Programming with Term-Indexed Types in Nax

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

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

Abstract. Nax, a language designed to support term indices, enjoys the merits of logical consistency (in the Curry–Howard sense) like formal proof assistants (e.g., Agda) and Hindley–Milner-style type inference like functional languages (e.g., Haskell). Examples in this article demonstrate that programming with term indices in Nax is as pleasant as programming in Haskell or Agda, while enjoying the merits of both. We also discuss strengths and limitations of the Nax approach to indexed types.

Keywords: indexed datatypes, GADTs, lightweight dependent types, universe polymorphism, universe subtyping, term-indexed types

1 Introduction

During the past decade, the functional programming community achieved partial success in their goal of maintaining fine-grained properties by only moderately extending functional language type systems [5, 4, 22]. This approach is often called *"lightweight"*[3] in contrast to the approach taken by fully dependent type systems (e.g., Coq, Agda). The Generalized Algebraic Data Type (GADT) extension, implemented in the Glasgow Haskell Compiler (GHC) and in OCaml[15, 10], has made the lightweight approach widely applicable to everyday functional programming tasks.

Unfortunately, most practical lightweight implementations lack **logical consistency** and **type inference**. In addition, they often lack term indexing, so **term indices are faked** (or, simulated) by additional type structure replicating the requisite term structure. A recent extension in GHC, datatype promotion [24], addresses the issue of term indices, but the issues of logical consistency and type inference still remain.

Nax is a programming language designed to support both type and term indexed datatypes, logical consistency, and type inference.

(1) **Nax is strongly normalizing and logically consistent.**
Types in Nax can be given logical interpretations as propositions and the programs of those types as proofs of those propositions. Theories behind strong normalization and logical consistency are Mendler-style recursion [1] and System F_i (to be published).

[⋆] supported by NSF grant 0910500.
[3] e.g., http://okmij.org/ftp/Computation/lightweight-dependent-typing.html

(2) **Nax supports Hindley–Milner-style type inference.**
Nax needs few type annotations. In particular, annotations for top-level functions, which are usually required for bidirectional type checking in dependently typed languages, are unnecessary. Type annotations are only required when introducing GADTs and as index transformers attached to pattern matching constructs for GADTs (Table 1).

(3) **Nax programs are expressive and concise.**
Nax programs are similar in size to their Haskell and Agda equivalents (Sect. 2), yet they still retain logical consistency and type inference. Despite several features unique to Nax, explained in Table 1, these features do not necessarily add verbosity.

(4) **Nax supports term indices within a relatively simple type system.**
The type system of Nax (Sect. 3.1) is based on a two level universe structure, just like Haskell, yet it allows nested term indices (Sect. 3.2) as in languages based on a universe structure of countably many levels (e.g., Coq, Agda).

The detailed mechanism behind (1) and (2) above are beyond of the scope of this paper, and will be discussed in sequel publications and Ahn's dissertation. We demonstrate, through a series of examples – a type-preserving evaluator (Sect. 2.1), a generic path datatype (Sect. 2.2), and a stack-safe compiler (Sect. 2.3), that programming in Nax is as simple as programming in Haskell or Agda. Then, we discuss the key design principles behind indexed datatypes in Nax (Sect. 3.1) and its strengths and limitations (Sect. 3.2).

The "**deriving** fixpoint T" clause following **data** $F : \overline{k} \to \kappa \to \kappa$ **where** $\cdots$ automatically derives a recursive type synonym $T\,\overline{a} = \mu_{[\kappa]}\ (F\,\overline{a}) : \kappa$ and its constructor functions. For instance, the deriving clause below left automatically derives the definitions below right:

data $L : \star \to \star \to \star$ **where**	**synonym** $List\ a = \mu_{[\star]}\ (L\ a)$
$Nil\ \ : L\ a\ r$	$nil = \mathtt{In}_{[\star]}\ Nil$
$Cons : a \to r \to L\ a\ r$	$cons\ x\ xs = \mathtt{In}_{[\star]}\ (Cons\ x\ xs)$
deriving fixpoint *List*	

The **synonym** keyword defines a type synonym, just like Haskell's **type** keyword.

In Nax, **data** declarations cannot be recursive. Instead, to define recursive types, one uses a fixpoint type operator $\mu_{[\kappa]} : (\kappa \to \kappa) \to \kappa$ over non-recursive base structures of kind $\kappa \to \kappa$ (e.g., $(L\ a) : \star \to \star$). Nax provides the usual data constructor $\mathtt{In}_{[\kappa]}$ to construct recursive values of the type $\mu_{[\kappa]}$. $\mathtt{In}_{[\kappa]}$ is used to define the normal constructor functions of recursive types (e.g., *nil* and *cons*).

However, one cannot pattern match against $\mathtt{In}_{[\kappa]}\ e$ in Nax. Instead, Nax provides several well-behaved (i.e., always terminating) Mendler-style recursion combinators, such as **mcata**, that work naturally over μ types, even with indices.

To support type inference, Nax requires programmers to annotate Mendler-style combinators with index tranformers. For instance, Nax can infer that the term $(\lambda x \to \mathbf{mcata}_{\{\{i\}\ \{j\}.\ T_2\ \{j\}\ \{i\}\}}\ x\ \ \mathbf{with}\ \cdots)$ has type $T_1\ \{i\}\ \{j\} \to T_2\ \{j\}\ \{i\}$ using the information in the index transformer $\{\{i\}\ \{j\}\ .\ T_2\ \{j\}\ \{i\}\}$.

Table 1: NAX features: **deriving** fixpoint, **synonym**, μ, In, and **mcata**

2 The trilingual Rosetta Stone

In this section, we introduce three examples (Figs. 1, 2 and 3) that use term indexed datatypes to enforce program invariants. Each example is written in three different languages – like the Rosetta Stone – Haskell on the left, Nax in the center, and Agda on the right. We have crafted these programs to look as similar as possible, by choosing the same identifiers and syntax structure whenever possible, so that anyone already familiar with Haskell-like languages or Agda-like languages will understand our Nax programs just by comparing them with the programs on the left and right. The features unique to Nax are summarized in Table 1.

The three examples we introduce are the following:

- A type-preserving evaluator for a simple expression language (Sect. 2.1),
- A generic *Path* datatype that can be specialized to various list-like structures with indices (Sect. 2.2), and
- A stack-safe compiler for the same simple expression langauge, which uses the *Path* datatype (Sect. 2.3).

We adopted the examples from Conor McBride's keynote talk [16] at ICFP 2012 (originally written in Agda). All the example code was tested in GHC 7.4.1 (should also work in later versions such as GHC 7.6.x), our prototype Nax implementation, and Agda 2.3.0.1.

2.1 Type preserving evaluator for an expression language

In a language that supports term-indices, one writes a type-preserving evaluator as follows: (1) define a datatype TypeUniverse which encodes types of the object language; (2) define a datatype Value (the range of object language evaluation) indexed by terms of the type TypeUniverse; (3) define a datatype ObjectLanguage indexed by the same type TypeUniverse; and (4) write the evaluator (from expressions to values) that preserves the term indices representing the type of the object language. Once the evaluator type checks, we are confident that the evaluator is type-preserving, relying on type preservation of the host-language type system. In Fig. 1, we provide a concrete example of such a type-preserving evaluator for a very simple expression language (*Expr*).

Our TypeUniverse (*Ty*) for the expression language consists of numbers and booleans, represented by the constants `I` and `B`. We want to evaluate an expression, to get a value, which may be either numeric (`IV` n) or boolean (`BV` b). Note that the both the *Expr* and *Val* datatypes are indexed by constant terms (`I` and `B`) of TypeUniverse (*Ty*). The terms of TypeUniverse are also known as *type representations*.

An expression (*Expr*) is either a value (`VAL` v), a numeric addition (`PLUS` e_1 e_2), or a conditional (`IF` e_0 e_1 e_2). Note that the term indices of *Expr* ensure that expressions are type correct by construction. For instance, a conditional expression `IF` e_0 e_1 e_2 can only be constructed when e_0 is a boolean expression (i.e., indexed by `B`) and e_1 and e_2 are expressions of the same type (i.e., both indexed

Haskell + GADTs, DataKinds, KindSignatures

```
data Ty = I | B

data Val :: Ty → ⋆ where
  IV :: Int → Val I
  BV :: Bool → Val B

plus_V :: Val I → Val I → Val I
plus_V (IV n) (IV m) = IV (n + m)

if_V :: Val B → Val t → Val t → Val t
if_V (BV b) v1 v2 = if b then v1 else v2


data Expr :: Ty → ⋆ where
  VAL  :: Val t → Expr t
  PLUS :: Expr I → Expr I → Expr I
  IF   :: Expr B →
            Expr t → Expr t → Expr t

eval :: Expr t → Val t
eval (VAL v)        = v
eval (PLUS e1 e2) =
  plus_V (eval e1) (eval e2)
eval (IF e0 e1 e2) =
  if_V (eval e0) (eval e1) (eval e2)
```

Nax

```
data Ty = I | B

data Val : Ty → ⋆ where
  IV : Int → Val {I}
  BV : Bool → Val {B}
-- plus_V : Val {I} → Val {I} → Val {I}
plus_V (IV n) (IV m) = IV (n + m)
-- if_V : Val {B} → Val {t} → Val {t} → Val {t}
if_V (BV b) v1 v2 = if b then v1 else v2


data E : (Ty → ⋆) → (Ty → ⋆)  where
  VAL  : Val {t} → E r {t}
  PLUS : r {I} → r {I} → E r {I}
  IF   : r {B} → r {t} → r {t} → E r {t}
    deriving fixpoint Expr
-- eval : Expr {t} → Val {t}
eval e = mcata_{{t}. Val {t}} e  with
           ev (VAL v)        = v
           ev (PLUS e1 e2) =
             plus_V (ev e1) (ev e2)
           ev (IF e0 e1 e2) =
             if_V (ev e0) (ev e1) (ev e2)
```

Agda

```
data Ty : ⋆  where I : Ty
                   B : Ty
data Val : Ty → ⋆  where
  IV : ℕ → Val I
  BV : Bool → Val B
plus_V : Val I → Val I → Val I
plus_V (IV n) (IV m) = IV (n + m)
if_V : Val B → {t : Ty} → Val t → Val t → Val t
if_V (BV b) v1 v2 = if b then v1 else v2


data Expr : Ty → ⋆  where
  VAL  : {t : Ty} → Val t → Expr t
  PLUS : Expr I → Expr I → Expr I
  IF   : Expr B →
           {t : Ty} → Expr t → Expr t → Expr t

eval : {t : Ty} → Expr t → Val t
eval (VAL v)        = v
eval (PLUS e1 e2) =
  plus_V (eval e1) (eval e2)
eval (IF e0 e1 e2) =
  if_V (eval e0) (eval e1) (eval e2)
```

Fig. 1: A type-preserving evaluator (*eval*) that evaluates an expression (*Expr*) to a value (*Val*).

by t). Then, we can write an evaluator (eval) (from expressions to values) which preserves the index that represents the object language type. The definition of eval is fairly straightforward, since our expression language is a very simple one. Note that the functions in Nax do not need type annotations (they appear as comments).

2.2 Generic *Path*s parametrized by a binary relation

In this section we introduce a generic Path datatype.[4] We will instantiate Path into three different types of lists – plain lists, length indexed lists (List' and Vec in Fig. 2) and a Code type, in order to write a stack-safe compiler (Fig. 3).

The type constructor Path expects three arguments, that is, $\mathit{Path}\ x\ \{i\}\ \{j\} : \star$. The argument $x : \{\iota\} \to \{\iota\} \to \star$ is binary relation describing legal transitions (i.e. $x\ \{i\}\ \{j\}$ is inhabited if one can legally step from i to j). The arguments $i : \iota$ and $j : \iota$ represent the initial and final vertices of the Path. A term of type $\mathit{Path}\ x\ \{i\}\ \{j\}$ witnesses a (possibly many step) path from i to j following the legal transition steps given by the relation $x : \{\iota\} \to \{\iota\} \to \star$.

The Path datatype provides two ways of constructing witnesses of paths. First, $\mathit{pNil} : \mathit{Path}\ x\ \{i\}\ \{i\}$ witnesses an empty path (or, ϵ-transition) from a vertex to itself, which always exists regardless of the choice of x. Second, $\mathit{pCons} : x\ \{i\}\ \{j\} \to \mathit{Path}\ x\ \{j\}\ \{k\} \to \mathit{Path}\ x\ \{i\}\ \{k\}$ witnesses a path from i to k, provided that there is a single step transition from i to j and that there exists a path from j to k.

The function $\mathit{append} : \mathit{Path}\ x\ \{i\}\ \{j\} \to \mathit{Path}\ x\ \{j\}\ \{k\} \to \mathit{Path}\ x\ \{i\}\ \{k\}$ witnesses that there exists a path from i to k provided that there exist two paths from i to j and from j to k. Note that the implementation of append is exactly the same as the usual append function for plain lists. We instantiate Path by providing a specific relation to instantiate the parameter x.

Plain lists ($\mathit{List}'\ a$) are path oblivious. That is, one can always add an element (a) to a list ($\mathit{List}'\ a$) to get a new list ($\mathit{List}'\ a$). We instantiate x to the degenerate relation $(\mathit{Elem}\ a) : \mathit{Unit} \to \mathit{Unit} \to \star$, which is tagged by a value of type a and witnesses a step with no interesting information. Then, we can define $\mathit{List}'\ a$ as a synonym of $\mathit{Path}\ (\mathit{Elem}\ a)\ \{U\}\ \{U\}$, and its constructors nil' and cons'.

Length indexed lists ($\mathit{Vec}\ a\ \{n\}$) need a natural number index to represent the length of the list. So, we instantiate x to a relation over natural numbers $(\mathit{Elem}_V\ a) : \mathit{Nat} \to \mathit{Nat} \to \star$ tagged by a value of type a witnessing steps of size one. The relation $(\mathit{Elem}_V\ a)$ counts down exactly one step, from $\mathit{succ}\ n$ to n, as described in the type signature of $\mathit{MkElem}_V : a \to \mathit{Elem}\ a\ \{`\mathit{succ}\ n\}\ \{n\}$. Then, we define $\mathit{Vec}\ a\ \{n\}$ as a synonym $\mathit{Path}\ (\mathit{Elem}_V\ a)\ \{n\}\ \{`\mathit{zero}\}$, counting down from n to zero. In Nax, in a declaration, backquoted identifiers appearing inside index terms enclosed by braces refer to functions or constants in the current scope (e.g., $`\mathit{zero}$ appearing in $\mathit{Path}\ (\mathit{Elem}_V\ a)\ \{n\}\ \{`\mathit{zero}\}$ refers to the predefined $\mathit{zero} : \mathit{Nat}$). Names without backquotes (e.g., n and a) are implicitly universally quantified.

[4] There is a Haskell library package for this http://hackage.haskell.org/package/thrist

HASKELL + GADTs, DataKinds, PolyKinds

```
data Path x i j  where
  PNil   :: Path x i i
  PCons :: x i j → Path x j k
                  → Path x i k

append :: Path x i j → Path x j k
                     → Path x i k
append PNil           ys = ys
append (PCons x xs) ys =
        PCons x (append xs ys)

-- instantiating to a plain regular list
data Elem a i j  where
  MkElem :: a → Elem a () ()
type List′ a = Path (Elem a) () ()
nil′ = PNil  :: List′ a
cons′ :: a → List′ a → List′ a
cons′ = PCons . MkElem
-- instantiating to a length indexed list
data Nat = Z | S Nat
data Elem_V a i j  where
  MkElem_V :: a → Elem_V a (S n) n
type Vec a n = Path (Elem_V a) n Z
vNil = PNil  :: Vec a Z
vCons :: a → Vec a n → Vec a (S n)
vCons = PCons . MkElem_V
```

NAX

```
data P : ({ι} → {ι} → ⋆) →
         ({ι} → {ι} → ⋆) →
         ({ι} → {ι} → ⋆)  where
  PNil   : P x r {i} {i}
  PCons : x {i} {j} → r {j} {k} → P x r {i} {k}
    deriving fixpoint Path
-- append : Path {i} {j} → Path {j} {k}
append l =       --       → Path {i} {k}
  mcata_{{i} {j}. Path x {j} {k}→Path x {i} {k}} l
    with
      app PNil           ys = ys
      app (PCons x xs) ys =
          pCons x (app xs ys)
-- instantiating to a plain regular list
data Unit = U
data Elem : ⋆ → Unit → Unit → ⋆ where
  MkElem : a → Elem a {U} {U}
synonym List′ a = Path (Elem a) {U} {U}
nil′ = pNil  -- :List′ a
-- cons′ : a → List′ a → List′ a
cons′ x = pCons (MkElem x)
-- instantiating to a length indexed list
data Elem_V : ⋆ → Nat → Nat → ⋆ where
  MkElem_V : a → Elem_V a {‘succ n} {n}
synonym Vec a {n}
            = Path (Elem_V a) {n} {‘zero}
vNil = pNil  -- :Vec a {‘zero}
-- vCons : a → Vec a {n} → Vec a {‘succ n}
vCons x = pCons (MkElem_V x)
```

AGDA

```
data Path {I : ⋆} (X : I → I → ⋆) : I → I → ⋆
  where
    PNil   : {i : I} → Path X i i
    PCons : {i j k : I} →
            X i j → Path X j k → Path X i k

append : {I : ⋆} → {X : I → I → ⋆} → {i j k : I}
     → Path X i j → Path X j k → Path X i k
append PNil           ys = ys
append (PCons x xs) ys =
        PCons x (append xs ys)
-- instantiating to a plain regular list
record Unit : ⋆  where constructor ⟨⟩
List′ : ⋆ → ⋆
List′ a = Path (λ i j → a) ⟨⟩ ⟨⟩
nil′ : {a : ⋆} → List′ a
nil′ = PNil
cons′ : {a : ⋆} → a → List′ a → List′ a
cons′ = PCons
-- instantiating to a length indexed list
Vec : ⋆ → ℕ → ⋆
Vec a n = Path (λ i j → a) n zero
vNil : {a : ⋆} → Vec a zero
vNil = PNil
vCons : {a : ⋆} {n : ℕ} →
        a → Vec a n → Vec a (suc n)
vCons = PCons
```

Fig. 2: A generic indexed list (*Path*) parameterized by a binary relation (x, X) over indices (i, j, k) and its instantiations (*List′*, *Vec*).

For plain lists and vectors, the relations, $(\mathit{Elem}\ a)$ and $(\mathit{Elem}_V\ a)$, are parameterized by the type a. That is, the transition step for adding one value to the path is always the same, independent of the value. Note that both *Elem* and Elem_V have only one data constructor *MkElem* and MkElem_V, respectively, since all "small" steps are the same. In the next subsection, we will instantiate *Path* with a relation witnessing stack configurations, with multiple constructors, each witnessing different transition steps for different machine instructions.

The Haskell code is similar to the Nax code, except that it uses general recursion and kinds are not explicitly annotated on datatypes.[5] In Agda, there is no need to define wrapper datatypes like *Elem* and Elem_V since type level functions are no different from term level functions.

2.3 Stack safe compiler for the expression language

In Figure 3, we implement a stack-safe compiler for the same expression language (*Expr* in Fig. 1) discussed in Sect. 2.1. In Fig. 1 of that section we implemented an index preserving evaluator $\mathit{eval} : \mathit{Expr}\ \{t\} \to \mathit{Val}\ \{t\}$. Here, the stack-safe compiler $\mathit{compile} : \mathit{Expr}\ \{t\} \to \mathit{Code}\ \{ts\}\ \{`\mathit{cons}\ t\ ts\}$ uses the index to enforce stack safety – an expression of type t compiles to some code, which when run on a stack machine with an initial stack configuration ts, terminates with the final stack configuration $\mathit{cons}\ t\ ts$.

A stack configuration is an abstraction of the stack that tracks only the types of the values stored there. We represent a stack configuration as a list of type representations $(\mathit{List}\ \mathit{Ty})$.[6] For instance, the configuration for the stack containing three values (from top to bottom) [3, True, 4] is $\mathit{cons}\ \mathtt{I}\ (\mathit{cons}\ \mathtt{B}\ (\mathit{cons}\ \mathtt{I}\ \mathit{Nil}))$.

To enforce stack safety, each instruction $(\mathit{Inst} : \mathit{List}\ \mathit{Ty} \to \mathit{List}\ \mathit{Ty} \to \star)$ is indexed with its initial and final stack configuration. For example, $\mathit{aDD} : \mathit{Inst}\ \{`\mathit{cons}\ \mathtt{I}\ (`\mathit{cons}\ \mathtt{I}\ ts)\}\ \{`\mathit{cons}\ \mathtt{I}\ ts\}$ instruction expects two numeric values on top of the stack. Running the *aDD* instruction will consume those two values, replacing them with a new numeric value (the result of the addition) on top of the stack, leaving the rest of the stack unchanged.

We define *Code* as a *Path* of stack consistent instructions (i.e., $\mathit{Code}\ \{ts\}\ \{ts'\}$ is a synonym for $\mathit{Path}\ \mathit{Inst}\ \{ts\}\ \{ts'\}$ from Sect. 2.2). For example, the com-

[5] In Haskell, kinds are inferred by default. The `KindSignatures` extension in GHC allows kind annotations.

[6] The astute reader may wonder why we use *List* instead of the already defined List' in Fig. 2, which is exactly the plain list we want. In Nax and Agda, it is possible to have term indices of $\mathit{List}'\ \mathit{Ty}$ instead of $\mathit{List}\ \mathit{Ty}$. (In Nax and Agda, the *List* datatype is defined in their standard libraries.) Unfortunately, it is not the case in Haskell. Haskell's datatype promotion does not allow promoting datatypes indexed by the already promoted datatypes. Recall that $\mathit{List}'\ \mathit{Ty}$ is a synonym of $\mathit{Path}\ (\mathit{Elem}\ \mathit{Ty})\ ()\ ()$, which cannot be promoted to an index since it is indexed by the already promoted unit term (). In the following section, we will discuss further on how the two approaches of Nax versus Haskell differ in their treatment of term indexed types.

Haskell + KindSignatures, TypeOperators, GADTs, DataKinds, PolyKinds

```
data List a = Nil | a :. List a ;  infixr :.
data Inst :: List Ty → List Ty → ⋆ where
  PUSH  :: Val t → Inst ts (t :. ts)
  ADD   :: Inst (I :. I :. ts) (I :. ts)
  IFPOP :: Path Inst ts ts' →
           Path Inst ts ts' →
           Inst (B :. ts) ts'



type Code sc sc' = Path Inst sc sc'

compile :: Expr t → Code ts (t :. ts)
compile (VAL v) =
  PCons (PUSH v) PNil
compile (PLUS e₁ e₂) =
  append (append (compile e₁) (compile e₂))
         (PCons ADD PNil)
compile (IF e₀ e₁ e₂) =
  append (compile e₀)
         (PCons (IFPOP (compile e₁)
                       (compile e₂))
                PNil)
```

Nax

```
data Instr : (List Ty → List Ty → ⋆) →
             (List Ty → List Ty → ⋆)  where
  PUSH  : Val {t} → Instr r {ts} {`cons t ts}
  ADD   : Instr r {`cons I (`cons I ts)} {`cons I ts}
  IFPOP : Path r {ts} {ts'} →
          Path r {ts} {ts'} →
          Instr r {`cons B ts} {ts'}
    deriving fixpoint Inst

synonym Code {sc} {sc'} = Path Inst {sc} {sc'}
  --  Path (μ[List Ty→List Ty→⋆] Instr) {sc} {sc'}
compile e =
  mcata{{t}. Code {ts} {`cons t ts}} e  with
    cmpl (VAL v)        =
      pCons (pUSH v) pNil
    cmpl (PLUS e₁ e₂)  =
      append (append (cmpl e₁) (cmpl e₂))
             (pCons aDD pNil)
    cmpl (IF e₀ e₁ e₂) =
      append (cmpl e₀)
             (pCons (iFPOP (cmpl e₁)
                           (cmpl e₂))
                    pNil)
```

Agda

```
data Inst : List Ty → List Ty → ⋆  where
  PUSH  : {t : Ty} {ts : List Ty} →
          Val t → Inst ts (t :: ts)
  ADD   : {ts : List Ty} →
          Inst (I :: I :: ts) (I :: ts)
  IFPOP : {ts ts' : List Ty} →
          Path Inst ts ts' →
          Path Inst ts ts' →
          Inst (B :: ts) ts'
Code : List Ty → List Ty → ⋆
Code sc sc' = Path Inst sc sc'
compile : {t : Ty} → {ts : List Ty} →
          Expr t → Code ts (t :: ts)
compile (VAL v)        =
  PCons (PUSH v) PNil
compile (PLUS e₁ e₂) =
  append (append (compile e₁) (compile e₂))
         (PCons ADD PNil)
compile (IF e₀ e₁ e₂) =
  append (compile e₀)
         (PCons (IFPOP (compile e₁)
                       (compile e₂))
                PNil)
```

Fig. 3: A stack-safe compiler

piled code consisting of the three instructions $\mathit{inst}_1 : \mathit{Inst}\ \{\mathit{ts}_0\}\ \{\mathit{ts}_1\}$, $\mathit{inst}_2 : \mathit{Inst}\ \{\mathit{ts}_1\}\ \{\mathit{ts}_2\}$, and $\mathit{inst}_3 : \mathit{Inst}\ \{\mathit{ts}_2\}\ \{\mathit{ts}_3\}$ has the type $\mathit{Code}\ \{\mathit{ts}_0\}\ \{\mathit{ts}_3\}$.

3 Discussion

Indexed types (e.g., *Val* in Fig. 1) are classified by kinds (e.g., $\mathit{Ty} \to \star$). What do valid kinds look like? Sorting rules define kind validity (or, well-sortedness). Different programming languages, that support term indices, have made different design choices. In this section, we compare the sorting rules of Nax with the sorting rules of other languages (Sect. 3.1). Then, we compare the class of indexed datatypes supported by Nax with those supported in other languages (Sect. 3.2).

3.1 Universes, Kinds, and Well-sortedness

The concrete syntax for kinds appears similar among Haskell, Nax, and Agda. For instance, in Fig. 1, the kind $\mathit{Ty} \to \star$ has exactly the same textual representation in all of the three languages. However, each language has its own universe structure, kind syntax, and sorting rules, as summarized in Fig. 4.

Figure 5 illustrates differences and similarities between the mechanism for checking well-sortedness, by comparing the justification for the well-sortedness of the kind $\mathit{List}\ \mathit{Ty} \to \star$. The important lessons of Fig. 5 are that the Nax approach is closely related to *universe subtyping* in Agda, and, the datatype promotion in Haskell is closely related to *universe polymorphism* in Agda.

In Nax, we may form a kind arrow $\{A\} \to \kappa$ whenever A is a type (i.e., $\vdash_{\mathsf{ty}} A : \star$). Note that types may only appear in the domain (the left-hand-side of the arrow) but not in the codomain (the right-hand-side of the arrow). Modulo right associativity of arrows (i.e., $\kappa_1 \to \kappa_2 \to \kappa_3$ means $\kappa_1 \to (\kappa_2 \to \kappa_3)$), kinds in Nax always terminate in $\star$. For example,[7]

$$\star \to \star \to \star, \qquad \{\mathit{Nat}\} \to \{\mathit{Nat}\} \to \star, \qquad (\{\mathit{Nat}\} \to \star) \to \{\mathit{Nat}\} \to \star.$$

The sorting rule ($\{\}{\to}$) could be understood as a specific use of universe subtyping ($\star \leqslant \square$) hard-wired within the arrow formation rule. Agda needs a more general notion of universe subtyping, since Agda is a dependently typed language with stratified universes, which we will shortly explain.

Agda has countably many stratified type universes for several good reasons. When we from a kind arrow $\kappa_1 \to \kappa_2$ in Agda, the domain κ_1 and the codomain κ_2 must be the same universe (or, sort), as specified by the ($\to$) rule in Fig. 4, and the arrow kind also lies in the same universe. However, requiring κ_1, κ_2, and $\kappa_1 \to \kappa_2$ to be in exactly the same universe can cause a lot of code duplication. For example, $\mathit{List}\ \mathit{Ty} \to \star_0$ cannot be justified by the ($\to$) rule since $\vdash \mathit{List}\ \mathit{Ty} : \star_0$ while $\vdash \star_0 : \star_1$. To work around the universe difference, one could define

[7] The Nax implementation allows programmers to omit curly braces in kinds when the domain of arrow kind obviously looks like a type. For instance, $\mathit{Nat} \to \star$ is considered as $\{\mathit{Nat}\} \to \star$ since *Nat* is obviously a (nullary) type constructor because it starts with an uppercase. In Sect. 2, we omitted curly braces to help readers compare Nax with other languages. From now on, we will consistently put curly braces for clarity.

HASKELL + `DataKinds`	NAX	AGDA
$\star : \Box$	$\star : \Box$	$\underset{\star}{\overset{\Vert}{\star_0}} : \underset{\Box}{\overset{\Vert}{\star_1}} : \star_2 : \star_3 : \cdots$
$\kappa ::= \star \mid \kappa \to \kappa \mid T\,\overline{\kappa}$	$\kappa ::= \star \mid \kappa \to \kappa \mid \{A\} \to \kappa$	term/type/kind/sort merged into one pseudo-term syntax
$(\to)\dfrac{\vdash_{\mathsf{k}} \kappa_1 : \Box \qquad \vdash_{\mathsf{k}} \kappa_2 : \Box}{\vdash_{\mathsf{k}} \kappa_1 \to \kappa_2 : \Box}$	$(\to)\dfrac{\vdash_{\mathsf{k}} \kappa_1 : \Box \qquad \vdash_{\mathsf{k}} \kappa_2 : \Box}{\vdash_{\mathsf{k}} \kappa_1 \to \kappa_2 : \Box}$	$(\to)\dfrac{\vdash \kappa_1 : \star_i \qquad \vdash \kappa_2 : \star_i}{\vdash \kappa_1 \to \kappa_2 : \star_i}$
$(\uparrow^{\Box}_{\star})\dfrac{\begin{array}{c}\vdash_{\mathsf{ty}} T : \star^n \to \star \\ \vdash_{\mathsf{k}} \kappa : \Box \text{ for each } \kappa \in \overline{\kappa}\end{array}}{\vdash_{\mathsf{k}} T\,\overline{\kappa} : \Box}$	$(\{\}\!\to)\dfrac{\vdash_{\mathsf{ty}} A : \star \qquad \vdash_{\mathsf{k}} \kappa : \Box}{\vdash_{\mathsf{k}} \{A\} \to \kappa : \Box}$	$(\leqslant)\dfrac{\vdash \kappa : s \qquad s \leqslant s'}{\vdash \kappa : s'}$

Fig. 4: Universes, kind syntax, and selected sorting rules of Haskell, Nax, and Agda. Haskell's and Nax's kind syntax are simplified to exclude kind polymorphism. Agda's $(\to)$ rule is simplified to only allow non-dependent kind arrows.

NAX

$$(\{\}\!\to)\dfrac{\dfrac{\vdash_{\mathsf{ty}} \mathit{List} : \star \to \star \qquad \vdash_{\mathsf{ty}} \mathit{Ty} : \star}{\vdash_{\mathsf{ty}} \mathit{List}\ \mathit{Ty} : \star} \qquad \vdash_{\mathsf{k}} \star : \Box}{\vdash_{\mathsf{k}} \{\mathit{List}\ \mathit{Ty}\} \to \star : \Box}$$

AGDA

$$(\to)\dfrac{(\leqslant)\dfrac{(\to)\dfrac{\vdash \mathit{List} : \star \to \star \qquad \vdash \mathit{Ty} : \star}{\vdash \mathit{List}\ \mathit{Ty} : \star} \qquad \star \leqslant \Box}{\vdash \mathit{List}\ \mathit{Ty} : \Box} \qquad \vdash \star : \Box}{\vdash \mathit{List}\ \mathit{Ty} \to \star : \Box}$$

HASKELL

$$(\to)\dfrac{(\uparrow^{\Box}_{\star})\dfrac{\vdash_{\mathsf{ty}} \mathit{List} : \star \to \star \qquad (\uparrow^{\Box}_{\star})\dfrac{\vdash_{\mathsf{ty}} \mathit{Ty} : \star}{\vdash_{\mathsf{k}} \mathit{Ty} : \Box}}{\vdash_{\mathsf{k}} \mathit{List}\ \mathit{Ty} : \Box} \qquad \vdash_{\mathsf{k}} \star : \Box}{\vdash_{\mathsf{k}} \mathit{List}\ \mathit{Ty} \to \star : \Box}$$

AGDA
+ universe polymorphism

$$(\to)\dfrac{\dfrac{\dfrac{\vdash \mathit{List} : \forall\{i\} \to \star_i \to \star_i}{\vdash \mathit{List} : \Box \to \Box} \qquad \dfrac{\vdash \mathit{Ty} : \forall\{i\} \to \star_i}{\vdash \mathit{Ty} : \Box}}{\vdash \mathit{List}\ \mathit{Ty} : \Box} \qquad \vdash \star : \Box}{\vdash \mathit{List}\ \mathit{Ty} \to \star : \Box}$$

Fig. 5: Justifications for well-sortedness of the kind $\mathit{List}\ \mathit{Ty} \to \star$ in Nax, Haskell, Agda

datatypes List' and Ty', which are isomorphic to List and Ty, only at one higher level, such that $\vdash \mathit{List}'\ \mathit{Ty}' : \star_1$. Only then, one can construct $\mathit{List}'\ \mathit{Ty}' \to \star_0$. Furthermore, if one needs to form $\mathit{List}\ \mathit{Ty} \to \star_1$ we would need yet another set of duplicate datatypes List'' and Ty'' at yet another higher level. Universe subtyping provides a remedy to such a code duplication problem by allowing objects in a lower universe to be considered as objects in a higher universe.

This gives us a notion of subtyping such that $\star_i \leqslant \star_j$ where $i \leqslant j$.[8] With universe subtyping, we can form arrows from *Ty* to any level of universe (e.g., $\mathit{List}\ \mathit{Ty} \to \star_0$, $\mathit{List}\ \mathit{Ty} \to \star_1$, ...). Relating Agda's universes to sorts in Haskell and Nax, $\star_0$ and $\star_1$ correspond to $\star$ and $\Box$. So, we write $\star$ and $\Box$ instead of $\star_0$ and $\star_1$ in the justification of well-formedness of $\mathit{List}\ \mathit{Ty} \to \star$ in Agda, to make the comparisons align in Fig. 5.

In addition to universe subtyping, Agda also supports universe polymorphism,[9] which is closely reated to datatype promotion. In fact, it is more intuitive to understand the datatype promotion in Haskell as a special case of universe polymorphism. Since there are only two universes $\star$ and $\Box$ in Haskell, we can think of datatypes like *List* and *Ty* are defined polymorphically at both $\star$ and $\Box$. That is, $\mathit{List} : \Box \to \Box$ as well as $\mathit{List} : \star \to \star$, and similarly, $\mathit{Ty} : \Box$ as well as $\mathit{Ty} : \star$. So, $\mathit{List} : \Box \to \Box$ can be applied to $\mathit{Ty} : \Box$ at kind level, just as $\mathit{List} : \star \to \star$ can be applied at type level.

In summary, Nax provides a new way of forming kind arrows by allowing types, which are already fully applied at the type level, as the domain of an arrow. On the contrary, Haskell first promotes type constructors (e.g., *List*) and their argument types (e.g., *Ty*) to the kind level, and everything else (application of *List* to *Ty* and kind arrow formation) happens at kind level.

3.2 Nested Term Indices and Datatypes Containing Types

Nax supports nested term indices while Haskell's datatype promotion cannot. Examples in Sect. 2 only used rather simple indexed datatypes, whose terms indices are of non-indexed types (e.g., *Nat*, *List Ty*). One can imagine more complex indexed datatypes, where some term indices are themselves of term-indexed datatypes. Such nested term indices are often useful in dependently typed programming. For instance, Brady and Hammond [3] used an environment datatype with nested term indices in their EDSL implementation for verified resource usage protocols. Figure 6 illustrates transcriptions of their environment datatype (*Env*), originally written in Idris [2], into Nax and Agda. The datatype *Env* is indexed by a length indexed list (*Vec*), which is again indexed by a natural number (n). Note that the nested term-index n appears inside the curly braces nested twice ($\{\mathit{Vec}\ \mathit{st}\ \{n\}\}$). There is no Haskell transcription for *Env* because datatype promotion is limited to datatypes without term indices.

On the contrary, Haskell supports promoted datatypes that hold types as elements, although limited to types without term indices, while Nax does not. The heterogeneous list datatype (*HList*) in Fig. 7 is a well-known example[10] that uses datatypes containing types. Note that *HList* is indexed by $\mathit{List}\ \star$, which is a promoted list whose elements are of kind $\star$, that is, element are types. For instance, *hlist* in Fig. 7 contains three elements $3 : \mathit{Int}$, $\mathit{True} : \mathit{Bool}$, and $(1 :. 2 :. \mathit{Nil}) : \mathit{List}\ \mathit{Int}$, and its type is $\mathit{HList}\ (\mathit{Int} :. \mathit{Bool} :. \mathit{List}\ \mathit{Int} :. \mathit{Nil})$.

[8] See Ulf Norell's thesis [18] (Sect. 1.4) for the full description on universe subtyping.
[9] See http://wiki.portal.chalmers.se/agda/agda.php?n=Main.UniversePolymorphism.
[10] The *HList* library in Haskell by Kiselyov et al. [13] was originally introduced using type class constraints, rather than using GADTs and other relatively new extensions.

Nax

```
-- Environments for stateful resources index by length indexed lists
data V : ⋆ → (Nat → ⋆) → Nat → ⋆ where
  VNil : V a r {`zero}
  VCons : a → r {n} → V a r {`succ n}
    deriving fixpoint Vec
data Envr : (({st} → ⋆) → {Vec st {n}} → ⋆)
          → (({st} → ⋆) → {Vec st {n}} → ⋆) where
  Empty : Envr r res {`vNil}
  Extend : res {x} → r res {xs} → Envr r res {`vCons x xs}
    deriving fixpoint Env

-- Usage example
data St = Read | Write  -- resource state
data Res : St → ⋆ where  -- resource
  File1 : Res {Read}
  File2 : Res {Write}
-- myenv : Env Res {`vCons Read (`vCons Write `vNil)}
myenv = extend File1 (extend File2 empty)

-- Environments additionaly index by singleton natural numbers
data SN : (Nat → ⋆) → (Nat → ⋆) where
  Szer : SN r {`zero}
  Ssuc : r {n} → SN r {`succ n}
    deriving fixpoint SNat
data Envr′ : (({st} → ⋆) → {SNat {n}} → {Vec st {n}} → ⋆)
           → (({st} → ⋆) → {SNat {n}} → {Vec st {n}} → ⋆) where
  Empty′ : Envr′ r res {`szer} {`vNil}
  Extend′ : res {x} → r res {n} {xs} → Envr′ r res {`ssuc n} {`vCons x xs}
    deriving fixpoint Env′
-- myenv′ : Env′ Res {`ssuc (`ssuc`szer)} {`vCons Read (`vCons Write `vNil)}
myenv′ = extend′ File1 (extend′ File2 empty′)
```

Agda

```
data Vec (a : ⋆) : ℕ → ⋆ where
  VNil   : {n : ℕ} → Vec a n
  VCons : {n : ℕ} → a → Vec a n → Vec a (suc n)
data Env {st} (res : st → ⋆) : {n : ℕ} → Vec st n → ⋆ where
  Empty  : Env res {0} VNil
  Extend : {n : ℕ} {x : st} {xs : Vec st n} →
             res x → Env res xs → Env res {suc n} (VCons x xs)
```

Fig. 6: Environments of stateful resources indexed by the length indexed list of states

HASKELL + GADTs, DataKinds, PolyKinds

```
data List a = Nil | a :. List a ;  infixr :.
data HList :: List ⋆ → ⋆ where
  HNil :: HList Nil
  HCons :: t → HList ts → HList (t :. ts)
hlist :: HList (Int :. Bool :. List Int :. Nil)
hlist = HCons 3 (HCons True (HCons (1 :. 2 :. Nil) HNil))
```

Fig. 7: Heterogeneous lists (*HList*) indexed by the list of element types (*List* ⋆).

4 Related Work

Singleton types, first coined by Hayashi [11], have been used in lightweight verification to simulate dependent types [23, 14]. Sheard et al. [21] demonstrated that singleton types can be defined just like any other datatypes in Omega [20], a language equipped with GADTs and rich kind structure. Nax's universe and kind structure is much simpler than Omega's (e.g., no user defined kinds in Nax), yet singleton types are definable with fewer worries about code duplication across different universes. Singleton types are typically indexed by the values of their non-singleton counterparts. For example, in Fig. 6, singleton natural numbers (*SNat*) are indexed by natural numbers (*Nat*). Note that we can index datatypes by singleton types in Nax, while datatype promotion cannot (recall Sect. 3.2). For instance *Env′* indexed by *SNat* in Fig. 6 is a more faithful transcription of the dependently typed version than *Env*, since *Env′* has a direct handle on size of the environment at type level, just by referring to the *SNat* index, without extra type level computation on the *Vec* index.

Eisenberg and Weirich [8], in the setting of Haskell's datatype promotion, automatically derive a singleton type (e.g., singleton natural numbers) and its associated functions (e.g., addition over singleton natural numbers) from their non-singleton counterparts (e.g., natural numbers and their addition). We think it would be possible to apply similar strategies to Nax, and even better, singleton types for already indexed datatypes would be derivable.

The kind arrow ($\{A\} \to \kappa$), from a type to a kind, predates Nax. Our kind syntax in Fig. 4, although developed independently, happens to coincide with the kind syntax of Deputy [6], a dependently typed system for low-level imperative languages with variable mutation and heap-allocated structure.

Curly braces in Nax are different from the curly braces in Agda or SHE.

In Nax, curly braces mean that things inside them are *erasable* (i.e., must still type correct without all the curly braces). Agda's curly braces mean that things in them would often be *inferable* so that programmers may omit them.

The concrete syntax for kinds in SHE[11] appears almost identical to Nax's concrete kind syntax, even using curly braces around types. However, SHE's

[11] http://personal.cis.strath.ac.uk/conor.mcbride/pub/she/

(abstract) kind syntax is virtually identical to the (abstract) kind syntax of datatype promotion, thus quite different from Nax, since $\{A\} :: \Box$ in SHE.

Kind polymorphism in Nax may be polymorphic over term-index variables ($i : A$) and type variables ($\alpha : \star$), as well as over kind variables ($\mathcal{X} : \Box$). That is, polymorphic kinds (or kind schemes) in Nax may be kind polymorphic ($\forall\mathcal{X}\,.\,\kappa$), type polymorphic ($\forall\alpha\,.\,\kappa$), term-index polymorphic ($\forall i\,.\,\kappa$), or combinations of them ($\forall\mathcal{X}\ \alpha\ i.\,\kappa$). For example, the kinds of P and *Path* in Fig. 2 are polymorphic over the type variable $\iota{:}\star$. In contrast, datatype promotion in Haskell only needs to consider polymorphic kinds ($\forall\mathcal{X}\,.\,\kappa$) quantified over kind variables ($\mathcal{X} : \Box$) since everything is already promoted to the kind level.

In Nax, kind polymorphism is limited to rank-1 since it is well-known that higher-rank kind polymorphism leads to a paradox [12]. In fact, type polymorphism in Nax is limited to rank-1 as well since the type inference is based on Hindley-Milner [17].

Concoqtion [9] is an extension of MetaOCaml with indexed types. Concoqtion share some similar design principles – Hindley–Milner-style type inference and *gradual typing by erasure* over (term) indices. Both in Nax and Concoqtion, a program using indexed types must still type check within the non-indexed sublanguage (OCaml for Concoqtion) when all indices are erased from the program. However, indices in Concoqtion differ from term indices discussed in this paper (Nax, datatype promotion, and dependently typed languages like Agda). Concoqtion indices are Coq terms rather than OCaml terms. Although this obviously leads to code duplication between the index world (Coq) and the program world (OCaml), Concoqtion enjoys practical benefits of having access to the Coq libraries for reasoning about indices. Comparison of Concoqtion and other related systems can be found in the technical report by Pasalic et al. [19].

5 Summary and Future Work

In Nax, programmers can enforce program invariants using indexed types, without excessive annotations (like functional programming languages) while enjoying logical consistency (like dependently typed proof assistants).

There are two approaches that allow term-indices without code duplication at every universe. *Universe subtyping* is independent of the number of universes. Even scaled down to two universes ($\star, \Box$), it adds no additional restrictions – term indices can appear at arbitrary depth. *Universe polymorphism* is sensitive to the number of universes. Unless there are countably infinite universes, nested term indices are restricted to depth $n - 1$ where n is the number of universes.

On the other hand, universe polymorphism can reuse datatypes at term level (*List* a where $a : \star$) at type-level to contain type elements (e.g., *List* $\star$), which is beyond universe subtyping. We envision that Nax extended with first-class datatype descriptions [7] would be able express the same concept reflected at term level, so that we would have no need for type level datatypes.

Bibliography

[1] Ahn, K.Y., Sheard, T.: A hierarchy of Mendler-style recursion combinators: Taming inductive datatypes with negative occurrences. In: ICFP '11. pp. 234–246. ACM (2011)

[2] Brady, E.: IDRIS —: systems programming meets full dependent types. In: PLPV. pp. 43–54. ACM (2011)

[3] Brady, E., Hammond, K.: Correct-by-construction concurrency: Using dependent types to verify implementations of effectful resource usage protocols. Fundam. Inform 102(2), 145–176 (2010)

[4] 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 (2002)

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

[6] Condit, J., Harren, M., Anderson, Z.R., Gay, D., Necula, G.C.: Dependent types for low-level programming. In: ESOP '07. LNCS, vol. 4421. Springer (2007)

[7] Dagand, P.E., McBride, C.: Transporting functions across ornaments. In: Proceedings of the 17th ACM SIGPLAN international conference on Functional programming. pp. 103–114. ICFP '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2364527.2364544

[8] Eisenberg, R.A., Weirich, S.: Dependently typed programming with singletons. In: Proceedings of the 2012 symposium on Haskell symposium. pp. 117–130. Haskell '12, ACM (2012)

[9] Fogarty, S., Pasalic, E., Siek, J., Taha, W.: Concoqtion: indexed types now! In: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. pp. 112–121. PEPM '07, ACM (2007)

[10] Garrigue, J., Normand, J.L.: Adding GADTs to OCaml: the direct approach. In: ML '11: Proceedings of the 2011 ACM SIGPLAN workshop on ML. ACM (2011)

[11] Hayashi, S.: Singleton, union and intersection types for program extraction. In: Theoretical Aspects of Computer Software (Sendai, Japan). pp. 701–730. No. 526 in LNCS, Springer (Sep 1991)

[12] Hurkens, A.J.C.: A simplification of Girard's paradox. In: Typed Lambda Calculus and Applications. pp. 266–278 (1995)

[13] Kiselyov, O., Lämmel, R., Schupke, K.: Strongly typed heterogeneous collections. In: Haskell 2004: Proceedings of the ACM SIGPLAN workshop on Haskell. pp. 96–107. ACM (2004)

[14] Kiselyov, O., Shan, C.: Lightweight static capabilities. Electr. Notes Theor. Comput. Sci 174(7), 79–104 (2007)

[15] Mandelbaum, Y., Stump, A.: GADTs for the OCaml masses. In: ML '09: Proceedings of the 2009 ACM SIGPLAN workshop on ML. ACM (2009)

[16] McBride, C.T.: Agda-curious?: an exploration of programming with dependent types. In: Proceedings of the 17th ACM SIGPLAN international conference on Functional programming. pp. 1–2. ICFP '12, ACM (2012)

[17] Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences 17, 348–375 (1978)

[18] Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (September 2007)

[19] Pasalic, E., Siek, J., Taha, W.: Concoqtion: Mixing dependent types and Hindley-Milner type inference. Technical report, Rice University (2006), `http://www.metaocaml.org/concoqtion/`

[20] Sheard, T.: Languages of the future. In: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. pp. 116–119. OOPSLA '04, ACM (2004)

[21] Sheard, T., Hook, J., Linger, N.: GADTs + extensible kind system = dependent programming. Techincal report, Portland State University (2005), `http://cs.pdx.edu/~sheard/`

[22] 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 (2003)

[23] Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation. pp. 249–257. PLDI '98, ACM (1998)

[24] Yorgey, B.A., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., Magalhães, J.P.: Giving Haskell a promotion. In: Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation. pp. 53–66. TLDI '12, ACM (2012)