System F_i
A Higher-Order Polymorphic λ-Calculus with Erasable Term-Indices

Ki Yung Ahn[1], Tim Sheard[1], Marcelo Fiore[2], and Andrew M. Pitts[2]

[1] Portland State University, Portland, Oregon, USA*
{kya,sheard}@cs.pdx.edu
[2] University of Cambridge, Cambridge, UK
{Marcelo.Fiore,Andrew.Pitts}@cl.cam.ac.uk

Abstract. We introduce a foundational lambda calculus, System F_i, for studying programming languages with term-indexed datatypes – higher-kinded datatypes whose indices range over data such as natural numbers or lists. System F_i is an extension of System F_ω that introduces the minimal features needed 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 logically-sound light-weight dependent programming languages. 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 strong normalization and logical consistency from System F_ω.

Keywords: term-indexed data types, generalized algebraic data types, higher-order polymorphism, type-constructor polymorphism, higher-kinded types, impredicative encoding, strong normalization, logical consistency.

1 Introduction

We are interested in the use of indexed types to state and maintain program properties. A type parameter (like `Int` in `(List Int)`) usually tells us something about data stored in values of that type. A type-index (like `3` in `(Vector Int 3)`) states an inductive property of values with that type. For example, values of type `(Vector Int 3)` have three elements.

Indexed types come in two flavors: *type-indexed* and *term-indexed* types.

An example of type-indexing is a definition of a *representation type* [8] using GADTs in Haskell:

```
data TypeRep t where
  RepInt  :: TypeRep Int
  RepBool :: TypeRep Bool
  RepPair :: TypeRep a -> TypeRep b -> TypeRep (a,b)
```

* Supported by NSF grant 0910500.

Here, a value of type `(TypeRep t)` is isomorphic in shape with the type-index `t`. For example, `(RepPair RepInt RepBool) :: TypeRep (Int,Bool)`.

An example of *Term-indices* are datatypes with indices ranging over data structures, such as natural numbers (like `Z`, (`S Z`)) or lists (like `Nil` or (`Cons Z Nil`)). A 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[1] or OCaml [10], which support GADTs with only type-indexing, term-indices are simulated (or faked) by reflecting data at the type-level with uninhabited type constructors. For example,

```
data S n
data Z
data Vec t n where
  Cons :: a -> Vec a n -> Vec a (S n)
  Nil  :: Vec a Z
```

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

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 in a higher-order polymorphic λ-calculus to embed term-indexed datatypes (Sect. 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 (Sect. 4), and its use to study properties of languages with term-indexed datatypes, including the embedding of term-indexed datatypes into the calculus (Sect. 6) using Church or Mendler style encodings, and proofs about these encodings. For instance, one can use System F_i to prove that the Mendler-style eliminators for GADTs [3] are normalizing.
- Showing that System F_i enjoys a simple erasure property (Sect. 5.2) and inherits meta-theoretic results, strong normalization and logical consistency, from F_ω (Sect. 5.3).

2 Motivation: 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 [5].

[1] See Sect. 7 for a very recent GHC extension, which enable true term-indices.

In System F, one can embed *regular datatypes*, like homogeneous lists:

Haskell: `data List a = Cons a (List a) | Nil`

System F: $\texttt{List}\ A \triangleq \forall X.(A \to X \to X) \to X \to X$

$$\texttt{Cons} \triangleq \lambda w.\lambda x.\lambda y.\lambda z.\, y\ w\,(x\ y\ z),\ \texttt{Nil} \triangleq \lambda y.\lambda z.z$$

In such regular datatypes, constructors have algebraic structure that directly translates into polymorphic operations on abstract types as encapsulated by universal quantification over types (of kind $*$).

In the more expressive System F_ω (where one can abstract over type constructors of any kind), one can encode more general *type-indexed datatypes* that go beyond the regular datatypes. 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 = PCons a (Powl(a,a)) | PNil`

```
-- PCons 1 (PCons (2,3) (PCons ((3,4),(1,2)) PNil)) :: Powl Int
```

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

Note the non-regular occurrence (`Powl(a,a)`) in the definition of (`Powl a`), and the use of universal quantification over higher-order kinds ($\forall X^{*\to *}$). The term encodings for `PCons` and `PNil` are exactly the same as the term encodings for `Cons` and `Nil`, but have different types.

What about term-indexed datatypes? What extensions to System F_ω are needed to embed term-indices as well as type-indices? 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 (in Nax[2]-like syntax):

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

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 natural numbers. By design, our syntax directly reflects the difference between type and term-indexing by enclosing the latter in curly braces. We also make this distinction in System F_i, where it is useful within the type system to guarantee the static nature of term-indexing.

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

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

where `Nat`, `Z`, and `S` respectively encode the natural number type and its two constructors, zero and successor. Again, the term encodings for `VCons` and `VNil` are exactly the same as the encodings for `Cons` and `Nil`, but have different types.

[2] We are developing a language called Nax whose theory is based on System F_i.

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 (see Fig. 1 for the highlighted extended syntax).

1. Type-indexed kinding ($A \to \kappa$), as in ($\texttt{Nat}\to\texttt{*}$) in the example above, where the compile-time nature of term-indexing will be reflected in the typing rules, enforcing that A be a closed type (rule (Ri) in Fig. 2).
2. Term-index abstraction $\lambda i^A.F$ (as $\lambda i^{\texttt{Nat}}.\cdots$ in the example above) for constructing (or introducing) term-indexed kinds (rule (λi) in Fig. 2).
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) term-indexed kinds, where the compile-time nature of indexing will be reflected in the typing rules, enforceing that the index be statically typed (rule $(@i)$ in Fig. 2) .
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 typing rules enforcing that the variable i be static of closed type A (rule $(\forall Ii)$ in Fig. 2).

As described above, System F_i maintains a clear-cut separation between type-indexing and term-indexing. This adds a level of abstraction to System F_ω and yields types that in addition to parametric polymorphism also keep track of inductive invariants using term-indices. All term-index information can be erased, since it is only used at compile-time. It is possible to project any well-typed System F_i term into a well-typed System F_ω term. For instance, the erasure of the F_i-type `Vec` is the F_ω-type `List`. This is established in Sect. 5 and used to deduce the strong normalization of System F_i.

3 Why Term-Indexed Calculi? (Rather Than Dependent Types)

We claim that a moderate extension to the polymorphic calculus (F_ω) is a better candidate than a dependently typed calculus for the basis of a practical programming system. We hope to design a unified system for programming as well as reasoning. Language designs based on indexed types can benefit from existing compiler technology and type inference algorithms for functional programming languages. In addition, theories for term-indexd datatypes are simpler than theories for full-fledged dependent datatypes, because term-indexd datatypes can be encoded as functions (using Church-like encodings).

The implementation technology for functional programming languages based on polymorphic calculi is quite mature. The industrial strength Glasgow Haskell Compiler (GHC), whose intermediate core language is an extension of F_ω, is used by thousands every day. Our term-indexed calculus F_i is closely related to F_ω by an index-erasure property. The hope is that a language implementation based on F_i can benefit from these technologies. We have built a language implementation of these ideas, which we call Nax.

Type inference algorithms for functional programming languages are often based on certain restrictions of the Curry-style polymorphic lambda calculi. These restrictions are designed to avoid higher-order unification during type inference. We have developed a conservative extension of Hindley–Milner type inference for Nax. This was possible because Nax is based on a restricted F_i. Dependently typed languages, on the other hand, are often based on bidirectional type checking, which requires annotations on top level definitions, rather than Hindley–Milner-style type inference.

In dependent type theories, datatypes are usually introduced as primitive constructs (with axioms), rather than as functional encodings (e.g., Church encodings). One can give functional encodings for datatypes in a dependent type theory, but one soon realizes that the induction principles (or, dependent eliminators) for those datatypes cannot be derived within the pure dependent calculi [11]. So, dependently typed reasoning systems support datatypes as primitives. For instance, Coq is based on Calculus of Inductive Constructions, which extends Calculus of Constructions [7] with dependent datatypes and their induction principles.

In contrast, in polymorphic type theories, all imaginable datatypes within the calculi have functional encodings (e.g., Church encodings). For instance, F_ω need not introduce datatypes as primitive constructs, since F_ω can embed all these datatypes, including non-regular recursive datatypes with type indices.

Another reason to use F_i is to extend the application of Mendler-style recursion schemes, which are well-understood in the context of polymorphic lambda calculi like F_ω. Researchers have thought about (though not published)[3] Mendler-style primitive recursion over dependently-typed functions over positive datatypes (i.e., datatypes that have a map), but not for negative (or, mixed-variant) datatypes. In System F_i, we can embed Mendler-style recursion schemes, (just as we embedded them in F_ω) that are also well-defined for negative datatypes.

4 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 Figs. 1, 2 and 3. 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 Figs. 1, 2 and 3, one obtains a version of System F_ω with Curry-style terms and the typing context separated into two parts (type-level context Δ and term-level context Γ).

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

Kinds (Fig. 1). 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.

[3] Tarmo Uustalu described this on a whiteboard when we met with him at the University of Cambridge in 2011.

$$
\begin{array}{llrcl}
\textit{Syntax:} & \text{Term Variables} & & & x, y, z, \ldots, i, j, k, \ldots \\
& \text{Type Constructor Variables} & & & X, Y, Z, \ldots \\
& \text{Sort} & & & \Box \\
& \text{Kinds} & \kappa & ::= & * \mid \kappa \to \kappa \mid A \to \kappa \\
& \text{Type Constructors} & A, B, F, G & ::= & X \mid A \to B \mid \lambda X^{\kappa}.F \mid F\,G \mid \forall X^{\kappa}.B \\
& & & & \mid \lambda i^{A}.F \mid F\,\{s\} \mid \forall i^{A}.B \\
& \text{Terms} & r, s, t & ::= & x \mid \lambda x.t \mid r\ s \\
& \text{Typing Contexts} & \Delta & ::= & \cdot \mid \Delta, X^{\kappa} \mid \Delta, i^{A} \\
& & \Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

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

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

Fig. 1. Syntax and Reduction rules of F_i

Sorting (Fig. 2). 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 A to be well-kinded under the empty type-level context avoids dependent kinds (i.e., kinds depending on type-level or value-level bindings). That is, A should 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 since X appears free, while $((\forall X^{*}.\,\mathit{List}\,X) \to *)$ is a well-sorted kind.

Typing contexts (Fig. 1). 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. Note, both x and i represent the same syntactic category of "Type Variables". The distinction between x and i is only a convention for the sake of readability.

Well-formed typing contexts (Fig. 2). 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 a closed type (not a type constructor!) without free variables.

Well-formed typing contexts:

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

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

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

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

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

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

$$(\lambda)\frac{\vdash \kappa : \Box \quad \Delta, X^\kappa \vdash F : \kappa'}{\Delta \vdash \lambda X^\kappa.F : \kappa \to \kappa'} \quad (\lambda i)\frac{\cdot \vdash A : * \quad \Delta, i^A \vdash F : \kappa}{\Delta \vdash \lambda i^A.F : A \to \kappa}$$

$$(@)\frac{\Delta \vdash F : \kappa \to \kappa' \quad \Delta \vdash G : \kappa}{\Delta \vdash F\,G : \kappa'} \quad (@i)\frac{\Delta \vdash F : A \to \kappa \quad \Delta; \cdot \vdash s : A}{\Delta \vdash F\,\{s\} : \kappa}$$

$$(\forall)\frac{\vdash \kappa : \Box \quad \Delta, X^\kappa \vdash B : *}{\Delta \vdash \forall X^\kappa.B : *} \quad (\forall i)\frac{\cdot \vdash A : * \quad \Delta, i^A \vdash B : *}{\Delta \vdash \forall i^A.B : *}$$

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

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

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

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

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

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

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

Fig. 2. Well-formedness, Sorting, Kinding, and Typing rules of F_i

Kind equality: $\boxed{\vdash \kappa = \kappa' : \Box}$ $\quad \dfrac{\cdot \vdash A = A' : * \qquad \vdash \kappa = \kappa' : \Box}{\vdash A \to \kappa = A' \to \kappa' : \Box}$

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

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

$$\frac{\Delta \vdash F = F' : A \to \kappa \qquad \Delta; \cdot \vdash s = s' : A}{\Delta \vdash F\,\{s\} = F'\,\{s'\} : \kappa}$$

Term equality: $\boxed{\Delta; \Gamma \vdash t = t' : A}$ $\quad \dfrac{\Delta; \Gamma, x : A \vdash t : B \qquad \Delta; \Gamma \vdash s : A}{\Delta; \Gamma \vdash (\lambda x.t)\, s = t[s/x] : B}$

Fig. 3. Equality rules of F_i (only the key rules are shown)

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 (Figs. 1 and 2). We extend the type constructor syntax by three constructs, and extend the kinding rules accordingly.

$\lambda i^{A}.F$ 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 term-index application. In contrast to the ordinary type-level type-application $(F\, G)$ where the argument (G) is a type (of arbitrary kind). The argument of an term-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 a term-index, which is erasable. Index applications eliminate indexed arrow kinds by the kinding rule $(@i)$. Note, we type check the term-index (s) under the current type-level context paired with the empty term-level context $(\Delta; \cdot)$ since we do not want the term-index (s) to depend on any term-level bindings. Otherwise, we would admit value dependencies in 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 (see Fig. 3). For instance, $A \to \kappa$ and $A' \to \kappa$ equivalent kinds when A' and A are equivalent types. Only the key equality rules are shown in

Fig. 3, and the other structural rules (one for each sorting/kinding/typing rule) and the congruence rules (symmetry, transitivity) are omitted.

Terms and their typing rules (Figs. 1 and 2). 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 only for 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. These rules are similar to the type generalization ($\forall I$) and the type instantiation ($\forall I$) rules, but involve indices, rather than types, and have additional side conditions compared to their type counterparts.

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 polymorphically, but instead would behave as a dependent function binder, which are usually denoted by Π in dependent type theories. Such side conditions on generalization rules for polymorphism are fairly standard in dependent type theories that distinguish between polymorphism (or, erasable arguments) and dependent functions (e.g., IPTS[17], ICC[16]).

The index instantiation rule ($\forall Ei$) requires that the term-index s, which instantiates i, be well-typed in the current type-level context paired with the empty term-level context ($\Delta; \cdot$) rather than the current term-level context, since we do not want indices to 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 access index variables already in scope. In examples like ($\lambda i^A.F\{s\}$) and ($\forall i^A.F\{s\}$), the term (s) should be able to access the index variable (i) already in scope.

5 Metatheory

The expectation is that System F_i has all the nice properties of System F_ω, yet is more expressive (i.e., can state finer grained program properties) because of the addition of term-indexed types.

We show some basic well-formedness properties for the judgments of F_i in Sect. 5.1. We prove erasure properties of F_i, which capture the idea that indices are erasable since they are irrelevant for reduction in Sect. 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 Sect. 5.3.

5.1 Well-Formedness and Substitution Lemmas

We want to show that kinding and typing derivations give well-formed results under well-formed contexts. That is, kinding derivations ($\Delta \vdash F : \kappa$) result in well-sorted kinds ($\vdash \kappa$) under well-formed type-level contexts ($\vdash \Delta$) (Proposition 1), and typing derivations ($\Delta; \Gamma \vdash t : A$) result in well-kinded types ($\Delta; \Gamma \vdash A : *$) under well-formed type and term-level contexts (Proposition 2).

Proposition 1. $\dfrac{\vdash \Delta \quad \Delta \vdash F : \kappa}{\vdash \kappa : \Box}$ **Proposition 2.** $\dfrac{\Delta \vdash \Gamma \quad \Delta; \Gamma \vdash t : A}{\Delta \vdash A : *}$

We can prove these well-formedness properties by induction over the judgment[4] and using the substitution lemma below.

Lemma 1 (substitution)

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

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

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

These substitution lemmas are fairly standard, comparable to substitution lemmas in other well-known systems such as F_ω or ICC.

5.2 Erasure Properties

We define a meta-operation of index erasure that projects F_i-types to F_ω-types.

Definition 1 (index erasure)

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

$\boxed{F^\circ}$

$$\begin{array}{ll}
X^\circ = X & (A \to B)^\circ = A^\circ \to B^\circ \\
(\lambda X^{\kappa}.F)^\circ = \lambda X^{\kappa^\circ}.F^\circ & (\lambda i^{A}.F)^\circ = F^\circ \\
(F\ G)^\circ = F^\circ\ G^\circ & (F\ \{s\})^\circ = F^\circ \\
(\forall X^{\kappa}.B)^\circ = \forall X^{\kappa^\circ}.B^\circ & (\forall i^{A}.B)^\circ = B^\circ
\end{array}$$

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

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

In addition, we define another meta-operation, which selects out all the index variable bindings from the type-level context. We use this in Theorem 6.

[4] The proof for Propositions 1 and 2 are mutually inductive. So, we prove these two propositions at the same time, using a combined judgment J, which is either a kinding judgment or a typing judgment (i.e., $J ::= \Delta \vdash F : \kappa \mid \Delta; \Gamma \vdash t : A$).

Definition 2 (index variable selection)

$$\boxed{\Delta^\bullet} \qquad \cdot^\bullet = \cdot \qquad (\Delta, X^\kappa)^\bullet = \Delta^\bullet \qquad (\Delta, i^A)^\bullet = \Delta^\bullet, i : A$$

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

Proof. By induction on the sort (κ). ■

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

Theorem 2 (index erasure on well-formed type-level contexts). $\dfrac{\vdash \Delta}{\vdash \Delta^\circ}$

Proof. By induction on the type-level context (Δ) and using Theorem 1. ■

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). $\dfrac{\vdash \kappa = \kappa' : \Box}{\vdash \kappa^\circ = \kappa'^\circ : \Box}$

Proof. By induction on the kind equality derivation $(\vdash \kappa = \kappa' : \Box)$. ■

Remark 3. For any well-sorted kind equality $\vdash \kappa = \kappa' : \Box$ in F_i, κ° and κ'° are the syntactically same F_ω kinds. Note that no variables can appear in the erased kinds by definition of the erasure operation on kinds.

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 $(\Delta \vdash F : \kappa)$. We use Theorem 2 in the (Var) case, Theorem 3 in the $(Conv)$ case, and Theorem 1 in the (λ) and $(\forall)$ cases. ■

Remark 4. In the theorem above, F° is a well-kinded type constructor in F_ω.

Lemma 2. $(F[G/X])^\circ = F^\circ[G^\circ/X]$ **Lemma 3.** $(F[s/i])^\circ = F^\circ$

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 the type constructor equality judgment $(\Delta \vdash F = F' : \kappa)$. We also use Proposition 1 and Lemmas 2 and 3. ■

Remark 5. When $\Delta \vdash F = F' : \kappa$ is a valid type constructor equality in F_i, $\Delta^\circ \vdash F^\circ = F'^\circ : \kappa^\circ$ is a valid type constructor equality in F_ω.

Theorem 6 (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. By induction on the term-level context (Γ) and using Theorem 4. ∎

Remark 6. We can also show that $\dfrac{\Delta \vdash \Gamma}{\Delta^\circ \vdash \Gamma^\circ}$ and prove Corollary 1 directly.

Theorem 7 (index erasure on well-typed terms). $\dfrac{\Delta \vdash \Gamma \quad \Delta; \Gamma \vdash t : A}{\Delta^\circ; (\Delta^\bullet, \Gamma)^\circ \vdash t : A^\circ}$

Proof. By induction on the typing derivation ($\Delta; \Gamma \vdash t : A$). We also make use of Theorems 1, 4, 5, and 6. ∎

Remark 7. In the theorem above, t is a well typed term in F_ω as well as in F_i.

Corollary 1 (index erasure on index-free well-typed terms)

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

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. Index erasure also implies logical consistency. By index erasure, we know that any well-typed term in F_i is a well-typed term in F_ω with its erased type. That is, there are no extra well-typed terms in F_i that are not well-typed in F_ω. By the saturated sets model (as in [1]), we know that the void type ($\forall X^*.X$) in F_ω is uninhabited. Therefore, the void type ($\forall X^*.X$) in F_i is uninhabited since it erases to the same void type in F_ω. Alternatively, logical consistency of F_i can be drawn from ICC. System F_i is a restriction of the *restricted implicit calculus* [15] or ICC$^-$ [4], which are restrictions of ICC [16] known to be logically consistent.

6 Encodings of Term-Indexed Datatypes

Recall that our motivation was a foundational calculus that can encode term-indexed datatypes. In Sect. 2, we gave Church encodings of `List` (a regular datatype), `Powl` (a type-indexed datatype), and `Vec` (a term-indexed datatype). In this section, we discuss a more complex datatype [6] involving nested term-indices, and several encoding schemes that we have seen used in practice – first, encoding indexed datatypes using equality constraints [8, 18] and second, encoding datatypes in the Mendler-style [2, 3].

Nested term-indices: System F_i is able to express datatypes with *nested term-indices* – term-indices which are themselves term-indexed datatypes. Consider the resource-state tracking environment [6] in Nax-like syntax below:

```
data Env : ({st} -> *) -> {Vec st {n}} -> * where
  Extend : res {x} -> Env res {xs} -> Env res {VCons x xs}
  Empty  : Env res {VNil}
```

Note that `Env` has a term-index of type `Vec`, which is again indexed by `Nat`. For simplicity,[5] assume that `n` is some fixed constant (e.g., `S(S(S Z))`, i.e., 3). Then, an `Env` tracks 3 independent resources (`res`), each which could be in a different state (`st`). For example, 3 files in different states – one open for reading, the next open for writing, and the third closed. We can encode `Env` in F_i as follows:

$$\begin{aligned}\texttt{Env} \triangleq \lambda Y^{\,\texttt{st}\to *}.\, \lambda i^{(\texttt{Vec st n})}.\, \forall X^{(\texttt{Vec st \{n\}})\to *}.&\\ (\forall j^{\texttt{st}}.\, \forall k^{(\texttt{Vec st n})}.\, Y\{j\} \to X\{k\} \to X\{\texttt{VCons}\; j\; k\}) \to X\{\texttt{VNil}\} \to X\{i\}&\end{aligned}$$

The term encodings for `Extend` and `Empty` are exactly the same as the term encodings for `Cons` and `Nil` of the `List` datatype in Sect. 2.

Encoding indexed datatypes using equality constraints: Systematic encodings of GADTs [8, 18], which are used in practical implementations, typically involve equality constraints and existential quantification. Here, we want to emphasize that such encoding schemes are expressible within System F_i, since it is possible to define equalities and existentials over both types and term-indices in F_i.

It is well known that Leibniz equality over type constructors can be defined within System F_ω as $(\overset{\kappa}{=}) \triangleq \lambda X_1^\kappa.\, \lambda X_2^\kappa.\, \forall X^{\kappa\to *}.\, X X_1 \to X X_2$. Similarly, Leibniz equality over term-indices is defined as $(\overset{A}{=}) \triangleq \lambda i^A.\, \lambda j^A.\, \forall X^{A\to *}.\, X\{i\} \to X\{j\}$ in System F_i. Then, we can encode `Vec` as the sum of its two data constructor types:

$$\texttt{Vec} \triangleq \lambda A^*.\, \lambda i^{\texttt{Nat}}.\, \forall X^{\texttt{Nat}\to *}.\, (\exists j^{\texttt{Nat}}.\, (\texttt{S}\; j \overset{\texttt{Nat}}{=} i) \times A \times X\{j\}) + (\texttt{Z} \overset{\texttt{Nat}}{=} i)$$

where $+$ and $\times$ are the usual impredicative encoding of sums and products. We can encode the existential quantification over indices ($\exists$ used in the encoding of `Vec` above) as $\exists i^A.B \triangleq \forall X^*.(\forall i^A.B \to X) \to X$, which is similar to the usual encoding of the existential quantification over types in System F or F_ω.

Compared to the simple Church encoded versions in Sect. 2, the encodings using equality constraints work particularly well with encodings of functions that constrain their domain types by restricting their indices. For instance, the function `safeTail` : `Vec` a $\{$`S` $n\}$ $\to$ `Vec` a $\{n\}$, which can only be applied to non-empty length indexed lists due the index of the domain type (`S` n).

[5] Nax supports rank-1 kind-level polymorphism. It would be virtually useless if nested term-indices were only limited to constants rather than polymorphic variables. We strongly believe rank-1 kind polymorphism does not introduce inconsistency, since rank-1 polymorphic systems are essentially equivalent to simply-typed systems by inlining the polymorphic definition with the instantiated arguments in each instantiation site.

The Mendler-style encoding: Recursive type theories that extend higher-order polymorphic lambda calculi typically come with a built-in recursive type operator $\mu_\kappa : (\kappa \to \kappa) \to \kappa$ for each kind κ, which yields recursive types $(\mu_\kappa\, F : \kappa)$ when applied to type constructors of appropriate kind $(F : \kappa \to \kappa)$. For instance, $\mathtt{List} \triangleq \lambda Y^{*}.\, \mu_{*}(\lambda X^{*}.Y \times X + \mathbb{1})$ where $\mathbb{1}$ is the unit type. One benefit of factoring out the recursion at type-level (e.g., μ_*) from the base structure (e.g., $\lambda X^{*}.Y \times X + \mathbb{1}$) of recursive types is that such factorized (or, two-level) representations are more amenable to express generic recursion schemes (e.g., catamorphism) that work over different recursive datatypes. Interestingly, there exists an encoding scheme, namely the Mendler style, which can embed μ_κ within Systems like F_ω or F_i. The Mendler-style encoding can keep the theoretical basis small, while enjoying the benefits of factoring out the recursion at type-level.

7 Related Work

System F_i is most closely related to Curry-style System F_ω [2, 12] and the Implicit Calculus of Constructions (ICC) [16]. All terms typable in a Curry-style System F_ω are typable (with the same type) in System F_i and all terms typable in F_i are typable (with the same type[6]) in ICC.

As mentioned in Sect. 5.3, we can derive strong normalization of F_i from System F_ω, and derive logical consistency of F_i from certain restrictions of ICC [4, 15]. In fact, ICC is more than just an extension of System F_i with dependent types and stratified universes, since ICC includes η-reduction and η-equivalence. We do not foresee any problems adding η-reduction and η-equivalence to System F_i. Although System F_i accepts fewer terms than ICC, it enjoys simpler erasure properties (Theorem 7 and Corollary 1) just by looking at the syntax of kinds and types, which ICC cannot enjoy due to its support for full dependent types. In System F_i, term-indices appearing in types (e.g., s in $F\{s\}$) are always erasable. Mishra-Linger and Sheard [17] generalized the ICC framework to one which describes erasure on arbitrary Church-style calculi (EPTS) and Curry-style calculi (IPTS), but only consider β-equivalence for type conversion.

In the practical setting of programming language implementation, Yorgey et al. [19], inspired by McBride [14], recently designed an extension to Haskell's GADTs by allowing datatypes to be used as kinds. For instance, `Bool` is promoted to a kind (i.e., $\mathtt{Bool} : \Box$) and its data constructors `True` and `False` are promoted to types. They extended System F_C (the Glasgow Haskell Compiler's intermediate core language) to support *datatype promotion* and named it System $F_C^{\uparrow}$. The key difference between $F_C^{\uparrow}$ and F_i is in their kind syntax:

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

In $F_C^{\uparrow}$, all type constructors (F) are promotable to the kind level and become kinds when fully applied to other kinds $(F\boldsymbol{\kappa})$. On the other hand, in F_i, a type

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

can only appear as the domain of an index arrow kind $(A \to \kappa)$. The ramifications of this difference is that $F_C^{\uparrow}$ can express type-level data structures but not nested term-indices, while F_i supports the converse. Intuitively, a type constructor like $\texttt{List} : * \to *$ is promoted to a kind constructor $\texttt{List} : \Box \to \Box$, which enables type-level data structures such as $[\texttt{Nat}, \texttt{Bool}, \texttt{Nat} \to \texttt{Bool}] : \texttt{List}\, *$. Type-level data structures motivate type-level computations over promoted data. This is made possible by type families[7]. The promotion of polymorphic types naturally motivates kind polymorphism $(\forall \mathcal{X}.\kappa)$. Kind polymorphism of arbitrary rank is known to break strong normalization and cause logical inconsistency [13]. In a *programming language*, inconsistency is not an issue. However, when studying logically consistent systems, we need a more conservative approach, as in F_i.

8 Summary and Ongoing Work

System F_i is a strongly-normalizing, logically-consistent, higher-order polymorphic lambda calculus that was designed to support the definition of datatypes indexed by both terms and types. In terms of expressivity, System F_i sits between System F_ω and ICC. We designed System F_i as a tool to reason about programming languages with term-indexed datatypes. System F_i can express a large class of term-indexed datatypes, including datatypes with nested term-indices.

One limitation of System F_i is that it cannot express type-level data structures such as lists that contain type elements. We hope to overcome this limitation by extending F_i with first-class type representations [9], which reflect types as term-level data (a sort of a fully reflective version of `TypeRep` from Sect. 1).

Our goal is to build a unified programming and reasoning system, which supports (1) an expressive class of datatypes including nested term-indexed datatypes and negative datatypes, (2) logically consistent reasoning about program properties, and (3) Hindley–Milner-style type inference. Towards this goal, we are developing the programming language Nax based on System F_i. Nax is given semantics in terms of System F_i. That is, all the primitive language constructs of Nax that are not present in F_i have translations into System F_i. Such constructs include Mendler-style eliminators, recursive type operators, and pattern matching.

Some language features we want to include in Nax go beyond F_i. One of them is a recursion scheme that guarantee normalization due to paradigmatic use of indices in datatypes. For instance, some recursive computations always reduce a natural number term-index in every recursive call. Although such computations obviously terminate, we cannot express them in System F_i, since term-indices in them are erasable – F_i only accepts terms that are already type-correct in F_ω. We plan to explore extensions to System F_i that enable such computations while maintaining logical consistency.

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

References

1. Abel, A., Matthes, R.: Fixed points of type constructors and primitive recursion. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 190–204. Springer, Heidelberg (2004)
2. Abel, A., Matthes, R., Uustalu, T.: Iteration and coiteration schemes for higher-order and nested datatypes. TCS 333(1-2), 3–66 (2005)
3. Ahn, K.Y., Sheard, T.: A hierarchy of Mendler-style recursion combinators: Taming inductive datatypes with negative occurrences. In: ICFP 2011, pp. 234–246. ACM (2011)
4. Barras, B., Bernardo, B.: The implicit calculus of constructions as a programming language with dependent types. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 365–379. Springer, Heidelberg (2008)
5. Böhm, C., Berarducci, A.: Automatic synthesis of typed lambda-programs on term algebras. TCS 39, 135–154 (1985)
6. Brady, E., Hammond, K.: Correct-by-construction concurrency: Using dependent types to verify implementations of effectful resource usage protocols. Fundam. Inform. 102(2), 145–176 (2010)
7. Coquand, T., Huet, G.: The calculus of constructions. Rapport de Recherche 530, INRIA, Rocquencourt, France (May 1986)
8. Crary, K., Weirich, S., Morrisett, G.: Intensional polymorphism in type-erasure semantics. In: ICFP 1998, pp. 301–312. ACM (1998)
9. Dagand, P.E., McBride, C.: Transporting functions across ornaments. In: ICFP 1998, ICFP 2012, pp. 103–114. ACM (2012)
10. Garrigue, J., Normand, J.L.: Adding GADTs to OCaml: the direct approach. In: ML 2011. ACM (2011)
11. Geuvers, H.: Induction is not derivable in second order dependent type theory. In: Abramsky, S. (ed.) TLCA 2001. LNCS, vol. 2044, pp. 166–181. Springer, Heidelberg (2001)
12. Giannini, P., Honsell, F., Rocca, S.R.D.: Type inference: Some results, some problems. Fundam. Inform. 19(1/2), 87–125 (1993)
13. 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)
14. McBride, C.: Homepage of the Strathclyde Haskell Enhancement (SHE) (2009), `http://personal.cis.strath.ac.uk/conor/pub/she/`
15. Miquel, A.: A model for impredicative type systems, universes, intersection types and subtyping. In: LICS, pp. 18–29. IEEE Computer Society (2000)
16. Miquel, A.: The implicit calculus of constructions. In: Abramsky, S. (ed.) TLCA 2001. LNCS, vol. 2044, pp. 344–359. Springer, Heidelberg (2001)
17. Mishra-Linger, N., Sheard, T.: Erasure and polymorphism in pure type systems. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 350–364. Springer, Heidelberg (2008)
18. Sheard, T., Pašalić, E.: Meta-programming with built-in type equality. In: LFM 2004, pp. 106–124 (2004)
19. Yorgey, B.A., Weirich, S., Cretin, J., Jones, S.L.P., Vytiniotis, D., Magalhães, J.P.: Giving Haskell a promotion. In: TLDI, pp. 53–66. ACM (2012)