The Nax programming language (work in progress)

Ki Yung Ahn[1], Tim Sheard[1], and Author2 SName2[2]

[1] Portland State University, Portland, Oregon, USA *
kya@cs.pdx.edu sheard@cs.pdx.edu
[2] University of Cambridge, Cambridge, UK
todo@cl.cam.ac.uk

1 Introduction

During the past decade, the functional programming community gained partial success in programming with fine-grained properties by moderate extension to the type system [Cheney and Hinze(2003),Cheney and Hinze(2002),Xi et al.(2003)Xi, Chen, and Chen]. This approach is often coined as being *"lightweight"* (e.g., lightweight dependent types[3], lightweight program verification), since using proof assistants for lightweight task is likely to require much more effort (heavyweight).

The Generalized Algebraic Data Type (GADT) extension implemented in the Glasgow Haskell Compiler (GHC) has popularized this approach even to everyday functional programming tasks. Unfortunately, implementations supporting the lightweight approach (e.g., GHC) lack **correctness guarantee** and **type inference** in general. In addition, practical implementations often lack support for term indices, so, term indices are faked (or, simulated) by types structure replicating term structure. We believe that the lightweight approach can become more productive and reliable if we can resolve these problems.

Problem 1. ***Correctness guarantee***

Proof assistants based on dependent types can *express fine-grained properties* using dependent types and also *guarantee correctness* since the calculi they are based on are strongly normalizing and logically consistent. For instance, Coq is based on the Calculus of Inductive Constructions, which is a dependently-typed λ-calculi known to be strongly normalizing and logically consistent.

But, fine-grained properties are not expressible in functional programming languages based on polymorphic type system, due to the lack of dependent types. Even if those fine-grained properties were somehow expressible, we cannot have guarantee of correctness on those properties. Recall that general purpose programming languages are neither strongly normalizing nor logically consistent, because they are capable of expressing diverging computations by design. So, the lightweight approach in conventional functional languages can only raise some confidence on correctness (assuming that inconsistent fragment of the type system was never used for reasoning about desired properties) but cannot guarantee correctness of the desired properties as in proof assistants.

* supported by NSF grant 0910500.

[3] http://okmij.org/ftp/Computation/lightweight-dependent-typing.html

Problem 2. ***Type inference***

Type inference makes type-safe programming pleasant for everyday programming tasks since programmers are free from tedious type annotations. Many typed functional programming languages including Haskell 98 and SML are based on the Hindley-Milner type system (HM), which is not only *type-safe* but also *type-inferable*.

An essential feature for the lightweight approach is *indexed datatypes*, which are datatypes with *heterogeneous type parameters* (a.k.a. *indices*). Such datatypes used in the lightweight approach are beyond the polymorphic type schemes in HM. Inclusion of rather simple subset of indexed datatypes, such as nested datatypes [Bird and Meertens(1998)], already makes type inference undecidable [Henglein(1993)].

So, functional language implementations supporting lightweight approach require type annotations on both indexed datatype declarations and function definitions involving indexed datatypes, in order to recover decidability of type inference.[4] Type annotations on datatype declarations are absolutely necessary either when the result types of data constructors have heterogeneous indices or when there the argument types of data constructors have existential indices. However, it still remains to be answered *where and how much type annotations are needed on function definitions*.

Problem 3. ***Faked term indices***

The indexed datatypes can only have static dependencies (i.e., indices must be static), unlike full-fledged dependent types in proof assistants, which can depend on both static and dynamic objects. Therefore, having term indices does not imply full-fledged dependent types,

Indexed datatypes can be indexed by either types or terms, or both. Type representations [Cheney and Hinze(2003)] (Fig. 1) used in datatype generic programming is a typical example of a type-indexed datatype. The length-index list, or `Vector`, (Fig. 2) should be a typical example of a term-indexed datatype. However, the `Vector` datatype declaration in Fig. 2 uses faked term indices, which are empty types `Succ` and `Zero` simulating the data constructors of `data Nat = Succ Nat | Zero` at term level. Such faked term indices can be problematic since they duplicate code (i.e., operations on `Nat` should be redefined at type level) and have less precise semantics than true term indices (e.g., cannot prevent `Succ Bool`).

Although there has been studies of indexed datatypes with true term indices [Zenger(1997)], including term indices in practical functional languages is not trivial. Allowing arbitrary terms at type level breaks the decidability of type checking due to diverging terms. Term indices in types implies that type equality depends on term equality. And, obviously, term equality will loop when one of the terms being compared diverges. Undecidability of type checking can be lifted once we have resolved *Problem 1*, and make sure that indices are normalizing.

[4] Type inference aided by type annotation is also called partial type inference or type reconstruction.

```
data Rep t where
  R_Int  :: Rep Int
  R_Char :: Rep Char
  R_List :: Rep a -> Rep [a]
  R_Pair :: Rep a -> Rep b -> Rep (a,b)
```

Fig. 1. A type representation for `Int`, `Char`, `[]`, and `(,)` in Haskell with GADTs

```
data Succ n
data Zero

data Vector a n where
  VCons :: a -> Vector m a -> Vector a (Succ m)
  VNil  :: Zero
```

Fig. 2. Length indexed list datatype `Vector` in Haskell with GADTs

To resolve these problems, we have designed and implemented a prototype of the Nax programming language. The current proptotpye of Nax is a strongly normalizing functional language supporting the following features (which are all illustrated in §2):

Two level datatypes. Recursive dataypes are introduced in two stages. First a non-recursive *structure* is introduced which abstracts over where recursive sub-components will appear. Then a *fix-point* is taken to define the recursive types (§2.1). To minimize the extra notation necessary to program in this manner an extensive *macro-facility* is provided. The most common macro forms can be automatically derived. This is illustrated in §2.3.

Indexed types with static term indices. A type constructor is applied to arguments. Arguments are either parameters or indices. A datatype is polymorphic over its parameters (in the sense that parametricity theorems hold over parameters). Parameters are always types. Indexed arguments can be either types or terms. An index usually encodes a static property about the shape or form of a value with that type. We use different kinding rules to separate term indices from type indices. For instance a length indexed list, `x`, might have type (`List Int 2`). The `Int` is a parameter, indicating the list contains integers, but the `2` is an index indicating that the list, `x`, has exactly two elements. Types are static in Nax. Types are only used for type checking and are computationally irrelevant, even though some parts of a type might include terms. In other words, Nax supports *index erasure*,

Recursive types of unrestricted polarity but restricted elimination. It is well known that unrestricted recursive types enable diverging computation even without any recursion at the term level. To design a normalizing language that supports recursive types, we must make a design choice that

limits the use of recursive types. There are two possible design choices. We may restrict the formation of recursive types (i.e., type definition) or we may restrict the elimination of recursive types (i.e., pattern matching). In Nax, we make the latter design choice, so that we can define *all the recursive datatypes* available in modern functional languages.

Mendler style iteration and recursion combinators. Any useful normalizing language should support principled recursion operators that guarantee normalization. Such operators should be easy to use, and expressive over datatypes with both parameters and indices. Mendler style combinators meet both requirements. So, we adopt them in Nax.

Type inference (reconstruction) from minimal annotation. When we extend the Hindley-Milner type system with indexed data types, we no longer have type inference for completely unannotated terms. For example, this restriction shows up in languages which support GADTs, which support a kind of type indexing. Although complete type inference is not possible, partial type inference (reconstruction of missing type information) is still possible when sufficient type annotations are provided. Nax's systematic partition of type parameters from type indices provides a mechanism where it is possible to decide exactly where additional type annotations are needed, and to enforce that the programmer supply such annotations. This system faithfully extends the Hindley-Milner type inference (i.e., no additional annotations are needed for the programs that are already inferable by Hindley-Milner).

2 Nax by Example

We introduce programming in our implementation of Nax by providing examples. An example usually consists of several parts.

- Introducing data definitions to describe the data of interest. Recursive data is introduced in two stages. We must be careful to separate parameters from indices when using indices to describe static properties of data.
- Introduce macros, either by explicit definition, or by automatic fixpoint derivation to limit the amount of explicit notation that must be supplied by the programmer.
- Write a series of definitions that describe how the data is to be manipulated. Deconstruction of recursive data can only be performed with Mendler-style combinators to ensure strong normalization.

2.1 Two-level types

Non recursive datatypes are introduced by the **data** declaration. The data declaration can include arguments. The kind and separation of arguments into parameters and a indices is inferred. For example, the three non-recursive data types,

Bool, *Either*, and *Maybe*, familiar to many functional programmers, are introduced by declaring the kind of the type, and the type of each of the constructors. This is similar to the way GADTs are introduced in Haskell.

data $Bool : *$	**data** $Either : * \to * \to *$	**data** $Maybe : * \to *$
where	**where**	**where**
$False : Bool$	$Left : a \to Either\ a\ b$	$Nothing : Maybe\ a$
$True : Bool$	$Right : b \to Either\ a\ b$	$Just : a \to Maybe\ a$

Note the kind information ($Bool : *$) declares *Bool* to be a type, ($Either : * \to * \to *$) declares *Either* to be a type constructor with two type arguments, and ($Maybe : * \to *$) declares *Maybe* to be a type constructor with one type argument.

To introduce a recursive type, we first introduce a non recursive datatype that uses a parameter where the usual recursive components occur. By design, normal parameters of the introduced type are written first (a in L below) and the type argument to stand for the recursive component is written last (the r of N, and the r of L below).

-- The fixpoint of N will	-- The fixpoint of $(L\ a)$ will
-- be the natural numbers.	-- be the polymorphic lists
data $N : * \to *$ **where**	**data** $L : * \to * \to *$ **where**
$Zero : N\ r$	$Nil\ \ : L\ a\ r$
$Succ : r \to N\ r$	$Cons : a \to r \to L\ a\ r$

A recursive type can be defined as the fixpoint of a (perhaps partially applied) non-recursive type constructor. Thus the traditional natural numbers are typed by $\mu^{[*]}\ N$ and the traditional lists with components of type a are typed by $\mu^{[*]}\ (L\ a)$. Note that the recursive type operator $\mu^{[\kappa]}$ is itself specialized with a kind argument inside square brackets ($[\kappa]$). The recursive type $(\mu^{[\kappa]}\ f)$ is well kinded only if the operand f has kind $\kappa \to \kappa$, in which case the recursive type $(\mu^{[k]}\ f)$ has kind κ. Since both N and $(L\ a)$ have kind $* \to *$, the recursive types $\mu^{[*]}\ N$ and $\mu^{[*]}\ (L\ a)$ have kind $*$. That is, they are both types, not type constructors.

2.2 Creating values

Values of a particular data type are created by use of constructor functions. For example *True* and *False* are nullary constructors (or, constants) of type *Bool*. (*Left* 4) is a value of type (*Either Int a*). Values of recursive types (i.e., those values with types such as $(\mu^{[k]}\ f)$ are formed by using the special $\mathsf{In}^{[\kappa]}$ constructor expression. Thus *Nil* has type $L\ a$ and $(\mathsf{In}^{[*]}\ Nil)$ has type $(\mu^{[*]}\ (L\ a))$. In general, applying the operator $\mathsf{In}^{[k]}$ injects a term of type $f\ (\mu^{[k]}\ f)$ to the recursive type $(\mu^{[k]}\ f)$. Thus a list of *Bool* could be created using the term $(\mathsf{In}^{[*]}\ (Cons\ True\ (\mathsf{In}^{[*]}\ (Cons\ False\ (\mathsf{In}^{[*]}\ Nil)))))$. A general rule of thumb, is to apply $\mathsf{In}^{[k]}$ to terms of non-recursive type to get terms of recursive type. Writing programs using two level types, and recursive injections has definite benefits,

but it surely makes programs rather annoying to write. Thus, we have provided Nax with a simple but powerful synonym (macro) facility.

2.3 Synonyms, constructor functions, and fixpoint derivation

We may codify that some type is the fixpoint of another, once and for all, by introducing a type synonym (macro).

synonym $Nat = \mu^{[*]}\ N$

synonym $List\ a = \mu^{[*]}\ (L\ a)$

In a similar manner we can introduce constructor functions that create recursive values without explicit mention of $\mathsf{In}^{\cdot}$ at their call sites (potentially many), but only at their site of definition (exactly once).

$zero \quad = \mathsf{In}^{[*]}\ Zero$

$succ\ n = \mathsf{In}^{[*]}\ (Succ\ n)$

$nil \quad = \mathsf{In}^{[*]}\ Nil$

$cons\ x\ xs = \mathsf{In}^{[*]}\ (Cons\ x\ xs)$

This is such a common occurrence that recursive synonyms and recursive constructor functions can be automatically derived. Automatic synonym and constructor derivation using Nax is both concise and simple. The clause "**deriving fixpoint** *List*" (below right) causes the **synonym** for *List* to be automatically defined. It also defines the constructor functions *nil* and *cons*. By convention, the constructor functions are named by dropping the initial upper-case letter in the name of the non-recursive constructors to lower-case. To illustrate, we provide side-by-side comparisons of Haskell and two different uses of Nax.

Haskell	*Nax with synonyms*	*Nax with derivation*
data $List\ a$	**data** $L : * \to * \to *$ **where**	**data** $L : * \to * \to *$ **where**
$= Nil$	$Nil : L\ a\ r$	$Nil : L\ a\ r$
$\mid\ Cons\ a\ (List\ a)$	$Cons : a \to r \to L\ a\ r$	$Cons : a \to r \to L\ a\ r$
	synonym $List\ a = \mu^{[*]}\ (L\ a)$	**deriving fixpoint** *List*
	$nil \quad = \mathsf{In}^{[*]}\ Nil$	
	$cons\ x\ xs = \mathsf{In}^{[*]}\ (Cons\ x\ xs)$	
$x = Cons\ 3\ (Cons\ 2\ Nil)$	$x = cons\ 3\ (cons\ 2\ nil)$	$x = cons\ 3\ (cons\ 2\ nil)$

2.4 Mendler combinators for non-indexed types

There are no restrictions on what kind of datatypes can be defined in Nax. There are also no restrictions on the creation of values. Values are created using constructor functions, and the recursive injection ($\mathsf{In}^{[k]}$). To ensure strong normalization, analysis of constructed values has some restrictions. Values of non-recursive types can be freely analysed using pattern matching. Values of recursive types must be analysed using one of the Mendler-style combinators. By design, we limit pattern matching to values of non-recursive types, by *not* providing any mechanism to match against the recursive injection ($\mathsf{In}^{[k]}$).

To illustrate simple pattern matching over non-recursive types, we give a Nax multi-clause definition for the $\neg$ function over the (non-recursive) *Bool* type, and a function that strips off the *Just* constructor over the (non-recursive) *Maybe* type using a case expression.

$$\begin{array}{l|l} \neg\ \mathit{True} = \mathit{False} & \mathit{unJust0}\ x = \mathbf{case}^{\{\}}\ x\ \mathbf{of}\ \mathit{Just}\ x \to x \\ \neg\ \mathit{False} = \mathit{True} & \qquad\qquad\qquad\quad \mathit{Nothing} \to 0 \end{array}$$

Analysis of recursive data is performed with Mendler-style combinators. In our implementation we provide 5 Mendler-style combinators: $\mathsf{MIt}^{\cdot}$ (fold or catamorphism or iteration), $\mathsf{MPr}^{\cdot}$ (primitive recursion), $\mathsf{McvIt}^{\cdot}$ (courses of values iteration), and $\mathsf{McvPr}^{\cdot}$ (courses of values primitive recursion), and $\mathsf{MsfIt}^{\cdot}$ (fold or catamorphism or iteration for recursive types with negative occurrences).

A Mendler-style combinator is written in a manner similar to a case expression. A Mendler-style combinator expression contains patterns, and the variables bound in the patterns are scoped over a term. This term is executed if the pattern matches. A mendler-style combinator expression differs from a case expression in that it also introduces additional names (or variables) into scope. These variables play a role similar in nature to the operations of an abstract datatype, and provide additional functionality in addition to what can be expressed using just case analysis.

For a visual example, compare the **case** expression to the $\mathsf{MIt}^{\cdot}$ expression. In the **case**, each *clause* following the **of** indicates a possible match of the scrutinee x. In the $\mathsf{MIt}^{\cdot}$, each *equation* following the **with**, binds the variable f, and matches the pattern to a value related to the scrutinee x.

$$\begin{array}{l|l} \mathbf{case}^{\{\}}\ x\ \mathbf{of}\ \mathit{Nil} \to e_1 & \mathsf{MIt}^{\{\}}\ x\ \mathbf{with}\ f\ (\mathit{Cons}\ x\ xs) = e_1 \\ \qquad\qquad\quad \mathit{Cons}\ x\ xs \to e_2 & \qquad\qquad\qquad\quad f\ \mathit{Nil} = e_2 \end{array}$$

The number and type of the additional variables depends upon which family of Mendler combinators is used to analyze the scrutinee. Each equation specifies (a potential) computation in an abstract datatype depending on whether the pattern matches. For the $\mathsf{MIt}^{\cdot}$ combinator (above) the abstract datatype has the following form. The scrutinee, x is a value of some recursive type $(\mu^{[*]}\ T)$ for a non-recursive type constructor T. In each clause, the pattern has type $(T\ r)$, for some abstract type r. The additional variable introduced (f) is an operator over the abstract type, r, that can safely manipulate only abstract values of type r.

Different Mendler-style combinators are implemented by different abstract types. Each abstraction safely describes a class of provably terminating computations over a recursive type. The number (and type) of abstract operations differs from one family of Mendler combinators to another. We give descriptions of three families of Mendler combinators, their abstractions, and the types of

the operators within the abstraction, below. In each description, the type *ans* represents the result type, when the Mendler combinator is fully applied.

$\mathsf{MIt}^{\{\,\}}\ x$ **with**
$\quad f\ \ p_i = e_i$

$x\ : \mu^{[*]}\ T$
$f\ : r \to ans$

$p_i : T\ r$
$e_i : ans$

$\mathsf{MIt}^{\{\psi\}}\ \varphi\ (\mathsf{In}^{[*]}\ x)$
$\quad = \varphi\ (\mathsf{MIt}^{\{\psi\}}\ \varphi)\ x$

$\mathsf{MPr}^{\{\,\}}\ x$ **with**
$\quad f\ \ cast\ \ p_i = e_i$

$x\ : \mu^{[*]}\ T$
$f\ : r \to ans$
$cast : r \to \mu^{[*]}\ T$
$p_i : T\ r$
$e_i : ans$

$\mathsf{MPr}^{\{\psi\}}\ \varphi\ (\mathsf{In}^{[*]}\ x)$
$\quad = \varphi\ (\mathsf{MPr}^{\{\psi\}}\ \varphi)\ (\mathsf{In}^{[*]})\ x$

$\mathsf{McvIt}^{\{\,\}}\ x$ **with**
$\quad f\ \ project\ \ p_i = e_i$

$x\ : \mu^{[*]}\ T$
$f\ : r \to ans$
$project : r \to T\ r$
$p_i\ : T\ r$
$e_i\ : ans$

$\mathsf{McvIt}^{\{\psi\}}\ \varphi\ (\mathsf{In}^{[*]}\ x)$
$\quad = \varphi\ (\mathsf{McvIt}^{\{\psi\}}\ \varphi)\ out\ x$
$\quad$ **where** $out\ (\mathsf{In}^{[*]}\ x) = x$

A Mendler-style combinator implements a (provably terminating) recursive function applied to the scrutinee. The abstract type and its operations ensure termination. Note that every operation above includes an abstract operator, $f : r \to ans$. This operation represents a recursive call in the function defined by the Mendler-style combinator. Other operations, such as *cast* and *project*, support additional functionality within the abstraction in which they are defined ($\mathsf{MPr}^{\cdot}$ and $\mathsf{McvIt}^{\cdot}$ respectively). The equations at the bottom of each section provide an operational understanding of how the operator works. These can be safely ignored until after we see some examples of how a Mendler-style combinator works in practice.

$$
\begin{array}{lll}
length\ y & = \mathsf{MIt}^{\{\,\}}\ y & \textbf{with}\ \ len\ Nil = zero \\
 & & \quad\ \ len\ (Cons\ x\ xs) = (succ\ zero) + len\ xs \\
tail\ x & = \mathsf{MPr}^{\{\,\}}\ x & \textbf{with}\ \ tl\ cast\ Nil = nil \\
 & & \quad\ \ tl\ cast\ (Cons\ y\ ys) = cast\ ys \\
factorial\ x & = \mathsf{MPr}^{\{\,\}}\ x & \textbf{with}\ \ fact\ cast\ Zero = succ\ zero \\
 & & \quad\ \ fact\ cast\ (Succ\ n) = times\ (succ\ (cast\ n))\ (fact\ n) \\
fibonacci\ x & = \mathsf{McvIt}^{\{\,\}}\ x & \textbf{with}\ \ fib\ out\ Zero = succ\ zero \\
 & & \quad\ \ fib\ out\ (Succ\ n) = \textbf{case}^{\{\,\}}\ (out\ n)\ \textbf{of} \\
 & & \qquad\qquad Zero \to succ\ zero \\
 & & \qquad\qquad Succ\ m \to fib\ n + fib\ m
\end{array}
$$

The *length* function uses the simplest kind of recursion where each recursive call is an application to a direct subcomponent of the input. Operationally, *length* works as follows. The scrutinee, y, has type $(\mu^{[*]}\ (L\ a))$, and has the form $(\mathsf{In}^{[*]}\ x)$. The type of y implies that x must have the form *Nil* or $(Cons\ x\ xs)$. The $\mathsf{MIt}^{\cdot}$ strips off the $\mathsf{In}^{[*]}$ and matches x against the *Nil* and $(Cons\ x\ xs)$ patterns. If the *Nil* pattern matches, then 0 is returned. If the $(Cons\ x\ xs)$ pattern matches, x and xs are bound. The abstract type mechanism gives the

pattern ($Cons\ x\ xs$) the type ($L\ a\ r$), so ($x : a$) and ($xs : r$) for some abstract type r. The abstract operation, ($len : r \to Int$), can safely be applied to xs, obtaining the length of the tail of the original list. This value is incremented, and then returned. The MIt· abstraction provides a safe way to allow the user to make recursive calls, *len*, but the abstract type, r, limits its use to direct subcomponents, so termination is guaranteed.

Some recursive functions need direct access, not only to the direct subcomponents, but also the original input as well. The Mendler-style combinator MPr· provides a safe, yet abstract mechanism, to support this. The Mendler MPr· abstraction provides two abstract operations. The recursive caller with type ($r \to ans$) and a casting function with type ($r \to \mu^{[k]}\ T$). The casting operation allows the user to recover the original type from the abstract type r, but since the recursive caller only works on the abstract type r, the user cannot make a recursive call on one of these cast values. The functions *factorial* (over the natural numbers) and *tail* (over lists) are both defined using MPr·.

Note how in *factorial* the original input is recovered (in constant time) by taking the successor of casting the abstract predecessor, n. In the *tail* function, the abstract tail, *ys*, is cast to get the answer, and the recursive caller is not even used.

Some recursive functions need direct access, not only to the direct subcomponents, but even deeper subcomponents. The Mendler-style combinator McvIt· provides a safe, yet abstract mechanism, to support this. The function *fibonacci* is a classic example of this kind of recursion. The Mendler McvIt· provides two abstract operations. The recursive caller with type ($r \to ans$) and a projection function with type ($r \to T\ r$). The projection allows the programmer to observe the hidden T structure inside a value of the abstract type r. In the *fibonacci* function above, we name the projection *out*. It is used to observe if the abstract predecessor, n, of the input, x, is either zero, or the successor of the second predecessor, m, of x. Note how recursive calls are made on the direct predecessor, n, and the second predecessor, m.

Each recursion combinator can be defined by the equation at the bottom of its figure. Each combinator can be given a naive type involving the concrete recursive type ($\mu^{[*]}\ T$), but if we instead give it a more abstract type, abstracting values of type ($\mu^{[*]}\ T$) into some unknown abstract type r, one can safely guarantee a certain pattern of use that insures termination. Informally, if the combinator works for some unknown type r it will certainly also work for the actual type ($\mu^{[*]}\ T$), but because it cannot assume that r has any particular structure, the user is forced to use the abstract operations in carefully proscribed ways.

2.5 Types with static indices

Recall that a type can have both parameters and indices, and that indices can be either types or terms. We define three types below each with one or more indices. Each example defines a non-recursive type, and then uses derivation to define synonyms for its fix point and recursive constructor functions. By convention, in

each example, the argument that abstracts the recursive components is called r. By design, arguments appearing before r are understood to be parameters, and arguments appearing after r are understood to be indices. To define a recursive type with indices, it is necessary to give the argument, r, a higher-order kind. That is, r should take indices as well, since it abstracts over a recursive type which takes indices.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Nest} : (* \to *) \to * \to * \to * \ \textbf{where} \\
\quad \mathit{Tip} \quad : a \to \mathit{Nest}\ r\ a \\
\quad \mathit{Fork} \ : r\ (a, a) \to \mathit{Nest}\ r\ a \\
\qquad \textbf{deriving fixpoint}\ \mathit{PowerTree} \\
\textbf{data}\ V : * \to (\mathit{Nat} \to *) \to \mathit{Nat} \to * \ \textbf{where} \\
\quad \mathit{Vnil} \quad : V\ a\ r\ \{\text{`}\mathit{zero}\} \\
\quad \mathit{Vcons} : a \to r\ \{n\} \to V\ a\ r\ \{\text{`}\mathit{succ}\ n\} \\
\qquad \textbf{deriving fixpoint}\ \mathit{Vector} \\
\textbf{data}\ \mathit{Tag} = E \mid O \\
\textbf{data}\ P : (\mathit{Tag} \to \mathit{Nat} \to *) \to \mathit{Tag} \to \mathit{Nat} \to * \ \textbf{where} \\
\quad \mathit{Base} \quad : P\ r\ \{E\}\ \{\text{`}\mathit{zero}\} \\
\quad \mathit{StepO} : r\ \{O\}\ \{i\} \to P\ r\ \{E\}\ \{\text{`}\mathit{succ}\ i\} \\
\quad \mathit{StepE} : r\ \{E\}\ \{i\} \to P\ r\ \{O\}\ \{\text{`}\mathit{succ}\ i\} \\
\qquad \textbf{deriving fixpoint}\ \mathit{Proof}
\end{array}
$$

Note, to distinguish type indices from term indices (and to make parsing unambiguous), we enclose term indices in braces ($\{...\}$). We also backquote (`) variables in terms that we expect to be bound in the current environment. Unbackquoted variables are taken to be universally quantified. By backquoting *succ*, we indicate that we want terms, which are applications of the successor function, but not some universally quantified function variable[5]. For non-recursive types without parameters, the kind of the fixpoint is the same as the kind of the recursive argument r. If the non-recursive type has parameters, the kind of the fixpoint will be composed of the parameters $\to$ the kind of the recursive argument r. For example, study the kinds of the fixpoints for the non-recursive types declared above in the table below.

non-recursive type	*Nest*	*V*	*P*
recursive type	*PowerTree*	*Vector*	*Proof*
kind of T	$* \to *$	$* \to \mathit{Nat} \to *$	$\mathit{Tag} \to \mathit{Nat} \to *$
kind of r	$* \to *$	$\mathit{Nat} \to *$	$\mathit{Tag} \to \mathit{Nat} \to *$
number of parameters	0	1	0
number of indices	1 (type)	1 (term)	2 (term,term)

[5] In the design of Nax we had a choice. Either, explicitly declare each universally quantified variable, or explicitly mark those variables not universally quantified. Since quantification is much more common than referring to variables already in scope, the choice was easy.

Recall, indices are used to track static properties about values with those types. A well-formed ($PowerTree\ x$) contains a balanced set of parenthesized binary tuples of elements. The index, x, describes what kind of values are nested in the parentheses. The invariant is that the number of items nested is always an exact power of 2. A ($Vector\ a\ \{n\}$) is a list of elements of type a, with length exactly equal to n, and a ($Proof\ \{E\}\ \{n\}$) witnesses that the natural number n is even, and a ($Proof\ \{O\}\ \{m\}$) witnesses that the natural number m is odd. Some example value with these types are given below.

$$
\begin{array}{l}
tree1 : PowerTree\ Int = tip\ 3 \\
tree2 : PowerTree\ Int = fork\ (tip\ (2,5)) \\
tree3 : PowerTree\ Int = fork\ (fork\ (tip\ ((4,7),(0,2)))) \\
v2 : Vector\ Int\ \{succ\ (succ\ zero)\} = (vcons\ 3\ (vcons\ 5\ vnil)) \\
p1 : P\ \{O\}\ \{succ\ zero\} \qquad\quad = stepE\ base \\
p2 : P\ \{E\}\ \{succ\ (succ\ zero)\} = stepO\ (stepE\ base)
\end{array}
$$

Note that in the types of the terms above, the indices in braces ($\{...\}$) are ordinary terms (not types). In these example we use natural numbers (e.g., $succ\ (succ\ zero)$) and elements (E and O) of the two-valued type Tag. It is interesting to note that sometimes the terms are of recursive types (e.g., Nat which is a synonym for $\mu^{[*]}\ N$), and some are non-recursive types (e.g., Tag).

2.6 Mendler-style combinators for indexed types

Mendler-style combinators generalize naturally to indexed types. The key observation that makes this generalization possible is that the types of the operations within abstraction have to be generalized to deal with the type indices in a consistent manner. How this is done is best first explained by example, and then later abstracted to its full general form.

Recall, a value of type ($PowerTree\ Int$) is a set of integers. This set is constructed as a balanced binary tree with pairs at the leaves (see *tree2* and *tree3* above). The number of integers in the set is an exact power of 2. Consider a function that adds up all those integers. One wants a function of type ($PowerTree\ Int \to Int$). One strategy to writing this function is to write a more general function of type ($PowerTree\ a \to (a \to Int) \to Int$). In Nax, we can do this as follows:

$$
\begin{array}{l}
genericSum\ t = \mathsf{MIt}^{\{\,a.\,(a \to Int) \to Int\,\}}\ t\ \textbf{with} \\
\qquad\qquad\qquad sum\ (Tip\ x)\ \ = \lambda f \to f\ x \\
\qquad\qquad\qquad sum\ (Fork\ x) = \lambda f \to sum\ x\ (\lambda(a,b) \to f\ a + f\ b) \\
sumTree\ t = genericSum\ t\ (\lambda x \to x)
\end{array}
$$

In general, the type of the result of a function over an indexed type, can depend upon what the index is. Thus, a Mendler-style combinator over a value with an indexed type, must be type-specialized to that value's index. Different values of

the same general type, will have different indices. After all, the role of an index is to witness an invariant about the value, and different values might have different invariants. Capturing this variation is the role of the clause $\{\, a.\ (a \to \mathit{Int}) \to \mathit{Int}\}$ following the keyword $\mathsf{MIt}^{\cdot}$. We call such a clause, an *index transformer*. In the same way that the type of the result depends upon the index, the type of the different components of the abstract datatype implementing the Mendler-style combinator also depend upon the index. In fact, everything depends upon the index in a uniform way. The index transformer captures this uniformity. One cannot abstract over the index transformer in Nax. Each Mendler-style combinator, over an indexed type, must be supplied with a concrete clause (inside the braces) that describe how the results depend upon the index. To see how the transformer is used, study the types of the terms in the following paragraph. Can you see the relation between the types and the transformer?

The scrutinee t has type $(\mathit{PowerTree}\ a)$ which is a synonym for $((\mu^{[*\to *]}\ \mathit{Nest})\ a)$. The recursive caller *sum* has type $(\forall\ a\ .\ r\ a \to (a \to \mathit{Int}) \to \mathit{Int})$, for some abstract type constructor r. Recall r has an index, so it must be a type constructor, not a type. The patterns $(\mathit{Tip}\ x)$ and $(\mathit{Fork}\ x)$ have type $(\mathit{Nest}\ r\ a)$ and the right hand sides of the equations: $(\lambda f \to f\ x)$ and $(\lambda f \to \mathit{sum}\ x\ (\lambda(a,b) \to f\ a {+} f\ b))$, have type $((a \to \mathit{Int}) \to \mathit{Int})$. Note that the dependency of $((a \to \mathit{Int}) \to \mathit{Int})$ on the index a, appears in both the result type, and the type of the recursive caller. If we think of an index transformer, like $\{\, a\ .\ (a \to \mathit{Int}) \to \mathit{Int}\}$, as a function: $\psi\ a = (a \to \mathit{Int}) \to \mathit{Int}$, we can succinctly describe the types of the abstract operations in the $\mathsf{MIt}^{\cdot}$ Mendler abstraction. In the table below, we put the general case on the left, and terms from the *genericSum* example, that illustrate the general case, on the right.

$\mathsf{MIt}^{\{\psi\}}\ x$ **with**
$f\ p_i = e_i$

$\psi : \kappa \to *$	$\{\, a\ .\ (a \to \mathit{Int}) \to \mathit{Int}\}$	$: * \to *$
$T : (\kappa \to *) \to \kappa \to *$	Nest	$: (* \to *) \to * \to *$
$x : (\mu^{[\kappa \to *]}\ T)\ a$	t	$: (\mu^{[* \to *]}\ \mathit{Nest})\ a$
$f : \forall\ (a : \kappa)\ .\ r\ a \to \psi\ a$	sum	$: \forall\ (a : *)\ .\ r\ a \to (a \to \mathit{Int}) \to \mathit{Int}$
$p_i : T\ r\ a$	$\mathit{Fork}\ x$	$: \mathit{Nest}\ r\ a$
$e_i : \psi\ a$	$\lambda f \to f\ x$	$: (a \to \mathit{Int}) \to \mathit{Int}$

The same scheme for $\mathsf{MIt}^{\cdot}$ generalizes to type constructors with term indices, and with multiple indices. To illustrate this we give the generic schemes for type constructors with 2 or 3 indices. In the table the variables κ_1, κ_2, and κ_3, stand for arbitrary kinds (either kinds for types, like $*$, or kinds for terms, like *Nat* or *Tag*).

$$
\begin{array}{l|l}
T : (\kappa_1 \to \kappa_2 \to *) \to (\kappa_1 \to \kappa_2 \to *) & T : (\kappa_1 \to \kappa_2 \to \kappa_3 \to *) \to (\kappa_1 \to \kappa_2 \to \kappa_3 \to^*) \\
\psi \ : \kappa_1 \to \kappa_2 \to * & \psi \ : \kappa_1 \to \kappa_2 \to \kappa_3 \to * \\
x \ : (\mu^{[\kappa_1 \to \kappa_2 \to *]} \ T) \ a \ b & x \ : (\mu^{[\kappa_1 \to \kappa_2 \to \kappa_3 \to *]} \ T) \ a \ b \ c \\
f \ : \forall \ (a : \kappa_1) \ (b : \kappa_2) \ . \ r \ a \ b \to \psi \ a \ b & f \ : \forall \ (a : \kappa_1) \ (b : \kappa_2) \ (c : \kappa_3) \ . \ r \ a \ b \ c \to \psi \ a \ b \ c \\
p_i : T \ r \ a \ b & p_i : T \ r \ a \ b \ c \\
e_i : \psi \ a \ b & e_i : \psi \ a \ b \ c
\end{array}
$$

The simplest form of index transformation, is where the transformation is a constant function. This is the case of the function that computes the integer length of a natural-number, length-indexed, list (what we called a *Vector*). Independent of the length the result is an integer. Such a function has type: $\mathit{Vector}\ a\ \{n\} \to \mathit{Int}$. We can write this as follows:

$$
\begin{array}{l}
\mathit{vlen}\ x = \mathsf{MIt}^{\{\{i\}.\ \mathit{Int}\}}\ x\ \ \textbf{with}\ \ \mathit{len}\ \mathit{Vnil} \qquad\quad = 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{len}\ (\mathit{Vcons}\ x\ xs) = 1 + \mathit{len}\ xs
\end{array}
$$

Let's study an example with a more interesting index transformation. A term with type $(\mathit{Proof}\ \{E\}\ \{n\})$, which is synonymous with the type $(\mu^{[\mathit{Tag} \to \mathit{Nat} \to *]}\ P\ \{E\}\ \{n\})$, witnesses that the term n is even. Can we transform such a term into a proof that $n + 1$ is odd? We can generalize this by writing a function which has both of the types below:
$\mathit{Proof}\ \{E\}\ \{n\} \to \mathit{Proof}\ \{O\}\ \{`\mathit{succ}\ n\}$, and
$\mathit{Proof}\ \{O\}\ \{n\} \to \mathit{Proof}\ \{E\}\ \{`\mathit{succ}\ n\}$.
We can capture this dependency by defining the term-level function *flip*, and using an $\mathsf{MIt}^{\cdot}$ with the index transformer: $\{\{t\}\ \{i\}\ .\ \mathit{Proof}\ \{`\mathit{flip}\ t\}\ \{`\mathit{succ}\ i\}\}$.

$$
\begin{array}{l}
\mathit{flip}\ E = O \\
\mathit{flip}\ O = E \\
\mathit{flop}\ x = \mathsf{MIt}^{\{\{t\}\ \{i\}.\ \mathit{Proof}\ \{`\mathit{flip}\ t\}\ \{`\mathit{succ}\ i\}\}}\ x\ \ \textbf{with} \\
\qquad\quad f\ \mathit{Base} \qquad = \mathit{stepE}\ \mathit{base} \\
\qquad\quad f\ (\mathit{StepO}\ p) = \mathit{stepE}\ (f\ p) \\
\qquad\quad f\ (\mathit{StepE}\ p) = \mathit{stepO}\ (f\ p)
\end{array}
$$

For our last term-indexed example, every length-indexed list has a length, which is either even or odd. We can witness this fact by writing a function with type: $\mathit{Vector}\ a\ \{n\} \to \mathit{Either}\ (\mathit{Even}\ \{n\})\ (\mathit{Odd}\ \{n\})$. Here, *Even* and *Odd* are synonyms for particular kinds of *Proof*. To write this function, we need the index transformation: $\{\{n\}\ .\ \mathit{Either}\ (\mathit{Even}\ \{n\})\ (\mathit{Odd}\ \{n\})\}$.

$$
\begin{array}{l}
\textbf{synonym}\ \mathit{Even}\ \{x\} = \mathit{Proof}\ \{E\}\ \{x\} \\
\textbf{synonym}\ \mathit{Odd}\ \ \{x\} = \mathit{Proof}\ \{O\}\ \{x\} \\
\mathit{proveEvenOrOdd}\ x = \mathsf{MIt}^{\{\{n\}.\ \mathit{Either}\ (\mathit{Even}\ \{n\})\ (\mathit{Odd}\ \{n\})\}}\ x\ \ \textbf{with}
\end{array}
$$

$$
\begin{aligned}
prEOO\ Vnil &= Left\ base \\
prEOO\ (Vcons\ x\ xs) &= \mathbf{case}^{\{\}}\ prEOO\ xs\ \mathbf{of} \\
&\qquad Left\ p \ \rightarrow Right\ (stepE\ p) \\
&\qquad Right\ p \rightarrow Left\ (stepO\ p)
\end{aligned}
$$

2.7 Recursive types of unrestricted polarity but restricted elimination

In Nax, programmers can define recursive data structures with both positive and negative polarity. The classic example is a datatype encoding the syntax of λ-calculus, which uses higher-order abstract syntax (HOAS). Terms in the λ-calculus are either variables, applications, or abstractions. In a HOAS representation, one uses Nax functions to encode abstractions. We give a two level description for recursive λ-calculus *Term*s, by taking the fixpoint of the non-recursive *Lam* datatype.

$$
\begin{aligned}
&\mathbf{data}\ Lam : * \rightarrow * \ \mathbf{where} \\
&\quad App :: r \rightarrow r \rightarrow Lam\ r \\
&\quad Abs :: (r \rightarrow r) \rightarrow Lam\ r \\
&\qquad \mathbf{deriving\ fixpoint}\ Term \\
&apply = abs\ (\lambda f \rightarrow abs\ (\lambda x \rightarrow app\ f\ x))
\end{aligned}
$$

Note that we don't need to include a constructor for variables, as variables are represented by Nax variables, bound by Nax functions. For example the lambda term: $(\lambda f.\lambda x.f\ x)$ is encoded by the Nax term *apply* above.

Note also, the recursive constructor: $abs : (Term \rightarrow Term) \rightarrow Term$, introduced by the **deriving fixpoint** clause, has a negative occurrence of the type *Term*. In a language with unrestricted analysis, such a type could lead to non-terminating computations. The Mendler $\mathsf{MIt}^{\cdot}$ and $\mathsf{MPr}^{\cdot}$ combinators limit the analysis of such types in a manner that precludes non-terminating computations. The Mendler-style combinator, $\mathsf{Mcvlt}^{\cdot}$, is too expressive to exclude non-terminating computations, and must be restricted to recursive datatypes with no negative occurrences.

Even though $\mathsf{MIt}^{\cdot}$ and $\mathsf{MPr}^{\cdot}$ allow us to safely operate on values of type *Term*, they are not expressive enough to write many interesting functions. Fortunately, there is a more expressive Mendler-style combinator that is safe over recursive types with negative occurrences. We call this combinator $\mathsf{Msflt}^{\cdot}$. This combinator is based upon an interesting programming trick, first described by Sheard and Fegaras [Fegaras and Sheard(1996)], hence the "sf" in the name $\mathsf{Msflt}^{\cdot}$. The abstraction supported by $\mathsf{Msflt}^{\cdot}$ is as follows:

$$
\begin{array}{l|l}
\mathsf{Msflt}^{\{\}}\ x\ \mathbf{with} & x \ : \mu^{[*]}\ T \\
\quad f\ inv\ p_i = e_i & f \ : r \rightarrow ans \\
& inv : ans \rightarrow r \\
& p_i \ : T\ r \\
& e_i \ : ans
\end{array}
$$

To use $\mathsf{Msflt}^{\cdot}$ the inverse allows one to cast an answer into an abstract value. To see how this works, study the function that turns a *Term* into a string. The strategy is to write an auxiliary function, *showHelp* that takes an extra integer argument. Every time we encounter a lambda abstraction, we create a new variable, `x`n (see the function *new*), where n is the current value of the integer variable. When we make a recursive call, we increment the integer. In the comments (the rest of a line after --), we give the type of a few terms, including the abstract operations *sh* and *inv*.

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\quad \text{-- } cat : List\ String \to String \\
\qquad\qquad\qquad\qquad\quad \text{-- } new : Int \to String \\
new\ n = cat\ [\texttt{"x"}, show\ n] \\
\qquad\qquad\qquad\qquad\quad \text{-- } showHelp : Term \to (Int \to String) \\
\qquad\qquad\qquad\qquad\quad \text{-- } sh : r \to (Int \to String) \\
\qquad\qquad\qquad\qquad\quad \text{-- } inv : (Int \to String) \to r \\
\qquad\qquad\qquad\qquad\quad \text{-- } (\lambda n \to new\ m) : Int \to String \\
showHelp\ x = \\
\quad \mathsf{Msflt}^{\{\}}\ x\ \textbf{with} \\
\qquad sh\ inv\ (App\ x\ y) = \lambda m \to cat\ [\texttt{"("}, sh\ x\ m, \texttt{" "}, sh\ y\ m, \texttt{")"}] \\
\qquad sh\ inv\ (Abs\ f) \quad = \lambda m \to cat\ [\texttt{"(fn "}, new\ m, \texttt{" => "}, \\
\qquad\qquad\qquad\qquad\qquad\qquad sh\ (f\ (inv\ (\lambda n \to new\ m)))\ (m+1), \texttt{")"}] \\
showTerm\ x = showHelp\ x\ 0 \\
showTerm\ apply : List\ Char = \texttt{"(fn x0 => (fn x1 => (x0 x1)))"}
\end{array}
$$

The final line of the example above illustrates applying *showTerm* to *apply*. Recall that $apply = abs\ (\lambda f \to abs\ (\lambda x \to app\ f\ x))$, which is the HOAS representation of the λ-calculus term $(\lambda f.\lambda x.f\ x)$.

2.8 Lessons from Nax

Nax is our first attempt to build a strongly normalizing, sound and consistent logic, based upon Mendler-style iteration. We would like to emphasize the lessons we learned along the way.

- Writing types as the fixed point of a non-recursive type constructor is quite expressive. It supports a wide variety of types including the regular types (*Nat* and *List*), nested types (*PowerTree*), GADTs (*Vector*), and mutually recursive types (*Even* and *Odd*).
- Two-level types, while expressive, are a pain to program with (all those $\mu^{[\kappa]}$ and $\mathsf{In}^{[\kappa]}$ annotations), so a strong synonym or macro facility is necessary. With syntactic support, one hardly even notices.
- The use of term-indexed types allows programmers to write types that act as logical relations, and form the basis for reasoning about programs. Formalizing this is a large part of a sequel to this paper.

– Using Mendler-style combinators is expressive, and with syntactic support (the **with** equations of the Mendler combinators), is easy to use. In fact Nax programs are often no more complicated than their Haskell counterparts.
– Type inference is an important feature of a programming language. We hope you noticed, that apart from index transformers, no type information is supplied in any of the Nax examples. The Nax compiler reconstructs all type information.
– Index transformers are the minimal information needed to extend Hindley-Milner type inference over GADTs. One can always predict where they are needed, and the compiler can enforce that the programmer supplies them. They are never needed for non-indexed types. Nax faithfully extends Hindley-Milner type inference.

In the future we want Nax programs to include both a logical fragment and a non-logical (or programatic) fragment, and we want the type system to separate the two. Our approach to formalizing the logical Nax, is to embed each logical feature of Nax into a lower level language known to be strongly normalizing and logically consistent. Our approach of distinguishing type and term indices is unique, and requires the extension of some previous work on normalizing calculi. It is the subject of the sequel. We have already wrote a paper on System F_i, which is an extension of Fw Mendler-style iteration

3 Nax

3.1 Language definition

The Nax language definition is described in Fig. 3 and Fig. 4. Fig. 3 illustrates syntax, reduction rules, and well-formedness conditions for typing contexts. Fig. 4 illustrates sorting, kinding, and typing rules.

Typing contexts The typing context of Nax is separated into three zones. In addition to the two zones of F_i(the type level context Δ and the term level context Γ), we have top level contexts (Σ). The top level contexts can contain three kinds of bindings: type constructor bindings ($T : \kappa$), data constructor bindings ($C : \sigma$), and top level variable bindings ($`x : \sigma$). These bindings are introduced from declarations ($D$). Type constructor bindings ($T : \kappa$) and data constructor bindings ($C : \sigma$) are introduced from datatype declarations (**data** $T : \ldots$ **where** $\ldots$). Top level variable bindings ($`x : \sigma$) are introduced from top level definitions ($`x = t$). The rules for well-formed contexts in Nax are similar to those rules in F_i.

Kinds and their sorting rules The kind syntax of Nax is exactly the same as the kind syntax of F_i. The sorting rules are the same as F_i except we judge the sorts of kinds under the top level context (Σ).

Syntax:

$$
\begin{array}{lrcl}
\text{Kinds} & \kappa & ::= & * \mid \kappa \to \kappa \mid A \to \kappa \\
\text{Type Constructors} & F, G, A, B & ::= & X \mid T \mid \mu^{\kappa}(T\,\overline{\tau}) \mid F\ G \mid F\ \{s\} \mid A \to B \\
\text{Type Schemes} & \sigma & ::= & A \mid \forall X.\sigma \mid \forall i.\sigma \\
\text{Terms} & r, s, t & ::= & x \mid `x \mid i \mid \lambda x.t \mid r\ s \mid \mathbf{let}\ x = s\ \mathbf{in}\ t \mid \varphi^{\psi} \mid \mathsf{Mlt}\ x.\varphi^{\psi} \mid \mathsf{In}^{\kappa} \\
\text{Program} & \mathit{Prog} & ::= & \overline{D}; t \\
\text{Declarations} & D & ::= & \mathbf{data}\ T : \overline{K} \to * \ \mathbf{where}\ \overline{C : \overline{A} \to T\,\overline{\tau}} \quad \mid\ `x = t \\
\text{List of Declarations} & \overline{D} & ::= & \cdot \mid D, \overline{D} \\
\text{Kind Arguments} & K & ::= & \kappa \mid A \\
\text{Type Arguments} & \tau & ::= & G \mid \{s\} \\
\text{Type Argument Variables} & \iota & ::= & X \mid i \\
\text{Index Transformers} & \psi & ::= & \cdot \mid \bar{\iota}.B \\
\text{Case Branches} & \varphi & ::= & \overline{C\,\overline{x} \to t} \\
\text{Contexts} & \Sigma & ::= & \cdot \mid \Sigma, T : \kappa \mid \Sigma, C : \sigma \mid \Sigma, `x : \sigma \\
 & \Delta & ::= & \cdot \mid \Delta, X^{\kappa} \mid \Delta, i^{\sigma} \\
 & \Gamma & ::= & \cdot \mid \Gamma, x : \sigma
\end{array}
$$

Well-formed contexts:

$$\boxed{\vdash \Sigma} \qquad \frac{}{\vdash \cdot} \qquad \frac{\vdash \Sigma \quad \Sigma \vdash \kappa : \Box}{\vdash \Sigma, T : \kappa}\big(T \notin \mathsf{dom}(\Sigma)\big)$$

$$\frac{\vdash \Sigma \quad \Sigma; \cdot \vdash \sigma : *}{\vdash \Sigma, C : \sigma}\big(C \notin \mathsf{dom}(\Sigma)\big) \qquad \frac{\vdash \Sigma \quad \Sigma; \cdot \vdash \sigma : *}{\vdash \Sigma, `x : \sigma}\big(`x \notin \mathsf{dom}(\Sigma)\big)$$

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

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

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

$$\frac{}{(\lambda x.t)\ s \rightsquigarrow [s/x]t} \qquad \frac{}{\mathbf{let}\ x = s\ \mathbf{in}\ t \rightsquigarrow [s/x]t}$$

$$\frac{C\,\overline{x} \to t \in \varphi}{\varphi^{\psi}(C\,\overline{t}) \rightsquigarrow [\overline{t}/\overline{x}]t} \qquad \frac{}{\mathsf{Mlt}\ x.\varphi^{\psi}\ (\mathsf{In}^{\kappa}t) \rightsquigarrow [\mathsf{Mlt}\ x.\varphi^{\psi}/x]\varphi^{\psi}\ t} \qquad \frac{`x = t \in \overline{D}}{`x \rightsquigarrow t}$$

$$\frac{t \rightsquigarrow t'}{\lambda x.t \rightsquigarrow \lambda x.t'} \qquad \frac{r \rightsquigarrow r'}{r\ s \rightsquigarrow r'\ s} \qquad \frac{s \rightsquigarrow s'}{r\ s \rightsquigarrow r\ s'} \qquad \frac{t_i \rightsquigarrow t_i'}{C\ t_1 \cdots t_i \cdots t_n \rightsquigarrow C\ t_1 \cdots t_i' \cdots t_n}$$

$$\frac{s \rightsquigarrow s'}{\mathbf{let}\ x = s\ \mathbf{in}\ t \rightsquigarrow \mathbf{let}\ x = s'\ \mathbf{in}\ t} \qquad \frac{t \rightsquigarrow t'}{\mathbf{let}\ x = s\ \mathbf{in}\ t \rightsquigarrow \mathbf{let}\ x = s\ \mathbf{in}\ t'} \qquad \frac{\varphi^{\psi} \rightsquigarrow \varphi'^{\psi}}{\mathsf{Mlt}\ x.\varphi^{\psi} \rightsquigarrow \mathsf{Mlt}\ x.\varphi'^{\psi}}$$

$$\frac{t_i \rightsquigarrow t_i'}{(C_1\,\overline{x_1} \to t_1; \cdots; C_i\,\overline{x_i} \to t_i; \cdots; C_n\,\overline{x_n} \to t_n)^{\psi} \ \rightsquigarrow\ (C_1\,\overline{x_1} \to t_1; \cdots; C_i\,\overline{x_i} \to t_i'; \cdots; C_n\,\overline{x_n} \to t_n)^{\psi}}$$

Fig. 3. Syntax and Reduction rules of Nax

Sorting:

$$\boxed{\Sigma \vdash \kappa : \square} \quad (A)\frac{}{\Sigma \vdash * : \square} \quad (R)\frac{\Sigma \vdash \kappa : \square \qquad \Sigma \vdash \kappa' : \square}{\Sigma \vdash \kappa \to \kappa' : \square} \quad (Ri)\frac{\Sigma; \cdot \vdash A : * \qquad \Sigma \vdash \kappa : \square}{\Sigma \vdash A \to \kappa : \square}$$

Kinding: $\boxed{\Sigma; \Delta \vdash \sigma : \kappa}$ $\quad (\forall)\dfrac{\Sigma; \Delta, X^{\kappa} \vdash \sigma : *}{\Sigma; \Delta \vdash \forall X.\sigma : *} \quad (\forall i)\dfrac{\Sigma; \Delta, i^{A} \vdash \sigma : *}{\Sigma; \Delta \vdash \forall i.\sigma : *}$

$$\boxed{\Sigma; \Delta \vdash F : \kappa} \quad (Var)\frac{X^{\kappa} \in \Delta \qquad \Sigma \vdash \Delta}{\Sigma; \Delta \vdash X : \kappa} \quad (TCon)\frac{T : \kappa \in \Sigma \qquad \Sigma \vdash \Delta}{\Sigma; \Delta \vdash T : \kappa} \quad (\mu)\frac{\Sigma; \Delta \vdash T\,\overline{\tau} : \kappa \to \kappa}{\Sigma; \Delta \vdash \mu^{\kappa}(T\,\overline{\tau}) : \kappa}$$

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

$$(\to)\frac{\Sigma; \Delta \vdash A : * \qquad \Sigma; \Delta \vdash B : *}{\Sigma; \Delta \vdash A \to B : *} \quad (Conv)\frac{\Sigma; \Delta \vdash A : \kappa \qquad \Sigma \vdash \kappa = \kappa' : \square}{\Sigma; \Delta \vdash A : \kappa'}$$

Typing:

$$\boxed{\Sigma \vdash Prog : A} \quad (\cdot; t)\frac{\Sigma; \cdot; \cdot \vdash t : A}{\Sigma \vdash \cdot; t : A} \quad (D)\frac{\Sigma \vdash D \Rrightarrow \Sigma' \qquad \Sigma' \vdash \overline{D};\ t : A}{\Sigma \vdash D, \overline{D};\ t : A}$$

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

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

$$(: C)\frac{C : \sigma \in \Sigma \qquad \Sigma; \Delta \vdash A \prec \sigma \qquad \Sigma; \Delta \vdash \Gamma}{\Sigma; \Delta; \Gamma \vdash C : A} \quad (: \text{`})\frac{\text{`}x : \sigma \in \Sigma \qquad \Sigma; \Delta \vdash A \prec \sigma \qquad \Sigma; \Delta \vdash \Gamma}{\Sigma; \Delta; \Gamma \vdash \text{`}x : A}$$

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

$$(\text{let})\frac{\Sigma; \Delta, \overline{\iota^{K}}; \Gamma \vdash s : A \qquad \Sigma; \Delta; \Gamma, x : \forall \overline{\iota}.A \vdash t : B}{\Sigma; \Delta; \Gamma \vdash \mathbf{let}\ x = s\ \mathbf{in}\ t : B}\ \begin{pmatrix}\overline{\iota} \cap \mathrm{FV}(s) = \emptyset \\ \overline{\iota} \cap \mathrm{FV}(\Gamma) = \emptyset\end{pmatrix} \quad (\text{case})\frac{\Sigma; \Delta; \Gamma \vdash^{\psi} \varphi : \forall \overline{\iota}.F\,\overline{\iota} \to \psi(\overline{\iota})}{\Sigma; \Delta; \Gamma \vdash \varphi^{\psi} : F\ \overline{\tau} \to \psi(\overline{\tau})}$$

$$(\text{MIt})\frac{\Sigma; \Delta, X^{\kappa}; \Gamma, x : \forall \overline{\iota'}.X\overline{\iota'} \to \psi(\overline{\iota'}) \vdash^{\psi} \varphi : \forall \overline{\iota}.F\,X\,\overline{\iota} \to \psi(\overline{\iota})}{\Sigma; \Delta; \Gamma \vdash \mathsf{MIt}\ x.\varphi^{\psi} : \mu^{\kappa}\ F\ \overline{\tau} \to \psi(\overline{\tau})}\big(X \notin \mathrm{FV}(\Gamma)\big)$$

$$(\text{In})\frac{}{\Sigma; \Delta; \Gamma \vdash \mathsf{In}^{\kappa} : F(\mu^{\kappa}F)\,\overline{\tau} \to \mu^{\kappa}F\ \overline{\tau}}$$

$$\boxed{\Sigma; \Delta; \Gamma \vdash^{\psi} \varphi : \sigma} \quad \frac{\Sigma|_T = \overline{C_k : \sigma_k}^{\,k=1..n} \qquad \overline{\Sigma; \Delta \vdash \overline{A} \to T\,\overline{\tau}\,\overline{\tau}_k \prec \sigma_k\ \Sigma; \Delta; \Gamma, \overline{x : A} \vdash t : \psi(\overline{\tau}_k)}^{\,k=1..n}}{\Sigma; \Delta; \Gamma \vdash^{\psi} \overline{C_k\ \overline{x} \to t}^{\,k=1..n} : \forall \overline{\iota}.T\,\overline{\tau}\,\overline{\iota} \to \psi(\overline{\iota})}$$

Extending the Global Context: $\boxed{\Sigma \vdash D \Rrightarrow \Sigma'}$

$$(\Sigma, T)\frac{\overline{\Sigma, T : \kappa;\ \overline{\iota^{K}} \vdash \overline{A} \to T\,\overline{\tau} : *}}{\Sigma \vdash \mathbf{data}\ T : \kappa\ \mathbf{where}\ \overline{C : \overline{A} \to T\,\overline{\tau}} \ \Rrightarrow\ \Sigma, T : \kappa, \overline{C : \forall \overline{\iota}.\overline{A} \to T\,\overline{\tau}}}$$

$$(\Sigma, \text{`}x)\frac{\Sigma; \overline{\iota^{K}}; \cdot \vdash t : A}{\Sigma \vdash \text{`}x = t \Rrightarrow \Sigma, \text{`}x : \forall \overline{\iota}.A}\big(\overline{\iota} \cap \mathrm{FV}(t) = \emptyset\big)$$

Fig. 4. Typing rules of Nax

Type constructors and their kinding rules The syntax for type constructors of Nax is similar to F_i, but different from F_i in two aspects.

Firstly, polymorphic types are separate out as type schemes (σ) in Nax since the type system of Nax is in flavour of Hindley-Milner to support type inference (or, reconstruction).

Secondly, there are no type level abstractions and index abstractions in Nax. Instead of defining type constructors expecting type arguments by abstraction and index abstraction at type level, Nax supports datatype declarations (**data** $T : \kappa$ **where** ...) and recursive type operators (μ^κ) as language constructs.

Intuitively, the kinding rule for the recursive type operator should be $\Sigma \vdash \mu^\kappa : (\kappa \to \kappa) \to \kappa$. However, we restrict the recursive type operator (μ^κ) only to be applied to datatypes ($T\,\overline{\tau}$). This restriction is evident in both the type constructor syntax in Fig. 3 and the kinding rule (μ) in Fig. 4. What this restriction really excludes are nested applications of recursive type operators. For instance, $\mu^\kappa(\mu^{\kappa\to\kappa}F)$ where $F : (\kappa \to \kappa) \to \kappa \to \kappa$ is not allowed although it would be well-kinded under the less restrictive kinding rule ($\Sigma \vdash \mu^\kappa : (\kappa \to \kappa) \to \kappa$). Motivation behind this restriction is type inference. In order to infer a type for a Mendler style iterator, we need to restrict the form of its body since the body must be polymorphic over the indices (see (MIt) rule). In general, we do not want polymorphic types to be first class since we want type inference. One simple design choice is to allow case branches (φ) to have polymorphic types, or type schemes, and annotate case branches with index transformers (φ^ψ). For the exact same reason (i.e., type inference), we restrict the body of the Mendler style iterators be case terms (i.e., ($\mathsf{MIt}\ x.\varphi^\psi$) instead of ($\mathsf{MIt}\ x.t$)).

Terms and their typing rules The term syntax of Nax has six additional term constructs than F_i: data constructors (C), top level variables ($`x$), polymorphic let bindings (**let** $x = s$ **in** t), eliminators for datatypes (φ^ψ), Mendler style iterators ($\mathsf{MIt}\ x.\varphi^\psi$), and constructors for recursive types (In^κ). Typing rules for them are provided in Fig. 4.

The typing rules ($: C$) and ($: `$) are for data constructors ($C$) and top level variables ($`x$) bound in the top level context (Σ). Data constructors (C) are introduced from datatype declarations (**data** $T : \kappa$ **where** ...) by the rule (Σ, T), and top level variables ($`x$) are introduced from top level definitions ($`x = t$) by the rule ($\Sigma, `x$). The typing rules ($: C$) and ($: `$) behave similar to the rule ($:$) for the variables and the rule ($: i$) for index variables. All these four rules ($:$), ($: i$), ($: C$), and ($: `$) for identifiers look up a certain context (one of the three zones Σ, Δ, and Γ). Since the Nax type system is in flavour of Hindley-Milner, identifiers are bound to type schemes (σ) and the typing rules for the identifiers instantiate type schemes to types (A). Note, a type instantiation ($\Sigma; \Delta \vdash A \prec \sigma$) is a judgement under the top level context (Σ) and the type level context (Δ), since the instantiated type needs to be well-kinded under Σ and Δ.

Polymorphic let bindings in Nax are just the usual polymorphic bindings of Hindley-Milner type system for generalizing types of local definitions into type

schemes. In Nax, we generalize over term indices as well as types. The typing rule for let bindings is the (let) rule.

Eliminators for datatypes (φ^{ψ}), or case-terms, are case branches (φ) annotated by index transformers (ψ). For non-indexed types, case-terms are the usual single level pattern matching expressions in functional languages. For example, a Nax case-term applied to non-indexed typed term ($\varphi^{\cdot}\ s$) corresponds to a Haskell case-expression over that term (**case** s **of** $\{\varphi\}$). Note, we give trivial index transformer annotation (i.e., $\psi = \cdot$) for non-indexed types (e.g., booleans, natural numbers) since there are no indices to worry about. For indexed types, the indexed transformer annotations proivide useful information for type reconstruction. For example, consider the following datatype declaration:

$$\textbf{data}\ \texttt{Judgement} : \texttt{Bool} \to * \ \textbf{where}\ \begin{array}{l}\texttt{TJ} : \texttt{Formula} \to \texttt{Judgement}\ \{\texttt{True}\};\\ \texttt{FJ} : \texttt{Formula} \to \texttt{Judgement}\ \{\texttt{False}\}\end{array}$$

The datatype `Judgement` is index by boolean terms (e.g., `True` and `False` of type `Bool`). The data constructor `TJ` contains a formula expected to be true and the data constructor `FJ` contains a formula expected to be false. We can define a function, which produces an inverted judgement by negating the formula contained in a given judgement, as follows[6]:

$$\begin{pmatrix}\texttt{TJ}\ x \to \texttt{FJ}(neg\ \ x);\\ \texttt{FJ}\ x \to \texttt{TJ}(neg\ \ x)\end{pmatrix}^{i\,.\,\texttt{Judgement}\ \{`not\ i\}}$$

where neg is a function that produces negated formula and $`not$ is a top level function that negates booleans.

Note that the index transformer ($i\,.\,\texttt{Judgement}\ \{`not\ i\}$) captures the idea that the resulting inverted judgement has opposite expectations from the given judgement. The types of such case-terms involving indexed types can also be inferred when we annotate the case-terms with appropriate index transformers. Reduction rules for case-terms (Fig. 3) are standard.

Mendler style iterators ($\mathsf{MIt}\ x.\varphi^{\psi}$) are eliminators for recursive types. A case-term expects a datatype argument (of type $T\,\overline{\tau}$). A Mendler style iterator expects a recursive type argument (of type $\mu^{\kappa}(T\,\overline{\tau})$). Intuitively, Mendler style iterators open up the recursive type ($\mu^{\kappa}(T\,\overline{\tau})$) and case branch over its base datatype structure (of type $T\,\overline{\tau}$). This intuition is captured by the reduction rule for Mendler style iterators (Fig. 3): $\mathsf{MIt}\ x.\varphi^{\psi}\ (\mathsf{In}^{\kappa} t) \rightsquigarrow [\mathsf{MIt}\ x.\varphi^{\psi}/x]\varphi^{\psi}\ t$. Note that a Mendler style iterator ($\mathsf{MIt}\ x.\varphi^{\psi}$) applied to a term of recursive type ($\mathsf{In}^{\kappa} t$) constructed by the In^{κ} constructor reduces to a case-term ($[\mathsf{MIt}\ x.\varphi^{\psi}/x]\varphi^{\psi}$) applied to the base structure (t) contained in the In^{κ} constructor. The variable (x) bound by MIt is a label for the recursive call. Note that the case-term ($[\mathsf{MIt}\ x.\varphi^{\psi}/x]\varphi^{\psi}$) appearing in the reduction rule substitutes x with the Mendler style iterator itself. However, unlike the fixpoint operator for unrestricted general recursion, Mendler style iterators are guaranteed to normalize because of their carefully designed typing rule (MIt) due to Mendler.

The constructors for recursive types (In^{κ}) are standard (see rule (In) in Fig. 4). The kind annotation κ on the In^{κ} constructor aids kind inference. If

[6] case branches are laid out in multiple lines for better readability

we were to simulate the recursive type operator μ^κ and its constructor In^κ in a functional language like Haskell (with GADT and kind annotation extensions), we would simulate them by the following recursive datatype:

$$\textbf{data}\ \mu^\kappa : (\kappa \to \kappa) \to \kappa\ \textbf{where}\ \mathsf{In}^\kappa : X(\mu^\kappa X)\bar{\iota} \to \mu^\kappa X\,\bar{\iota}$$

However, such a simulation of μ^κ by a recursive datatype cannot guarantee normalization of the language, since unlimited elimination of In^κ via case branches is already powerful enough to encode non-terminating computation even without using any recursion at term level. Thus, Nax provides μ^κ and In^κ as primitive language constructs, and only allow elimination of In^κ via Mendler style iteration.

Nax programs and their typing rules A Nax program $(\overline{D}; t)$ is a list of declarations $(\overline{D})$ followed by a term (t). A declaration can be either a datatype declaration $(\textbf{data}\ T : \kappa\ \textbf{where}\ \ldots)$ or a top level definition $(`x = t)$. The list of declarations $(\overline{D})$ are processed by the rules $(\Sigma, T)$ and $(\Sigma, `x)$ before type checking the term (t). The kinding and typing information from the datatype declarations and the top level definitions preceding the term are captured into the top level context (Σ) according to the rules (Σ, T) and $(\Sigma, `x)$. The top level context extended by these rules are used for type checking the term following the list of declarations. Therefore, the sorting, kinding, and typing rules of Nax (Fig. 4) involves Σ in addtion to Δ and Γ, while the corresponding rule of F_i (Fig. **??**) involves Δ and Γ only.

Reduction rules Reduction rules are defined in Fig. 3. First five rules are the redex rules that makes an actual reduction step on redexes. A redex is be one of the following: a lambda term applied to an argument, a let binding, a case term applied to a constructor term, a Mendler style iterator applied to an In^κ-constructed term, and a top level variable.

Note, the reduction rule for $`x$ mentions $\overline{D}$). Although we illustrate the reduction as a relation on terms $(t \rightsquigarrow t')$, we implicitly assume that there exists some fixed list of declarations $(\overline{D})$ for the reduction relation $(\rightsquigarrow)$. In order to make a reduction step for top level variables, we need to know the top level definition for $`x$, which should be contained in $\overline{D}$. Since the list of declarations are given by the input program $(\overline{D}; t)$ to type check, it is non-ambiguous which $\overline{D}$ to use for reducing $`x$. In case when it is ambiguous, we could use a notation like $t \overset{\overline{D}}{\rightsquigarrow} t'$ to make it more precise.

The other rules, following the top level variable reduction rule $(`x \rightsquigarrow t)$, are context rules to make a reduction step for the terms whose redexes appear inside their subterms.

3.2 Syntax-directed type system and type inference

The kinding and typing rules of Nax illustrated in Fig. 4 is not syntax directed since the conversion rules $(Conv)$ and $(=)$ are not syntax directed. These con-

version rules can apply to anywhere regardless of the syntactic category of terms and types.

We can easily adapt the system to be syntax directed by embedding the conversion rules into application-like rules (e.g., $(@i)$, $(\to E)$). Among the kinding rules, the only place where conversion is truly necessary is in the index application rule $(@i)$. We can define the syntax directed application rule $(@i)_s$ as follows:

$$(@i)_s \frac{\Sigma; \Delta \vdash_s F : A \to \kappa \qquad \Sigma; \Delta; \cdot \vdash_s s : A' \qquad \Sigma; \cdot \vdash_s A = A' : *}{\Sigma; \Delta \vdash_s F\,\{s\} : \kappa}$$

Among the typing rules, we need to embed conversion into the $(\to E)$ rule and probably the rules (let) and (MIt), and the branch checking rule as well. Once we finish describing the syntax directed type system, we should prove that it is equivalent to the typing rules in Fig. 4. We are still working on describing the type inference algorithm. Then, we would need to prove the correctness of the type inference algorithm with respect to the syntax directec typing rules.

4 Embedding Nax into strongly normalzing calculi

We will prove strong normalization and logical consistency of Nax by embedding Nax into a calculi known to have theose desired properties. We have developed System F_i, which is an extension of F_ω with erasable term indices, and proved its properties.[7] We can embed datatypes of Nax and the MIt and MsfIt combinator families into System F_i. But, other combinator families such as MPr is known to be embeddable into a different calculus, System Fix_ω [Abel and Matthes(2004)], when there are no term indices. We can similarly extend Fix_ω with erasable term indices – we call this calculus Fix_i, which can embed MPr with term indices. Properties if Fix_i needs to be checked but we strongly believe that the desired properties, i.e., strong normalization and logical consistency, will hold in Fix_i as well as in F_i.

5 Implementation

We used Haskell and its libraries (e.g., unbound [Weirich et al.(2011)Weirich, Yorgey, and Sheard]) to implement to the proptotpye of Nax.

6 Summary

6.1 Ongoing and future work

Here is the list of ongoing and future work:

- Correctness of the type inference algorithm
- Large eliminations
- Generalized arrow types in the recursion combinators
- Non-logical language fragments

[7] We submitted a paper on System F_i on POPL recently.

References

[Abel and Matthes(2004)] Abel, A., Matthes, R.: Fixed points of type constructors and primitive recursion. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL. Lecture Notes in Computer Science, vol. 3210, pp. 190–204. Springer (2004)

[Bird and Meertens(1998)] Bird, R.S., Meertens, L.G.L.T.: Nested datatypes. In: Proceedings of the Mathematics of Program Construction. pp. 52–67. MPC '98, Springer-Verlag, London, UK, UK (1998)

[Cheney and Hinze(2002)] Cheney, J., Hinze, R.: A lightweight implementation of generics and dynamics. In: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell. pp. 90–104. Haskell '02, ACM, New York, NY, USA (2002)

[Cheney and Hinze(2003)] Cheney, J., Hinze, R.: First-class phantom types. Tech. rep., Cornell University (2003)

[Fegaras and Sheard(1996)] Fegaras, L., Sheard, T.: Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 284–294. POPL '96, ACM (1996)

[Henglein(1993)] Henglein, F.: Type inference with polymorphic recursion. ACM Trans. Program. Lang. Syst. 15(2), 253–289 (Apr 1993)

[Weirich et al.(2011)Weirich, Yorgey, and Sheard] Weirich, S., Yorgey, B.A., Sheard, T.: Binders unbound. In: Proceedings of the 16th ACM SIGPLAN international conference on Functional programming. pp. 333–345. ICFP '11, ACM, New York, NY, USA (2011)

[Xi et al.(2003)Xi, Chen, and Chen] Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 224–235. POPL '03, ACM, New York, NY, USA (2003)

[Zenger(1997)] Zenger, C.: Indexed types. Theoretical Computer Science 187, 147–165 (1997)