdash \lambda i^A.F = \lambda i^A.F' : A \to \kappa} \qquad \frac{\Delta \vdash F = F' : A \to \kappa \quad \Delta; \cdot \vdash s = s' : A}{\Delta \vdash F\, \{s\} = F'\, \{s'\} : \kappa} \qquad \frac{\cdot \vdash A : * \quad \Delta, i^A \vdash B = B' : *}{\Delta \vdash \forall i^A.B = \forall i^A.B' : *}$$

$$\frac{\Delta \vdash F = F' : \kappa}{\Delta \vdash F' = F : \kappa} \qquad \frac{\Delta \vdash F = F' : \kappa \quad \Delta \vdash F' = F'' : \kappa}{\Delta \vdash F = F'' : \kappa}$$

Term equality: $\boxed{\Delta; \Gamma \vdash t = t' : A}$

$$\frac{\Delta; \Gamma, x : A \vdash t : B \quad \Delta; \Gamma \vdash s : A}{\Delta; \Gamma \vdash (\lambda x.t)\, s = t[s/x] : B} \qquad \frac{\Delta; \Gamma \vdash x : A}{\Delta; \Gamma \vdash x = x : A}$$

$$\frac{\Delta \vdash A : * \quad \Delta; \Gamma, x : A \vdash t = t' : B}{\Delta; \Gamma \vdash \lambda x.t = \lambda x.t' : B} \qquad \frac{\Delta; \Gamma \vdash r = r' : A \to B \quad \Delta; \Gamma \vdash s = s' : A}{\Delta; \Gamma \vdash r\ s = r'\ s' : B}$$

$$\frac{\vdash \kappa : \Box \quad \Delta, X^\kappa; \Gamma \vdash t = t' : B}{\Delta; \Gamma \vdash t = t' : \forall X^\kappa.B}\,(X \notin \mathrm{FV}(\Gamma)) \qquad \frac{\Delta; \Gamma \vdash t = t' : \forall X^\kappa.B \quad \Delta \vdash G : \kappa}{\Delta; \Gamma \vdash t = t' : B[G/X]}$$

$$\frac{\cdot \vdash A : * \quad \Delta, i^A; \Gamma \vdash t = t' : B}{\Delta; \Gamma \vdash t = t' : \forall i^A.B} \begin{pmatrix} i \notin \mathrm{FV}(t), \\ i \notin \mathrm{FV}(t'), \\ i \notin \mathrm{FV}(\Gamma) \end{pmatrix} \qquad \frac{\Delta; \Gamma \vdash t = t' : \forall i^A.B \quad \Delta; \cdot \vdash s : A}{\Delta; \Gamma \vdash t = t' : B[s/i]}$$

$$\frac{\Delta; \Gamma \vdash t = t' : A}{\Delta; \Gamma \vdash t' = t : A} \qquad \frac{\Delta; \Gamma \vdash t = t' : A \quad \Delta; \Gamma \vdash t' = t'' : A}{\Delta; \Gamma \vdash t = t'' : A}$$

Figure 2. Equality rules of F_i

Index applications eliminate indexed arrow kinds by the kinding rule $(@i)$. Note, we type check the index term (s) under the current type level context paired with the empty term level context $(\Delta; \cdot)$ since we do not want the index term (s) to depend on any term level bindings. Allowing such a dependency would admit true dependent types.

$\forall i^A.B$ is an index polymorphic type. The formation of indexed polymorphic types is governed by the kinding rule $\forall i$, which is very similar to the formation rule $(\forall)$ for ordinary polymorphic types.

In addition to the rules (λi), $(@i)$, and $(\forall i)$, we need a conversion rule $(Conv)$ at kind level. This is because the new extension to the kind syntax $A \to \kappa$ involves types. Since kind syntax involves types, we need more than simple structural equality over kinds. The new equality over kinds is the usual structural equality extended by type constructor equality when comparing indexed arrow kinds (see Figure 2).

Terms and their typing rules The term syntax is exactly the same as other Curry-style calclui. We write x for ordinary term variables introduced by term level abstractions $(\lambda x.t)$. We write i for index variables introduced by index abstractions $(\lambda i^A.F)$ and by index polymorphic types $(\forall i^A.B)$. As discussed earlier, the distinction between x and i is for the convenience of readability.

Since F_i has index polymorphic types $(\forall i^A.B)$, we need typing rules for index polymorphism: $(\forall Ii)$ for index generalization and $(\forall Ei)$ for index instantiation.

The index generalization rule $(\forall Ii)$ is similar to the type generalization rule $(\forall I)$, but generalizes over index variables (i) rather than type consturctor variables (X). The rule $(\forall Ii)$ has two side conditions while the rule $(\forall I)$ has only one side conditions. The additional side condition $i \notin \mathrm{FV}(t)$ in the $(\forall Ii)$ rule prevents terms from accessing the type level index variables introduced by index polymorphism. Without this side condition, $\forall$-binder would no longer behave polymorphicaly, but instead would behave as a dependent function, which are usually denoted by the Π-binder in dependent type theories. The rule $(\forall I)$ for ordinary type generalization does not need such additional side condition because type variables cannot appear in the syntax of terms. The side conditions on generalization rules for polymorphism is fairly standard in dependently typed languages supporting distinctions between polymorphism (or, erasable arguments) and dependent functions (e.g., IPTS[14], ICC[13]).

$$\dfrac{\cdot \vdash A : * \qquad \dfrac{\Delta, i^A \vdash F : A \to \kappa \qquad \dfrac{i^A \in \Delta, i^A \qquad \Delta \vdash \cdot}{\Delta, i^A; \cdot \vdash i : A}(:i)}{\Delta, i^A \vdash F\{i\} : \kappa}(@i)}{\Delta \vdash \lambda i^A.F\{i\} : A \to \kappa}(\lambda i)$$

Figure 3. Kinding derivation for an index abstraction

$$
\begin{array}{lll}
\texttt{Bool} & = & \forall X.X \to X \to X \\
\texttt{true} & : & \texttt{Bool} \;=\; \lambda x_1.\lambda x_2.x_1 \\
\texttt{false} & : & \texttt{Bool} \;=\; \lambda x_1.\lambda x_2.x_2 \\
\texttt{elim}_{\texttt{Bool}} & : & \texttt{Bool} \to \forall X.X \to X \to X \\
 & = & \lambda x.\lambda x_1.\lambda x_2.x\; x_1\; x_2 \qquad (\textbf{if}\; x\; \textbf{then}\; x_1\; \textbf{else}\; x_2) \\
\hline
A_1 \times A_2 & = & \forall X.(A_1 \to A_2 \to X) \to X \\
\texttt{pair} & : & \forall A_1^*.\forall A_2^*.A_1 \times A_2 \;=\; \lambda x_1.\lambda x_2.\lambda x'.x'\; x_1\; x_2 \\
\texttt{elim}_{(\times)} & : & \forall A_1^*.\forall A_2^*.A_1 \times A_2 \to \forall X.(A_1 \to A_2 \to X) \to X \\
 & = & \lambda x.\lambda x'.x\; x' \\
 & & \text{(by passing appropriate values to } x'\text{, we get} \\
 & & \;\textit{fst} = \lambda x.x(\lambda x_1.\lambda x_2.x_1),\; \textit{snd} = \lambda x.x(\lambda x_1.\lambda x_2.x_2)\;) \\
\hline
A_1 + A_2 & = & \forall X^*.(A_1 \to X) \to (A_2 \to X) \to X \\
\texttt{inl} & : & \forall A_1^*.\forall A_2^*.A_1 \to A_1 + A_2 \;=\; \lambda x.\lambda x_1.\lambda x_2.x_1\; x \\
\texttt{inr} & : & \forall A_1^*.\forall A_2^*.A_2 \to A_1 + A_2 \;=\; \lambda x.\lambda x_1.\lambda x_2.x_2\; x \\
\texttt{elim}_{(+)} & : & \forall A_1^*.\forall A_2^*.(A_1 + A_2) \to \\
 & & \quad \forall X^*.(A_1 \to X) \to (A_2 \to X) \to X \\
 & = & \lambda x.\lambda x_1.\lambda x_2.x\; x_1\; x_2 \\
 & & \quad (\textbf{case}\; x\; \textbf{of}\; \{\texttt{inl}\; x' \to x_1\; x'; \texttt{inr}\; x' \to x_2\; x'\})
\end{array}
$$

Figure 4. Embedding non-recursive datatypes

The index instantiation rule ($\forall Ei$) is similar to the type instantiation rule ($\forall Ei$), except that we type check the index term s to be instantiated for i in the current type level context paired with the empty term level context ($\Delta; \cdot$) rather than the current term level context. Since index terms are at type level, they should not depend on term level bindings.

In addition to the rules ($\forall Ii$) and ($\forall Ei$) for index polymorphism, we need an additional variable rule $(:i)$ to be able to access the index variables already in scope. Terms (s) used at type level in index applications ($F\{s\}$) should be able to access index variables already in scope. For example, $\lambda i^A.F\{i\}$ should be well-kinded under a context where F is well-kinded, justified by the derivation in Figure 3.

4. Embedding datatypes and their eliminators

We demonstrate some examples of embedding datatypes into System F_i. We first illustrate embeddings for both non-recursive datatypes and recursive datatypes, where we use Church encodings [5] for data constructors (§4.1). Then, we illustrate a more involved embedding for the recursive datatypes based on two-level types (§4.2).

$$
\begin{array}{lll}
\texttt{List} & = & \lambda A^*.\forall X^*.(A \to X \to X) \to X \to X \\
\texttt{cons} & : & \forall A^*.A \to \texttt{List}\,A \to \texttt{List}\,A \\
 & & \qquad = \lambda x_a.\lambda x.\lambda x_c.\lambda x_n.x_c\, x_a\, (x\; x_c\, x_n) \\
\texttt{nil} & : & \forall A^*.\texttt{List}\,A \;=\; \lambda x_c.\lambda x_n.\lambda x_n \\
\texttt{elim}_{\texttt{List}} & : & \forall A^*.\texttt{List}\,A \to \forall X^*.(A \to X \to X) \to X \to X \\
 & = & \lambda x.\lambda x_c.\lambda x_n.x\; x_c\, x_n \qquad (\textit{foldr}\; x_z\; x_c\; x \;\text{ in Haskell}) \\
\hline
\texttt{Powl} & = & \lambda A^*. \\
 & & \forall X^{*\to*}.(A \to X(A \times A) \to XA) \to XA \to XA \\
\texttt{pcons} & : & \forall A^*.A \to \texttt{Powl}(A \times A) \to \texttt{Powl}\,A \\
 & & \qquad = \lambda x_a.\lambda x.\lambda x_c.\lambda x_n.x_c\, x_a\, (x\; x_c\, x_n) \\
\texttt{pnil} & : & \forall A^*.\texttt{Powl}\,A \;=\; \lambda x_c.\lambda x_n.\lambda x_n \\
\texttt{elim}_{\texttt{Powl}} & : & \forall A^*.\texttt{Powl}\,A \to \\
 & & \forall X^{*\to*}.(A \to X(A \times A) \to XA) \to XA \to XA \\
 & = & \lambda x.\lambda x_c.\lambda x_n.x\; x_c\, x_n \\
\hline
\texttt{Vec} & = & \lambda A^*.\lambda i^{\texttt{Nat}}. \\
 & & \forall X^{\texttt{Nat}\to*}.(\forall i^{\texttt{Nat}}.A \to X\{i\} \to X\{\texttt{succ}\,i\}) \to \\
 & & \qquad X\{\texttt{zero}\} \to X\{i\} \\
\texttt{vcons} & : & \forall A^*.\forall i^{\texttt{Nat}}.A \to \texttt{Vec}\,A\,\{i\} \to \texttt{Vec}\,A\,\{\texttt{succ}\,i\} \\
 & & \qquad = \lambda x_a.\lambda x.\lambda x_c.\lambda x_n.x_c\, x_a\, (x\; x_c\, x_n) \\
\texttt{vnil} & : & \forall A^*.\texttt{Vec}\,A\,\{\texttt{zero}\} \;=\; \lambda x_c.\lambda x_n.\lambda x_n \\
\texttt{elim}_{\texttt{Vec}} & : & \forall A^*.\forall i^{\texttt{Nat}}.\texttt{Vec}\,A\,\{i\} \to \\
 & & \forall X^{\texttt{Nat}\to*}.(\forall i^{\texttt{Nat}}.A \to X\{i\} \to X\{\texttt{succ}\,i\}) \to \\
 & & \qquad X\{\texttt{zero}\} \to X\{i\} \\
 & = & \lambda x.\lambda x_c.\lambda x_n.x\; x_c\, x_n
\end{array}
$$

Figure 5. Embedding recursive datatypes

4.1 Embedding datatypes using Church-encoded terms

Church [5] demonstrated an embedding of natural numbers into the untyped λ-calculus, which he invented, in order to argue that the λ-calculus expressive enough for the foundation of logic and arithmetic. Church encoded the data constructors of natural numbers, successor and zero, as higher-order functions, $\texttt{succ} = \lambda x.\lambda x_s.\lambda x_z.x_s(x\, x_s x_z)$ and $\texttt{zero} = \lambda x_s.\lambda x_z.x_z$. The heart of the Church encoding is that a value is encoded as its elimination. The bound variables x_s and x_z stands for the operations needed for eliminating the successor case and the zero case. The Church encodings of successor and zero states that: to eliminate $\texttt{succ}\, x$, apply x_s to the elimination of the predecessor $(x\, x_s x_z)$; and, to eliminate $\texttt{zero}$, just return x_z. Since values themselves are eliminators, eliminator can be defined as applying the value itself to the needed operations for each data constructor case. For instance, we can define an eliminator for natural numbers as $\texttt{elim}_{\texttt{Nat}} = \lambda x.\lambda x_s.\lambda x_z.x\, x_s x_z$, which is just an η-expansion of the identity function $\lambda x.x$. Church encoded natural numbers are typable in polymorphic λ-calculi, such as System F_ω, as follows:

$$
\begin{array}{lll}
\texttt{Nat} & = & \forall X^*.(X \to X) \to X \to X \\
\texttt{succ} & : & \texttt{Nat} \to \texttt{Nat} \;=\; \lambda x.\lambda x_s.\lambda x_z.x_s(x\, x_s x_z) \\
\texttt{zero} & : & \texttt{Nat} \qquad\quad\; = \lambda x_s.\lambda x_z.x_z \\
\texttt{elim}_{\texttt{Nat}} & : & \texttt{Nat} \to \forall X^*.(X \to X) \to X \to X \\
 & = & \lambda x.\lambda x_s.\lambda x_z.x\, x_s x_z
\end{array}
$$

Similarly, other datatypes are also embeddable into polymorphic λ-calculi in this fashion. Embeddings of some well-known non-recursive datatypes are illustrated in Figure 4, and embeddings of the list-like recursive datatypes, which we discussed earlier as motivating examples (§2), are illustrated in Figure 5. Note that the term encodings for the constructors and eliminators of the list-like datatypes in Figure 5 are exactly the same. For instance, the term encodings for `nil`, `pnil`, and `vnil` coincide as $\lambda x_s.\lambda x_z.x_z$.

4.2 Embedding recursive datatypes as two-level types

We can divide a recursive datatype definition into two levels, by factoring out the recursive type operator, which weaves in the recursion to the datatype definition, and a non-recursive base structure, which describes the shape (i.e., number of data constructors and their types) of the datatype. We can program with two-level types in functional languages that support higher-order polymorphism[2], such as Haskell, as illustrated in Figure 6. The function $\mathtt{mit}_\kappa$ describes the Mendler-style iteration[3] for the recursive types defined by μ_κ. The use of two-level types has been recognized as a useful functional programming pearl [18] since two-level types separate the two concerns of (1) recursing deeper into recursive subcomponents and (2) handling different cases by the shape of the base structure. Although it is possible to write programs using two level datatypes, one could not expect logical consistency in such general purpose functional languages.

Interestingly, there exists an embedding of the recursive type operator μ_κ, its data constructor $\mathtt{In}_\kappa$, and the Mendler-style iterator $\mathtt{mit}\kappa$ at each kind κ into a higher-order polymorphic λ-calculus, as illustrated in Figure 7. In addition to illustrating the general form of embedding μ_κ, we also fully expand the embeddings for some instances (μ_*, $\mu_{*\to*}$, $\mu_{\mathtt{Nat}\to*}$), which is used in Figure 6. This embedding allows arbitrary type- and term-indexed recursive datatypes be embeddable into System F_i, so that we can reason about these datatypes in a logically consistent calculus. However, it is important to note that there does not exist an embedding of arbitrary destruction (or, pattern matching away) of the $\mathtt{In}_\kappa$ constructor. It is known that having arbitrary recursive datatypes together with the ability to arbitrarily destruct (or, unroll) the values of recursive types is powerful enough to define non-terminating computation in a type safe way, which leads to logical inconsistency.

4.3 Leibniz index equality

The quantification over type-indexed kinding available in System F_i allows the definition of *Leibniz-equality type* constructors $\mathtt{Eq}_A : A \to A \to *$ on closed types A. These are as follows

$$\mathtt{Eq}_A \triangleq \lambda i^A.\,\lambda j^A.\,\mathtt{LEq}_A\{i\}\{j\} \times \mathtt{LEq}_A\{j\}\{i\}$$

where

$$\mathtt{LEq}_A \triangleq \lambda i^A.\,\lambda j^A.\,\forall X^{A\to*}.\,X\{i\} \to X\{j\}\ \ .$$

Note that $\Delta \vdash F\{s\}\{t\} = F\{s'\}\{t'\} : *$ for $F = \mathtt{Eq}_A, \mathtt{LEq}_A$ and all $\Delta;\cdot \vdash s = s' : A, \Delta;\cdot \vdash t = t' : A$.

As a basic property, one has that the types

$$\forall i^A.\,\mathtt{LEq}_A\{i\}\{i\}$$
$$\forall i^A.\,\forall j^A.\,\forall k^A.\,\mathtt{LEq}_A\{i\}\{j\} \to \mathtt{LEq}_A\{j\}\{k\} \to \mathtt{LEq}_A\{i\}\{k\}$$
$$\forall i^A.\,\forall j^A.\,\mathtt{LEq}_A\{i\}\{j\} \to \forall f^{A\to B}.\,\mathtt{LEq}_B\{f\,i\}\{f\,j\}$$

are inhabited, and hence that Leibniz equality is a congruence.

In applications, the types $\mathtt{LEq}_A$ are useful in constraining the term-indexing of datatypes as parameterised by coercions. A general such construction is given by the type constructors $\mathtt{Ext}_{A,B}$: $(A \to B) \to (A \to *) \to B \to *$, which are in spirit right Kan extensions (e.g. [1]), defined as

$$\mathtt{Ext}_{A,B} \triangleq \lambda f^{A\to B}.\,\lambda X^{A\to*}.\,\lambda j^B.\forall i^A.\,\mathtt{LEq}_B\{j\}\{f\,i\} \to X\{i\}$$

for closed types A and B. Here, for closed $t : A \to B$, $F : A \to *$, and $s : B$, a closed term $u : (\mathtt{Ext}_{A,B}\ \{t\}\ F)\{s\}$ is a polymorphic function that, for every closed $r\ :\ A$, given a generic coercion $\forall X^{B\to*}.\,X\{s\} \to X\{t\,r\}$ provides output of type $F\{r\}$.

As a concrete example to which the same principle applies, consider the type constructor

$$\begin{aligned}\mathtt{Fin} \triangleq \lambda i^{\mathtt{Nat}}.\,\forall X^{\mathtt{Nat}\to*}.&\\ &(\forall j^{\mathtt{Nat}}.\,\mathtt{LEq}_{\mathtt{Nat}}\{i\}\{\mathtt{S}\,j\} \to X\{i\})\\ &\quad\to (\forall j^{\mathtt{Nat}}.\,X\{j\} \to X\{\mathtt{S}\,j\})\\ &\qquad\to X\{i\}\end{aligned}$$

For it, one derives constructors

$$\mathtt{FinZ} : \forall i^{\mathtt{Nat}}\,\forall j^{\mathtt{Nat}}.\,\mathtt{LEq}_{\mathtt{Nat}}\{i\}\{\mathtt{S}\,j\} \to \mathtt{Fin}\{i\}$$
$$\mathtt{FinS} : \forall i^{\mathtt{Nat}}.\,\mathtt{Fin}\{i\} \to \mathtt{Fin}\{\mathtt{S}\,i\}$$

```
newtype μ* (f :: * -> *)
  = In* (f (μ* f))

data ListF (a::*) (r::*)
  = Cons a r       | Nil

type List a = μ* (ListF a)
cons x xs = In* (Cons x xs)
nil       = In* Nil

mit* :: (∀ r.(r->x) -> f r -> x) -> Mu0 f -> x
mit* phi (In* z) = phi (mit* phi) z

newtype μ(*→*) (f :: (*->*) -> (*->*)) (a::*)
  = In(*→*) (f (Mu(*→*) f)) a

data PowlF (r::*->*) (a::*)
  = PCons a (r(a,a)) | PNil

type Powl a = μ(*→*) PowlF a
pcons x xs = In(*→*) (PCons x xs)
pnil       = In(*→*) PNil

mit(*→*) :: (∀ r a.(∀a.r a->x a) -> f r a -> x a)
         -> μ(*→*) f a -> x a
mit(*→*) phi (In(*→*) z) = phi (mit(*→*) phi) z

-- above is Haskell (with some GHC extensions)
-- below is Haskell-ish psudocode

newtype μ(Nat→*) (f::(Nat->*)->(Nat->*)) {n::Nat}
  = In(Nat→*) (f (μ(Nat→*) f)) {n}

data VecF (a::*) (r::Nat->*) {n::Nat} where
  VCons :: a -> r n -> VecF a r {S n}
  VNil  :: VecF a r {Z}

type Vec a {n::Nat} = μ(Nat→*) (VecF a) {n}
vcons x xs = In(Nat→*) (VCons x xs)
vnil       = In(Nat→*) VNil

mit(Nat→*)::(∀ r n.(∀n.r{n}->x{n})->f r {n}->x{n})
         -> μ(Nat→*) f {n} -> x{n}
mit(Nat→*) phi (In(Nat→*) z) = phi (mit(Nat→*) phi) z
```

Figure 6. 2-level types and their Mendler-style iterators in Haskell

[2] a.k.a. higher-kinded polymorphism, or type-constructor polymorphism

[3] An iteration is a principled recursion scheme guaranteed to terminate for any well-founded input. Also known as fold, or catamorphism.

notation: $\boldsymbol{\lambda}\mathbb{I}^\kappa.F = \lambda I_1^{\kappa_1}.\cdots.\lambda I_n^{\kappa_n}.F \qquad \forall\mathbb{I}^\kappa.B = \forall I_1^{\kappa_1}.\cdots.\forall I_n^{\kappa_n}.B \qquad F\mathbb{I} = F I_1 \cdots I_n \qquad F \xrightarrow{\kappa} G = \forall\mathbb{I}^\kappa.F\mathbb{I} \to G\mathbb{I}$

where $\kappa = \kappa_1 \to \cdots \to \kappa_n \to *$ and I_i is an index variable (i_i) when κ_i is a type, $\mathbb{I} = I_1, \ \ldots \ \ldots \ , \ I_n$ a type constructor variable (X_i) otherwise.

$$
\begin{array}{lll}
\mu_\kappa & : (\kappa \to \kappa) \to \kappa & = \lambda F^{\kappa\to\kappa}.\boldsymbol{\lambda}\mathbb{I}^\kappa.\forall X^\kappa.(\forall X_r^\kappa.(X_r \xrightarrow{\kappa} X) \to (F X_r \xrightarrow{\kappa} X)) \to X\mathbb{I} \\
\mu_* & : (* \to *) \to * & = \lambda F^{*\to*}.\qquad \forall X^*.(\forall X_r^*.(X_r \to X) \to (F\, X_r \to X)) \to X \\
\mu_{*\to*} & : ((* \to *) \to (* \to *)) \to (* \to *) & \\
& = \lambda F^{(*\to*)\to(*\to*)}.\lambda X_1^*.\forall X^{*\to*}.(\forall X_r^{*\to*}.(\forall X_1^*.X_r X_1 \to X X_1) \to (\forall X_1^*.F\, X_r X_1 \to X X_1)) \to X X_1 & \\
\mu_{\mathtt{Nat}\to*} & : ((\mathtt{Nat} \to *) \to (\mathtt{Nat} \to *)) \to (\mathtt{Nat} \to *) & \\
& = \lambda F^{(\mathtt{Nat}\to*)\to(\mathtt{Nat}\to*)}.\lambda i_1^{\mathtt{Nat}}.\forall X^{\mathtt{Nat}\to*}.(\forall X_r^{\mathtt{Nat}\to*}.(\forall i_1^{\mathtt{Nat}}.X_r i_1 \to X i_1) \to (\forall i_1^{\mathtt{Nat}}.F\, X_r i_1 \to X i_1)) \to X i_1 &
\end{array}
$$

$$
\begin{array}{lll}
\mathtt{In}_\kappa & : \forall F^{\kappa\to\kappa}.F(\mu_\kappa F) \xrightarrow{\kappa} \mu_\kappa F & = \lambda x_r.\lambda x_\varphi.x_\varphi\,(\mathtt{mit}_\kappa\, x_\varphi)\, x_r \\
\mathtt{mit}_\kappa & : \forall F^{\kappa\to\kappa}.\forall X^\kappa.(\forall X_r^\kappa.(X_r \xrightarrow{\kappa} X) \to (F X_r \xrightarrow{\kappa} X)) \to (\mu_\kappa F \xrightarrow{\kappa} X) & = \lambda x_\varphi.\lambda x_r.x_r\, x_\varphi
\end{array}
$$

Figure 7. Embedding of the recursive operators (μ_κ), their data constructors ($\mathtt{In}_\kappa$), and the Mendler-style iterators ($\mathtt{mit}_\kappa$).

so that

$$\mathtt{FinZ}\,(\lambda x.\, x) : \mathtt{Fin}\{\mathtt{S}\, t\} \ , \quad \mathtt{FinS} : \mathtt{Fin}\{t\} \to \mathtt{Fin}\{\mathtt{S}\, t\}$$

for all closed $t : \mathtt{Nat}$.

With $\mathtt{Fin}$, one defines the type of λ-terms in context as:

$$
\begin{array}{l}
\lambda i^{\mathtt{Nat}}.\, \forall X^{\mathtt{Nat}\to*}. \\
\quad (\forall j^{\mathtt{Nat}}.\, \mathtt{Fin}\{j\} \to X\{j\}) \\
\qquad \to (\forall j^{\mathtt{Nat}}.\, X\{j\} \to X\{j\} \to X\{j\}) \\
\qquad\quad \to (\forall j^{\mathtt{Nat}}.\, X\{\mathtt{S}\, j\} \to X\{j\}) \\
\qquad\qquad \to X\{i\}
\end{array}
$$

5. Metatheory

The expectation is that System F_i has all the nice properties of System F_ω, yet is more expressive because of the addition of term-indexed types.

We show some basic well-formedness properties for the judgments of F_i in §5.1. We prove erasure properties of F_i, which captures the idea that indices are erasable since they are irrelevant for reduction in §5.2. We show strong normalization, logical consistence, and subject reduction for F_i by reasoning about well-known calculi related to F_i in §5.3.

5.1 Well-formedness properties and substitution lemmas

We want to show that the sorting, kinding, and typing derivations give well-formed results under well-formed contexts. That is, sorting derivations result in well-formed sorts (Proposition 1), kinding derivations result in well-sorted kinds under well-formed type level contexts (Proposition 2), and typing derivations result in well-kinded types under well-formed type and term level contexts (Proposition 3).

Since the definitions of sorting, kinding, and typing rules are mutually recursive, these three properties are considered as one big property (illustrated below) in order to be more rigorous abouts the induction principle used in the proof.

Proposition (The big well-formedness property of F_i, roughly[4]).

case $\boxed{\vdash \kappa : \square}$ $\quad \dfrac{\vdash \kappa : \mathfrak{s}}{\mathfrak{s} = \square}$
(Proposition 1)

case $\boxed{\Delta \vdash F : \kappa}$ $\quad \dfrac{\vdash \Delta \quad \Delta \vdash F : \kappa}{\vdash \kappa : \square}$
(Proposition 2)

case $\boxed{\Delta; \Gamma \vdash t : A}$ $\quad \dfrac{\Delta \vdash \Gamma \quad \Delta; \Gamma \vdash t : A}{\Delta \vdash A : *}$
(Proposition 3)

The big well-formedness property has one of the three forms – $\boxed{\vdash \kappa : \square}$ (sorting), $\boxed{\Delta \vdash F : \kappa}$ (kinding), and $\boxed{\Delta; \Gamma \vdash t : A}$ typing. That is, a derivation for a judgment of either sorting, kinding, or typing results in either a well-formed sort (when it is a sorting judgment), a well-sorted kind (when it is a kinding judgment), or a well-kinded type (when it is a typing judgment), under well-formed contexts for the judgment (no context for sorting judgments, Δ for kinding judgments, and $\Delta; \Gamma$ for typing judgments).

We can prove the big well-formedness property of F_i by induction on the derivation of a judgment, which can be any one of the three forms. Here, we illustrate the proof for the three propositions as if they were separate proofs. Because it provides a more intuitive proof sketch, during the proof description, the proof for each proposition references the other properties (which are yet another application of the induction hypothesis of the big well-formedness property). So, when we say "by induction" during the proofs, what we really mean is the induction hypothesis of the big well-formedness property.

Proposition 1 (sorting derivations result in well-formed sorts)**.**

$$\frac{\vdash \kappa : \mathfrak{s}}{\mathfrak{s} = \square}$$

Proof. Obvious since $\square$ is the only sort in F_i. $\square$

Proposition 2 (kinding derivations under well-formed contexts result in well-sorted kinds)**.**

$$\frac{\vdash \Delta \quad \Delta \vdash F : \kappa}{\vdash \kappa : \square}$$

Proof. By induction on the derivation.

case (Var) Trivial by the second well-formedness rule of Δ.

[4] Technically, this is not yet completely rigorous since there are three more forms of judgments in the mutually recursive definition. The *kind equality*, *type considered equality*, and *term equality* rules are part of the mutually recursive definition along with the sorting, kinding, and typing rules. So, the complete description of the big well-formedness property will consist of six cases, which correspond to Proposition 1, Proposition 2, Proposition 3, Lemma 1, Lemma 2, and Lemma 3.

case ($Conv$) By induction and Lemma 1.

case (λ)

By induction and Proposition 1 we know that $\vdash \kappa : \Box$.

By the second well-formedness rule of Δ, we know that $\vdash \Delta, X^\kappa$ since we already know that $\vdash \kappa : \Box$ and $\vdash \Delta$ from the property statement.

By induction, we know that $\vdash \kappa' : \Box$ since we already know that $\vdash \Delta, X^\kappa$ and that $\Delta, X^\kappa \vdash F : \kappa'$ from induction hypothesis.

By the sorting rule (R), we know that $\vdash \kappa \to \kappa' : \Box$ since we already know that $\vdash \kappa : \Box$ and $\vdash \kappa' : \Box$.

case ($@$) By induction, easy.

case (λi)

By induction and Proposition 3 we know that $\cdot \vdash A : *$. By the third well-formedness rule of Δ, we know that $\vdash \Delta, i^A$ since we already know that $\cdot \vdash A : *$ and that $\vdash \Delta$ from the property statement.

By induction, we know that $\vdash \kappa : \Box$ since we already know that $\vdash \Delta, i^A$ and that $\Delta, i^A \vdash F : \kappa$ from the induction hypothesis.

By the sorting rule (Ri), we know that $\vdash A \to \kappa : \Box$ since we already know that $\cdot \vdash A : *$ and $\vdash \kappa : \Box$.

case ($@i$) By induction and Proposition 3, easy.

case ($\to$) Trivial since $\vdash * : \Box$.

case ($\forall$) Trivial since $\vdash * : \Box$.

case ($\forall i$) Trivial since $\vdash * : \Box$.

$\Box$

The basic structure of the proof for the following proposition on typing derivations is similar to above. So, we illustrate the proof for most of the cases, which can be done by applying the induction hypothesis, rather bravely. We elaborate more on interesting cases ($\forall E$) and ($\forall Ei$) which involve substitutions in the types resulting from the typing judgments.

Proposition 3 (typing derivations under well-formed contexts result in well-kinded types)**.**

$$\frac{\Delta \vdash \Gamma \quad \Delta; \Gamma \vdash t : A}{\Delta \vdash A : *}$$

Proof. By induction on the derivation.

case ($:$) Trivial by the second well-formedness rule of Γ.

case ($: i$) Trivial by the third the well-formedness rule of Δ.

case ($=$) By induction and Lemma 2.

case ($\to I$) By induction and well-formedness of Γ.

case ($\to E$) By induction.

case ($\forall I$) By induction and well-formedness of Δ.

case ($\forall E$)

By induction we know that $\Delta \vdash \forall X^\kappa . B : *$.

By the kinding rule ($\forall$), which is the only kinding rule able to derive $\Delta \vdash \forall X^\kappa . B : *$, we know that $\Delta, X^\kappa \vdash B : *$.

Then, we use the type substitution lemma (Lemma 4).

case ($\forall Ii$) By induction and well-formedness of Δ.

case ($\forall Ei$)

By induction we know that $\Delta \vdash \forall i^A . B : *$.

By the kinding rule ($\forall i$), which is the only kinding rule able to derive $\Delta \vdash \forall i^A . B : *$, we know that $\Delta, i^A \vdash B : *$.

Then, we use the index substitution lemma (Lemma 5).

$\Box$

Lemma 1 (kind equality is well-sorted)**.** $\dfrac{\vdash \kappa = \kappa' : \Box}{\vdash \kappa : \Box \quad \vdash \kappa' : \Box}$

Proof. By induction on the derivation of kind equality and using the sorting rules. $\Box$

Lemma 2 (type constructor equality is well-kinded)**.**

$$\frac{\Delta \vdash F = F' : \kappa}{\Delta \vdash F : \kappa \quad \Delta \vdash F' : \kappa}$$

Proof. By induction on the derivation of type constructor equality and using the kinding rules. Also use the type substitution lemma (Lemma 4) and the index substitution lemma (Lemma 5). $\Box$

Lemma 3 (term equality is well-typed)**.**

$$\frac{\Delta, \Gamma \vdash t = t' : A}{\Delta, \Gamma \vdash t : A \quad \Delta, \Gamma \vdash t' : A}$$

Proof. By induction on the derivation of term equality and using the typing rules. Also use the term substitution lemma (Lemma 6). $\Box$

The proofs for the three lemmas above are straightforward once we have dealt with the interesting cases for the equality rules involving substitution. We can prove those interesting cases by applying the substitution lemmas. The other cases fall into two categories: firstly, the equality rules following the same structure of the sorting, kinding, and typing rules; and secondly, the reflexive rules and the transitive rules. The proof for the equality rules following the same structure of the sorting, kinding, and typing rules can be proved by induction and applying the corresponding sorting, kinding, and typing rules. The proof for the reflexive rules and the transitive rules can be proved simply by induction.

Lemma 4 (type substitution)**.** $\dfrac{\Delta, X^\kappa \vdash F : \kappa' \quad \Delta \vdash G : \kappa}{\Delta \vdash F[G/X] : \kappa'}$

Lemma 5 (index substitution)**.** $\dfrac{\Delta, i^A \vdash F : \kappa \quad \Delta; \cdot \vdash s : A}{\Delta \vdash F[s/i] : \kappa}$

Lemma 6 (term substitution)**.** $\dfrac{\Delta; \Gamma, x : A \vdash t : B \quad \Delta; \cdot \vdash s : A}{\Delta, \Gamma \vdash t[s/x] : B}$

5.2 Erasure properties

Definition 1 (index erasure)**.**

$$\boxed{\kappa^\circ} \quad *^\circ = * \quad (\kappa_1 \to \kappa_2)^\circ = {\kappa_1}^\circ \to {\kappa_2}^\circ \quad (A \to \kappa)^\circ = \kappa^\circ$$

$$\boxed{F^\circ} \quad X^\circ = X \qquad (A \to B)^\circ = A^\circ \to B^\circ$$
$$(\lambda X^\kappa . F)^\circ = \lambda X^{\kappa^\circ} . F^\circ \qquad (\lambda i^A . F)^\circ = F^\circ$$
$$(F\ G)^\circ = F^\circ\ G^\circ \qquad (F\ \{s\})^\circ = F^\circ$$
$$(\forall X^\kappa . B)^\circ = \forall X^{\kappa^\circ} . B^\circ \qquad (\forall i^A . B)^\circ = B^\circ$$

$$\boxed{\Delta^\circ} \quad \cdot^\circ = \cdot \quad (\Delta, X^\kappa)^\circ = \Delta^\circ, X^{\kappa^\circ} \quad (\Delta, i^A)^\circ = \Delta^\circ$$

$$\boxed{\Gamma^\circ} \quad \cdot^\circ = \cdot \quad (\Gamma, x : A)^\circ = \Gamma^\circ, x : A^\circ$$

Example 1. *The meta-operation of index-erasure discards all indexing information. The effect of this on most datatypes is to project the indexing invariants while retaining the type structure. For instance, the index erasure of* $\texttt{Vec} : * \to \texttt{Nat} \to *$ *as introduced in Section 2 is* $\texttt{List} \triangleq \lambda A^* . \forall X^* . X \to (A \to X \to X) \to X : * \to *$*. However, one can build pathological examples. To this end, let* $\texttt{Void} \triangleq \forall X^* . X$ *and consider* $\forall F^{\texttt{Void} \to *} . \forall i^{\texttt{Void}} . \forall j^{\texttt{Void}} . F\{i\} \to F\{j\}$*. This uninhabited type has index erasure* $\texttt{Unit} \triangleq \forall X^* . X \to X$*.*

Theorem 1 (index erasure on well-sorted kinds)**.** $\dfrac{\vdash \kappa : \Box}{\vdash \kappa^\circ : \Box}$

Proof. By induction on the sorting derivation. $\Box$

Remark 1. For any well-sorted kind κ in F_i, κ° is a kind in F_ω.

Theorem 2 (index erasure on well-formed type level contexts)**.**

$$\frac{\vdash \Delta}{\vdash \Delta^\circ}$$

Proof. By induction on the derivation for well-formed type level context and using Theorem 1. $\square$

Remark 2. For any well-formed type level context Δ in F_i, Δ° is a well-formed type level context in F_ω.

Theorem 3 (index erasure on kind equality)**.** $\dfrac{\vdash \kappa = \kappa' : \square}{\vdash \kappa^\circ = \kappa'^\circ : \square}$

Proof. By induction on the kind equality judgement. $\square$

Remark 3. For any well-sorted kind equality $\vdash \kappa = \kappa' : \square$ in F_i, $\vdash \kappa^\circ = \kappa'^\circ : \square$ is a well-sorted kind equality in F_ω.

The three theorems above on kinds are rather simple to prove since there is no need to consider mutual recursion in the definition of kinds due to the erasure operation on kinds. Recall that the erasure operation on kinds discards the type (A) appearing in the index arrow type $(A \to \kappa)$. So, there is no need to consider the types appearing in kinds and the index terms appearing in those types, after the erasure.

Theorem 4 (index erasure on well-kinded type constructors)**.**

$$\frac{\vdash \Delta \qquad \Delta \vdash F : \kappa}{\Delta^\circ \vdash F^\circ : \kappa^\circ}$$

Proof. By induction on the kinding derivation.

case (Var) Use Theorem 2.
case $(Conv)$ By induction and using Theorem 3.
case (λ) By induction and using Theorem 1.
case $(@)$ By induction.
case (λi)
We need to show that $\Delta^\circ \vdash (\lambda i^A.F)^\circ : (A \to \kappa)^\circ$, which simplifies to $\Delta^\circ \vdash F^\circ : \kappa^\circ$ by Definition 1.
By induction, we know that $(\Delta, i^A)^\circ \vdash F^\circ : \kappa^\circ$, which simplifies $\Delta^\circ \vdash F^\circ : \kappa^\circ$ by Definition 1.
case $(@i)$
We need to show that $\Delta^\circ \vdash (F\ \{s\})^\circ : \kappa^\circ$, which simplifies to $\Delta^\circ \vdash F^\circ : \kappa^\circ$ by Definition 1.
By induction we know that $\Delta^\circ \vdash F^\circ : (A \to \kappa)^\circ$, which simplifies to $\Delta^\circ \vdash F^\circ : \kappa^\circ$ by Definition 1.
case $(\to)$ By induction.
case $(\forall)$
We need to show that $\Delta^\circ \vdash (\forall X^\kappa.B)^\circ : *^\circ$, which simplifies to $\Delta^\circ \vdash \forall X^{\kappa^\circ}.B^\circ : *$ by Definition 1.
Using Theorem 1, we know that $\vdash \kappa^\circ : \square$.
By induction we know that $(\Delta, X^\kappa)^\circ \vdash B^\circ : *^\circ$, which simplifies to $\Delta^\circ, X^{\kappa^\circ} \vdash B^\circ : *$ by Definition 1.
Using the kinding rule $(\forall)$, we get exactly what we need to show: $\Delta^\circ \vdash \forall X^{\kappa^\circ}.B^\circ : *$.
case $(\forall i)$
We need to show that $\Delta^\circ \vdash (\forall i^A.B)^\circ : *^\circ$, which simplifies to $\Delta^\circ \vdash B^\circ : *$ by Definition 1.
By induction we know that $(\Delta, i^A)^\circ \vdash B^\circ : *^\circ$, which simplifies $\Delta^\circ \vdash B^\circ : *$ by Definition 1. $\square$

Theorem 5 (index erasure on type constructor equality)**.**

$$\frac{\Delta \vdash F = F' : \kappa}{\Delta^\circ \vdash F^\circ = F'^\circ : \kappa^\circ}$$

Proof. By induction on the derivation of type constructor equality.

Most of the cases are done by applying the induction hypothesis and sometimes using Proposition 2.

The only interesting cases, which are worth elaborating, are the equality rules involving substitution. There are two such rules.

$$\frac{\Delta, X^\kappa \vdash F : \kappa' \qquad \Delta \vdash G : \kappa}{\Delta \vdash (\lambda X^\kappa.F)\, G = F[G/X] : \kappa'}$$

We need to show $\Delta^\circ \vdash ((\lambda X^\kappa.F)\, G)^\circ = (F[G/X])^\circ : \kappa'^\circ$, which simplifies to $\Delta^\circ \vdash (\lambda X^{\kappa^\circ}.F^\circ)\, G^\circ = (F[G/X])^\circ : \kappa'^\circ$ by Definition 1.

By induction, we know that $(\Delta, X^\kappa)^\circ \vdash F^\circ : \kappa'^\circ$, which simplifies to $\Delta^\circ, X^{\kappa^\circ} \vdash F^\circ : \kappa'^\circ$. by Definition 1.

Using the kinding rule (λ), we get $\Delta^\circ \vdash \lambda X^{\kappa^\circ} F^\circ : \kappa^\circ \to \kappa'^\circ$.

Using the kinding rule $(@)$, we get $\Delta^\circ \vdash (\lambda X^{\kappa^\circ} F^\circ) G^\circ : \kappa^\circ$.

Using the very equality rule of this case,
we get $\Delta^\circ \vdash (\lambda X^{\kappa^\circ} F^\circ) G^\circ = F^\circ[G^\circ/X] : \kappa^\circ$.

All we need to check is $([G/X]F)^\circ = F^\circ[G^\circ/X]$, which is easy to see.

$$\frac{\Delta, i^A \vdash F : \kappa \qquad \Delta; \cdot \vdash s : A}{\Delta \vdash (\lambda i^A.F)\, \{s\} = F[s/i] : \kappa}$$

By induction we know that $\Delta^\circ \vdash F^\circ : \kappa^\circ$.

The erasure of the left hand side of the equality is
$((\lambda i^A.F)\, \{s\})^\circ = (\lambda i^A.F)^\circ = F^\circ$.

All we need to show is $(F[s/i])^\circ = F^\circ$, which is obvious since index variables can only occur in index terms and index terms are always erased. Recall the index erasure over type constructors in Definition 1, in particular, $(\lambda i^A.F)^\circ = F^\circ$, $(F\{s\})^\circ = F^\circ$, and $(\forall i^A.B)^\circ = B^\circ$. $\square$

Remark 4. For any well-kinded type constructor equality $\Delta \vdash F = F' : \kappa$ in F_i, $\Delta^\circ \vdash F^\circ = F'^\circ : \kappa^\circ$ is a well-kinded type constructor equality in F_ω.

The proofs for the two theorems above on type constructors need not consider mutual recursion in the definition of type constructors due to the erasure operation. Recall that the erasure operation on type constructors discards the index term (s) appearing in the index application $(F\ \{s\})$. So, there is no need to consider the index terms appearing in the types after the erasure.

Theorem 6 (index erasure on well-formed term level contexts)**.**

$$\frac{\Delta \vdash \Gamma}{\Delta^\circ \vdash \Gamma^\circ}$$

Proof. By induction on Γ.

case $(\Gamma = \cdot)$ It trivially holds.
case $(\Gamma = \Gamma', x : A)$, we know that $\Delta \vdash \Gamma'$ and $\Delta \vdash A : *$ by the well-formedness rules and that $\Delta^\circ \vdash \Gamma'^\circ$ by induction.
From $\Delta \vdash A : *$, we know that $\Delta^\circ \vdash A^\circ : *$ by Theorem 4.
We know that $\Delta^\circ \vdash \Gamma'^\circ, x : A^\circ$ from $\Delta^\circ \vdash \Gamma'^\circ$ and $\Delta^\circ \vdash A^\circ : *$ by the well-formedness rules.
Since $\Gamma'^\circ, x : A^\circ = (\Gamma', x : A)^\circ = \Gamma^\circ$ by definition, we know that $\Delta^\circ \vdash \Gamma^\circ$. $\square$

Theorem 7 (index erasure on index-free well-typed terms)**.**

$$\frac{\Delta \vdash \Gamma \qquad \Delta; \Gamma \vdash t : A \qquad \mathsf{dom}(\Delta) \cap \mathrm{FV}(t) = \emptyset}{\Delta^\circ; \Gamma^\circ \vdash t : A^\circ}$$

Proof. By induction on the typing derivation. Interesting cases are the index related rules $(:\, i)$, $(\forall Ii)$, and $(\forall Ei)$. Proofs for the other cases are straightforward by induction and applying other erasure theorems corresponding to the judgment forms.

case (:) By Theorem 6, we know that $\Delta^\circ \vdash \Gamma^\circ$ when $\Delta \vdash \Gamma$. By definition of erasure on term-level context, we know that $x : A^\circ \in \Gamma^\circ$ when $x : A \in \Gamma$.

case (: i) Vacuously true since t does not contain any index variables (i.e., $\mathsf{dom}(\Delta) \cap \mathrm{FV}(t) = \emptyset$).

case ($\rightarrow I$) By Theorem 4, we know that $\circ \vdash A^\circ : *$. By induction, we know that $\Delta^\circ; \Gamma^\circ, x : A^\circ \vdash t^\circ : B^\circ$. Applying the ($\rightarrow I$) rule to what we know, we have $\Delta^\circ; \Gamma^\circ \vdash \lambda x.t^\circ : A^\circ \rightarrow B^\circ$.

case ($\rightarrow E$) Straightforward by induction.

case ($\forall I$) By Theorem 1, we know that $\vdash \kappa^\circ : \Box$. By induction, we know that $\Delta^\circ, X^{\kappa^\circ}; \Gamma^\circ \vdash t : B^\circ$. Applying the ($\forall I$) rule to what we know, we have $\Delta^\circ; \Gamma^\circ \vdash t : \forall X^{\kappa^\circ}.B^\circ$.

case ($\forall E$) By induction, we know that $\Delta^\circ; \Gamma^\circ \vdash t : \forall X^{\kappa^\circ}.B^\circ$. By Theorem 4, we know that $\Delta^\circ \vdash G^\circ : \kappa^\circ$. Applying the ($\forall E$) rule, we have $\Delta^\circ; \Gamma^\circ \vdash t : B^\circ[G^\circ/X]$.

case ($\forall Ii$) By Theorem 4, we know that $\circ \vdash A^\circ : *$. By induction, we know that $\Delta^\circ; \Gamma^\circ \vdash t : B^\circ$, which is what we want since $(\forall i^A.B)^\circ = B^\circ$.

case ($\forall Ei$) By induction, we know that $\Delta^\circ; \Gamma^\circ \vdash t : B^\circ$, which is what we want since $(B[s/i])^\circ = B^\circ$.

case (=) By Theorem 5 and induction. □

Remark 5. The pathological considerations of Example **??** show that the implication of the theorem does not admit a converse.

Definition 2 (index variable selection)**.**

$$\cdot^\bullet = \cdot \qquad (\Delta, X^A)^\bullet = \Delta^\bullet \qquad (\Delta, i^A)^\bullet = \Delta^\bullet, i^A$$

The index variable selection operation ($^\bullet$) selects all the index variable bindings from the type level context.

Theorem 8 (index erasure on well-formed term level contexts prepended by index variable selection)**.**

$$\frac{\Delta \vdash \Gamma}{\Delta^\circ \vdash (\Delta^\bullet, \Gamma)^\circ}$$

Proof. Straightforward by Theorem 6 and the typing rule (: i). □

Theorem 9 (index erasure on well-typed terms)**.**

$$\frac{\Delta \vdash \Gamma \qquad \Delta; \Gamma \vdash t : A}{\Delta^\circ; (\Delta^\bullet, \Gamma)^\circ \vdash t : A^\circ}$$

Proof. The proof is almost the same as as Theorem 7, except for the (: i) case. The proof for the (: i) case is easy since $i^A \in \Delta^\bullet$ when $i^A \in \Delta$ by definition of the index variable selection operation. The indices from Δ being prepended to Γ do not affect the proof for the other cases. □

5.3 Strong normalization and logical consistency

Strong normalization is a corollary of the erasure property since we know that System F_ω is strongly normalizing. Logical consistency is immediate since System F_i is a strict subset of the *restricted implicit calculus* [12], which is a restriction of ICC [13]. Subject reduction is also immediate since F_i is a strict subset of ICC.

5.4 TODO about void type instantiation TODO

Why did we bother to design F_i? What is different from a Curry-style dependent calculus with implicit arguments such as ICC? The following rule is the instantiation

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash t : \forall x^A.B}\ (x \notin \mathrm{FV}(t))$$

Consider when $A = Void$ and $B = \forall i^{Void}.NeverEverVoid\,\{i\}$. In the calculus above, we can instantiate i with y provided that $(y : Void) \in \Gamma$. It is a void type instantiation, Uh-Oh ...

In F_i we cannot instantiate B with any of the term variables since index instantiation can not refer to the term-level context but only refer to the type-level context Δ. Recall

$$(\forall Ei)\frac{\Delta; \Gamma \vdash t : \forall i^A.B \qquad \Delta; \cdot \vdash s : A}{\Delta; \Gamma \vdash t : B[s/i]}$$

Proposition 4 (anti-dependency on arrow kinds)**.**

$$\frac{\vdash \Delta, X^\kappa \qquad \Delta, X^\kappa \vdash F : \kappa'}{X \notin \mathrm{FV}(\kappa')}$$

Proof. By Proposition 2, $\vdash \kappa'$. Note that $\vdash \kappa'$ does not involve any type level context.

Therefore, X cannot appear free in κ'. □

Proposition 5 (anti-dependency on indexed arrow kinds)**.**

$$\frac{\vdash \Delta, i^A \qquad \Delta, i^A \vdash F : \kappa}{i \notin \mathrm{FV}(\kappa)}$$

Proof. By Proposition 2, $\vdash \kappa'$. Note that $\vdash \kappa'$ does not involve any type level context. Therefore, i cannot appear free in κ'. □

Proposition 6 (anti-dependency on arrow types)**.**

$$\frac{\Delta \vdash \Gamma, x : A \qquad \Delta; \Gamma, x : A \vdash t : B}{x \notin \mathrm{FV}(B)}$$

Proof. By Proposition 3, $\Delta \vdash B : *$. Note that $\Delta \vdash \kappa'$ does not involve any term level context. Therefore, x cannot appear free in B. □

Remark 6. Our system is more strong??? than anti-dependency on arrow types TODO

6. Related work

Among theoretical calculi, F_i is most closely related to Curry-style System F_ω[1, 2, 7] and Implicit Calculus of Constructions (ICC) [13]. All terms typable in Curry-style System F_ω are typable in System F_i with the same type, and all terms typable in F_i are typable in ICC with the same type.[5] We have discussed that we can derive strong normalization, logical consistency, and subject reduction of F_i, from F_ω and a subset of ICC. In fact, ICC is more than just an extension of F_i, as described in our work, with dependent types and stratified universes. ICC includes η-reduction and the extensionality typing rule. We do not foresee any problem of adding η-reduction and the extensionality typing rule to F_i. Although System F_i accepts less terms than ICC, F_i enjoys more simple erasure properties (Theorem 7 and Theorem 9), which ICC cannot not provide due to its support for full dependent types. In System F_i, index terms appearing in types (e.g., s in $F\{s\}$) are always erasable. Mishra-Linger and Sheard [14] formalized a more generic framework than ICC, which describes the erasure on arbitrary Church-style calculi (EPTS) and Curry-style calculi (IPTS), but only consider β-equivalence for type conversion.

In §3.1, we have mentioned that Curry-style calculi enjoys better reduction properties (e.g.,$\beta\eta$-reduction is Church-Rosser) than Church-style calculi. Nederpelt [16] showed a counterexample to the Church-Rosser property for $\beta\eta$-reduction of Church-style terms. Geuvers [6] proved that $\beta\eta$-reduction is Church-Rosser in functional PTSs, which is a certain class of Church-style calculi.

[5] The $*$ kind in F_ω and F_i corresponds to Set in ICC.

Seldin [17] discusses the relation between the Church-style typing and the Curry-style typing.

Stephanie's Rw and related work. they prove parametricity in the presense of indices of GADTs

In a more practical setting for language implementation,

Yorgey et al. [20], inspired by McBride [11], designed a language extension to Haskell, promoting datatypes to be used as kinds. For instance, `Bool` is promoted to a kind (i.e., `Bool` : $\Box$) and its data constructors `Ture` and `False` are promoted to type level. To support this in GHC, they extended System F_C (GHC's intermediate language, or, GHC Core), naming the extended GHC Core as System $F_C^{\uparrow}$. The key difference between $F_C^{\uparrow}$ and F_i is in the extension to the kind syntax, as illustrated below:

$$\begin{array}{rl} F_C^{\uparrow}\ \textbf{kinds}: & \kappa ::= * \mid \kappa \to \kappa \mid F\vec{\kappa} \mid \mathcal{X} \mid \forall\mathcal{X}.\kappa \mid \cdots \\ \mathsf{F}_i\ \textbf{kinds}: & \kappa ::= * \mid \kappa \to \kappa \mid A \to \kappa \end{array}$$

In $F_C^{\uparrow}$, any type constructor (F) is promoted to kind level and becomes a kind when fully applied to other kinds ($F\vec{\kappa}$). In F_i, on the other hand, a type can only appear at the domain of an index arrow kind ($A \to \kappa$). This seemingly small difference to the kind syntax makes $F_C^{\uparrow}$ to develop into a drastically more expressive language than F_i. The promotion of a type constructor, for instance, `List` : $* \to *$ to a kind constructor `List` : $\Box \to \Box$ enables type-level data structures such as $[\mathtt{Nat}, \mathtt{Bool}, \mathtt{Nat} \to \mathtt{Bool}] : \mathtt{List}\, *$. Having type-level data structure motivates type-level computation over promoted data, which they make it possible by extending the type families[6]. In addition, the promotion of polymorphic types naturally motivates kind polymorphism ($\forall\mathcal{X}.\kappa$), which is known to break strong normalization and cause logical inconsistency [8]. For the purpose of extending a functional programming language, inconsistency is not an issue. However, for our interest of studying term-indexed datatypes in a logically consistent calculi, we need a more conservative approach, as in F_i, starting from smallest possible extension that maintains normalization and consistency.

Swamy et al. [19] value dependent types in F-star from MSR

what others to discuss?

Translating Generalized Algebraic Data Types to System F Martin Sulzmann and Meng Wang

7. Conclusion and Future work

TODO

We are also developing a programming language Nax, which supports type inference with little annotation, based on System F_i.

We wonder what extension we need to enable large eliminations (i.e., computing types from term-indices).

We are exploring whether Leibniz equality over indices (i.e., $s_1 = s_2$ encoded as $\forall X^{A\to *}.X\{s_1\} \to X\{s_2\}$) may help us express functions whose domains are restricted by term-indices (e.g., `vtail :: Vec a {S n} -> Vec a n`). **[MF: Omit this? As I added the, I should say experimental, §§4.3 on this.]**

A. Appendix

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

References

[1] A. Abel, R. Matthes, and T. Uustalu. Generalized iteration and coiteration for higher-order nested datatypes. In *FoSSaCS*, volume 2620 of *LNCS*, pages 54–69. Springer, 2003.

[2] A. Abel, R. Matthes, and T. Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theoretical Computer Science*, 333(1-2):3 – 66, 2005.

[3] K. Y. Ahn and T. Sheard. A hierarchy of Mendler-style recursion combinators: Taming inductive datatypes with negative occurrences. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 234–246, New York, NY, USA, 2011. ACM.

[4] C. Böhm and A. Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[5] A. Church. A set of postulates for the foundation of logic (2nd paper). *Annals of Mathematics*, 34(4):839–864, Oct. 1933.

[6] H. Geuvers. The Church-Rosser property for $\beta\eta$-reduction in typed λ-calculi. In *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 453–460, June 1992.

[7] P. Giannini, F. Honsell, and S. R. D. Rocca. Type inference: Some results, some problems. *Fundam. Inform.*, 19(1/2):87–125, 1993.

[8] Girard, J.-Y. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII, June 1972.

[9] L.-J. Guillemette. *A Type-Preserving Compiler from System F to Typed Assembly Language*. PhD thesis, Oct. 2010.

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

[11] C. McBride. The Strathclyde Haskell Enhancement. URL `http://personal.cis.strath.ac.uk/conor/pub/she/`.

[12] A. Miquel. A model for impredicative type systems, universes, intersection types and subtyping. In *LICS*, pages 18–29. IEEE Computer Society, 2000.

[13] A. Miquel. The implicit calculus of constructions. In *TLCA*, pages 344–359, 2001.

[14] N. Mishra-Linger and T. Sheard. Erasure and polymorphism in pure type systems. In R. M. Amadio, editor, *FoSSaCS*, volume 4962 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2008.

[15] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[16] R. P. Nederpelt. *Strong Normalization in a Typed Lambda Calculus with Lambda Structured Types*. PhD thesis, Eindhoven University of Technology, 1973.

[17] J. P. Seldin. On the relation between the Church-style typing and the Curry-style typing. Submitted to Journal of Applied Logic. URL `http://people.uleth.ca/~jonathan.seldin/RCS2.pdf`.

[18] T. Sheard and E. Pasalic. Two-level types and parameterized modules. *J. Funct. Program.*, 14(5):547–587, Sept. 2004. ISSN 0956-7968.

[19] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 266–278, New York, NY, USA, 2011. ACM.

[20] B. A. Yorgey, S. Weirich, J. Cretin, S. L. P. Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *TLDI*, pages 53–66. ACM, 2012.

[6] A GHC extension to define type-level functions in Haskell