The Nax programming language (work in progress)

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

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

1 Introduction

During the past decade, the functional programming community achieved partial success in their goal of maintaining fine-grained properties by moderate extensions to the type system of functional languages[4, 3, 9]. This approach is often called *"lightweight"* (e.g., lightweight dependent types[4], lightweight program verification), since using a full blown proof assistant to maintain similar properties is likely to require much more effort (heavyweight).

The Generalized Algebraic Data Type (GADT) extension, implemented in the Glasgow Haskell Compiler (GHC), has made this approach possible even when performing everyday functional programming tasks. Unfortunately, implementations supporting a lightweight approach (e.g., GHC) lack **correctness guarantees** and **type inference** in general. In addition, practical implementations often lack support for GADTs indexed by terms, so **term indices are faked** (or simulated) by additional type structure replicating the requisite 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* and also *guarantee correctness* since these calculi are based on strongly normalizing and logically consistent systems. 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, the same fine-grained properties are not expressible in ordinary functional programming languages with only simple polymorphic types, since such languages lack the expressivity of dependent types. Even if these fine-grained properties were somehow expressible, one would not have any guarantee of correctness. Recall that general purpose programming languages are neither strongly normalizing nor logically consistent, because they are (by design) capable of expressing diverging computations. So, the lightweight approach in conventional functional languages can only raise programmers confidence of correctness

* supported by NSF grant 0910500.

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

(assuming that the inconsistent fragment of the type system was never used for reasoning about the desired properties) but cannot guarantee the correctness of the desired properties as is the norm in proof assistants.

Problem 2. ***Type inference***

Type inference makes type-safe programming pleasant when performomng everyday programming tasks, since programmers are freed from including 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 supports *type inference*.

An essential feature of the lightweight approach is *indexed datatypes*, which are datatypes with *heterogeneous type parameters* (a.k.a. *indices*) which are made possible by the GADT extension. Such datatypes, often used in lightweight approaches, are beyond the capabilities of polymorphic type schemes used in HM. Inclusion of just a simple subset of the indexed datatypes, such as nested datatypes [2], already make type inference undecidable [6]. More sophisticated uses add even more complication[7].

So, functional language implementations, that support the lightweight approach, require type annotations on both indexed datatype declarations and on function definitions that pattern match indexed datatypes, in order to recover a semblence of type inference.[5] Type annotations on datatype declarations are absolutely necessary when either the result types of their data constructors have indices or when the argument types of their data constructors have existential indices. However, it still an open question of *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 compleytely known at type checking time), unlike full-fledged dependent types, as used in proof assistants, which can depend on both static and dynamic values. 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 [4] (Fig. 1) used in datatype generic programming are typical examples of type-indexed datatypes. The length-indexed list, or `Vector`, (Fig. 2) is an example of a term-indexed datatype. However, the `Vector` datatype declaration in Fig. 2 uses faked term indices. These indexes are faked because rather than use the real term constructors of the natural numbers (defined by `data Nat = Succ Nat | Zero`) it uses the uninhabited types `Succ` and `Zero` to simulate the data constructors of `Nat`. Such faked term indices are problematic since they (1) duplicate code (i.e., operations on `Nat` must be redefined at type level) and (2) have less precise semantics than true term indices (e.g., cannot prevent ill-typed types such as `Succ Bool`).

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

Although indexed datatypes with true term indices have been studied[10], including term indices in practical functional languages is not trivial. Allowing arbitrary terms at the 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.

```
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 proptotype 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).

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

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

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

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

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*
	$nil = \mathsf{In}^{[*]}\ Nil$	
	$cons\ x\ xs = \mathsf{In}^{[*]}\ (Cons\ x\ xs)$	
$x = Cons\ 3\ (Cons\ 2\ Nil)$	$x = cons\ 3\ (cons\ 2\ nil)$	$x = cons\ 3\ (cons\ 2\ nil)$

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 $\mathsf{MIt}^{\cdot}$ 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 $\mathsf{MPr}^{\cdot}$ provides a safe, yet abstract mechanism, to support this. The Mendler $\mathsf{MPr}^{\cdot}$ 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 $\mathsf{MPr}^{\cdot}$.

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 $\mathsf{McvIt}^{\cdot}$ provides a safe, yet abstract mechanism, to support this. The function *fibonacci* is a classic example of this kind of recursion. The Mendler $\mathsf{McvIt}^{\cdot}$ 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.

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

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[6]. 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.

[6] 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.

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

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.

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

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:

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

$$sum\ (Fork\ x) = \lambda f \to sum\ x\ (\lambda(a,b) \to f\ a + f\ b)$$
$$sumTree\ t = genericSum\ t\ (\lambda x \to x)$$

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 $\{\, a.\ (a \to Int) \to Int\}$ 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 $(PowerTree\ a)$ which is a synonym for $((\mu^{[*\to *]}\ Nest)\ a)$. The recursive caller *sum* has type $(\forall\ a\ .\ r\ a \to (a \to Int) \to Int)$, for some abstract type constructor r. Recall r has an index, so it must be a type constructor, not a type. The patterns $(Tip\ x)$ and $(Fork\ x)$ have type $(Nest\ r\ a)$ and the right hand sides of the equations: $(\lambda f \to f\ x)$ and $(\lambda f \to sum\ x\ (\lambda(a,b) \to f\ a + f\ b))$, have type $((a \to Int) \to Int)$. Note that the dependency of $((a \to Int) \to Int)$ 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 $\{\, a\ .\ (a \to Int) \to Int\}$, as a function: $\psi\ a = (a \to Int) \to Int$, 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**
$f\ p_i = e_i$

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

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: $Vector \ a \ \{n\} \to Int$. We can write this as follows:

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

Let's study an example with a more interesting index transformation. A term with type $(Proof \ \{E\} \ \{n\})$, which is synonymous with the type $(\mu^{[\,Tag \to Nat \to *]} \ P \ \{E\} \ \{n\})$, 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:
$Proof \ \{E\} \ \{n\} \to Proof \ \{O\} \ \{`succ \ n\}$, and
$Proof \ \{O\} \ \{n\} \to Proof \ \{E\} \ \{`succ \ n\}$.
We can capture this dependency by defining the term-level function *flip*, and using an $\mathsf{MIt}^{\cdot}$ with the index transformer: $\{\{t\} \ \{i\} \ .\ Proof \ \{`flip \ t\} \ \{`succ \ i\}\}$.

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

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: $Vector \ a \ \{n\} \to Either \ (Even \ \{n\}) \ (Odd \ \{n\})$. Here, *Even* and *Odd* are synonyms for particular kinds of *Proof*. To write this function, we need the index transformation: $\{\{n\} \ .\ Either \ (Even \ \{n\}) \ (Odd \ \{n\})\}$.

```
synonym Even {x} = Proof {E} {x}
synonym Odd  {x} = Proof {O} {x}
proveEvenOrOdd x = MIt^{{{n}. Either (Even {n}) (Odd {n})}} x with
                     prEOO Vnil          = Left base
                     prEOO (Vcons x xs) = case^{} prEOO xs of
                                            Left p  -> Right (stepE p)
                                            Right p -> Left (stepO p)
```

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.

```
data Lam : * -> * where
  App :: r -> r -> Lam r
  Abs :: (r -> r) -> Lam r
    deriving fixpoint Term
apply = abs (\f -> abs (\x -> app f x))
```

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: $abs : (Term \to Term) \to Term$, 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 MIt' and MPr' combinators limit the analysis of such types in a manner that precludes non-terminating computations. The Mendler-style combinator, McvIt', is too expressive to exclude non-terminating computations, and must be restricted to recursive datatypes with no negative occurrences.

Even though MIt' and MPr' 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 MsfIt'. This combinator is based upon an interesting programming trick, first described by Sheard and Fegaras [5], hence the "sf" in the name MsfIt'. The abstraction supported by MsfIt' is as follows:

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

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*.

```
                                  -- cat : List String → String
                                  -- new : Int → String
new n = cat ["x", show n]
                                  -- showHelp : Term → (Int → String)
                                  -- sh : r → (Int → String)
                                  -- inv : (Int → String) → r
                                  -- (λn → new m) : Int → String
showHelp x =
  Msflt{} x with
    sh inv (App x y) = λm → cat ["(", sh x m, " ", sh y m, ")"]
    sh inv (Abs f)   = λm → cat ["(fn ", new m, " => ",
                                 sh (f (inv (λn → new m))) (m + 1), ")"]
showTerm x = showHelp x 0
showTerm apply : List Char = "(fn x0 => (fn x1 => (x0 x1)))"
```

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. We have formalized this is the paper *System* F_i*: a higher-order polymorphic* λ*-calculus with erasable term indices*[**?**] which we have submitted to POPL.
– 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.

3 Embedding Nax into strongly normalizing calculus

Our approach to formalizing the Nax as a logical language, is to embed each logical feature of Nax into a lower level language, System F_i, which we have proven to be strongly normalizing and logically consistent[**?**]. We have designed System F_i, which is an extension of F_ω with erasable term indices, and proved its properties.[7] Our approach of distinguishing type and term indices is unique, and requires the extension of some previous work on normalizing calculi.

4 Implementation

We used Haskell and its libraries (e.g., unbound [8]) to implement a prototype of Nax as an interpreter.

5 Future Work

Nax is one thread of research in the Trellys project, a collaborative initiative to design a dependently-typed programming language with simple support for general recursion and other convenient but logically unsound features, yet still maintain a logically sound core. Here is a partial list of ongoing and future work in the Nax thread.

– *Embedding all of Nax into a strongly normalizing calculus*
We can embed datatypes of Nax and the MIt and MsfIt combinator families into System F_i. But, other combinator families, such as MPr, are only known

[7] We submitted a paper on System F_i on POPL recently. Along with this submission, the draft of that paper on System F_i is included as a supplimentary material.

to be embeddable into a different calculus, System Fix_ω [1], which does not support term indices. We believe we can similarly extend Fix_ω with erasable term indices – we call this calculus Fix_i, which can then embed MPr over datatypes with term indices. Properties of 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.

– *Including both a programatic and a logical fragment*
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. We believe we can extend the proof principles outlined in the paper *Step-Indexed Normalization for a Language with General Recursion*[**?**]

– *Large eliminations*
The current prototype of Nax only supports a limited form of large elimination (i.e. mapping indices from argument types to result types) due to the limited syntax of the index transformer. We hope to enrich the index transformer syntax to support more expressive large eliminations (e.g., if-then-else, or more generally, case expressions in index transformers) and still maintain our design goals of having both type inference and a logically consistent type system. Again, we will make sure that such new features are safe by embedding the new feature into the calculus Nax is based on, such as System F_i.

Bibliography

[1] 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)

[2] 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)

[3] 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)

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

[5] 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)

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

[7] Lin, C.K.: Practical Type Inference for the GADT Type System. Ph.D. thesis, Portland State University (2010)

[8] 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)

[9] 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)

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