A Simpler Encoding of Indexed Types

Yinsen Zhang
Department of Computer Science and Engineering
The Pennsylvania State University
USA
yqz5714@psu.edu

Abstract
In functional programming languages, generalized algebraic data types (GADTs) are very useful as the unnecessary pattern matching over them can be ruled out by the failure of unification of type arguments. In dependent type systems, this is usually called indexed types and it’s particularly useful as the identity type is a special case of it. However, pattern matching over indexed types is very complicated as it requires term unification in general. We study a simplified version of indexed types (called simpler indexed types) where we explicitly specify the selection process of constructors, and we discuss its expressiveness, limitations, and properties.


Keywords: inductive types, indexed types, type theory, dependent types, functional programming, generalized algebraic data types

ACM Reference Format:

1 Introduction
Correct-by-construction data structures are pleasant to work with, such as well-typed and well-scoped syntax trees [1]. A key aspect of correct-by-construction data structures is that they carry value-level information in their types. We start from two simple instances of such data structures: the finite set type and the sized vector type, which are base on the general notion of indexed types:

\[
\begin{align*}
\text{data } \text{Fin} &: \mathbb{N} \to \text{Type} \_ \_ \_ \\
\text{fzero} &: \forall n \to \text{Fin} \_ \_ \_ (\text{suc} n) \\
\text{fsuc} &: \forall n \to \text{Fin} \_ \_ \_ n \to \text{Fin} \_ \_ \_ (\text{suc} n)
\end{align*}
\]

\[
\begin{align*}
\text{data } \text{Vect} (A : \text{Type } \ell) &: \mathbb{N} \to \text{Type } \ell \_ \_ \_ \\
[\_ \_ \_] &: \text{Vect} A 0 \\
\text{\_\_\_\_} &: \forall n \to A \to \text{Vect} A n \to \text{Vect} A (\text{suc} n)
\end{align*}
\]

The indices of types are the parameters at the right-hand-side of the colons in the signatures of inductive types, which can be specialized by constructors. The two constructors of \text{Fin} specify the index as \text{suc} n, so when pattern matching over \text{Fin zero} requires no clauses. The algorithm for selecting constructors is a process of term unification, extracting a most-general-unifier and apply that to the rest of the telescope. For each constructor, we unify the type arguments with the indices it specifies, and there are three potential results [10, §2.1].

- Success positively – the constructor matches.
- Success negatively – the constructor does not matches.
- Failure – cannot decide, pattern matching cannot be performed.

Users will have to understand the error messages with unification failures, which is an accidental complexity brought into dependent type systems. The implementation of the unification algorithm also affects the selection of constructors.

We propose an alternative syntax for indexed types. First, for inductive types without indices, we use a Haskell-style syntax to describe its arguments, and we allow bindings in the parameters since we are working with dependent types:

\[
\begin{align*}
\text{data } \mathbb{N} &: \text{U} \\
\text{data } \text{List} (A : \text{U}) &: \text{U} \\
\_ \_ \_ \_ &: \text{zero} \\
\_ \_ \_ \_ \_ &: \forall n \to A \to \text{List} A n \to \text{List} A (\text{suc} n)
\end{align*}
\]

Then, we allow the constructors to perform a pattern matching over the type of the parameters. For instance, we define the sized vector type using the following syntax:

\[1\]This piece of code is written in Agda [24]. The types and functions are in blue while constructors are in green. There are other pieces of code written in Aya [3], where types are in green and constructors are in purple. In §3, we use the latter coloring.
We propose the new syntax because the term unification problem is related to the injectivity of type constructors, see the discussion in [11, §1]). Similarly the unification problem may become higher order (like in Higher suc, Higher pred, Higher (λ x → 2 + x + 3) could be pattern matched against!), generating more confusing instances.

These examples are impossible to construct with simpler indexed types. With the proposed syntax, we could avoid not only implementing such a complicated unification algorithm, but also explaining these unification failures in error messages.

1.2 Contributions
- We present the syntax (§2.1) and the type checking algorithm (§2.3) for simpler indexed types in §2.
- We discuss their limitations (§3.1), provide a translation of simpler indexed types to general indexed types (§3.2), and discuss the compilation of simpler indexed types (§3.3) in §3.
- We explore potential extension to simpler indexed types (§4.1) and compare it with similar work (§4.2) in §4.

2 Formalization of Simpler Indexed Types
In this section, we describe the core language syntax and the type checking of simpler indexed types. The coverage checking can be adapted from any other dependent type systems with indexed types by replacing the term unification with pattern matching, so we assume the existence of a suitable coverage check.

2.1 Core Language Syntax
The syntax of terms is presented in Fig. 1. It has spine-normal fully-applied applications on “definitions” (including types D, constructors c, and functions definitions f). Normal constructs such as λ-abstraction and the Π-type are also available. ⇒ is used in λ-abstractions instead of dots for consistency with function definitions and pattern matching clauses. We use _DD to denote a list of expressions, and ∅ when the list is empty.

| x, y ::= | variable names |
| A, B, u, v ::= | f _DD full application to functions |
|                | x _DD application to references |
|                | D _DD fully applied inductive type |
|                | c _DD fully applied constructor |
|                | (x : A) → B II-type |
|                | λx ⇒ u lambda abstraction |

Figure 1. Syntax of terms

Case-split expressions can be encoded as functions and can be easily added to our type theory, but they are unrelated to simpler indexed types. Therefore, we omit them. We will have two syntactic sugars for the Π-type: A → B for (x : A) → B, and Δ → B for (x₁ : A₁) → (x₂ : A₂) → ··· → (xₙ : Aₙ) → B where Δ = (x₁ : A₁)ᵢₑᵢ[₁,ₙ]. The latter is only used in §3.2.

1.1 Problematic Indexed Types
We propose the new syntax because the term unification problem generated by general indexed types could be very complicated. Here are some examples of indexed types in Agda that generate such unification problems:

| data Vec (A : _U_) (n : N) : _U_ |
| A, zero ⇒ vnil |
| A, suc n ⇒ vcons (x : A) (xs : Vec A n) |

This pattern matching is not a traditional pattern matching, say, it does not need to be covering (although in the Vec example it is) and it can contain seemingly unreachable patterns (like duplicated patterns). Instead, they represent the selection process of constructors directly. The type checking of pattern matching consists of two steps: the well-typedness checking. Similarly the unification problem may become higher order (like in Higher suc, Higher pred, Higher (λ x → 2 + x + 3) could be pattern matched against!), generating more confusing instances.

However, pattern matching is a basic construct in dependent type systems, and it is decidable and terminating – unlike the general term unification problem, where we normally give up higher-order cases to avoid undecidability. It is also more friendly to general users because they are required to understand one concept less.

Another example is the finite set type:

| data Fin (n : N) : _U_ |
| suc n ⇒ fzero |
| suc n ⇒ fsuc (x : Fin n) |

Another example is the finite set type:

| data Univ : Type₀ → Type₀ where |
| univ : ∀ u → Univ (u → N) |
| data Higher : (N → N) → Type₀ where |
| higher-suc : Higher suc |
| higher-pred : Higher pred |
| higher-misc : Higher (λ x → 2 + x + 3) |

We cannot perform pattern matching on the type Univ Bool since Agda cannot unify Bool and N (this unification problem is related to the injectivity of type constructors, see the discussion in [11, §1]). Similarly the unification problem may become higher order (like in Higher – neither Higher suc, Higher pred, Higher (λ x → 2 + x + 3) could be pattern matched against!), generating more confusing instances.

1.1 Problematic Indexed Types

We use

\[
\text{data Vec}(A : U)(n : N) : U
\]

| A, zero ⇒ vnil |
| A, suc n ⇒ vcons (x : A) (xs : Vec A n) |

These examples are impossible to construct with simpler indexed types. With the proposed syntax, we could avoid not only implementing such a complicated unification algorithm, but also explaining these unification failures in error messages.

1.2 Contributions

- We present the syntax (§2.1) and the type checking algorithm (§2.3) for simpler indexed types in §2.
- We discuss their limitations (§3.1), provide a translation of simpler indexed types to general indexed types (§3.2), and discuss the compilation of simpler indexed types (§3.3) in §3.
- We explore potential extension to simpler indexed types (§4.1) and compare it with similar work (§4.2) in §4.
The syntax for definitions, contexts and signatures is defined in Fig. 2. A signature is a list of declarations and a context is a list of bindings. Constructors are with or without a list of patterns. The variables in the same pattern are assumed to be distinct.

\[\Gamma, \Delta, \Theta ::= x_j : A_j \quad \text{context}\]
\[\text{decl ::= data } D \Delta \text{ cons}\quad \text{simpler indexed type}\]
\[| \text{func } f \Delta : A \text{ cls}\quad \text{function definition}\]
\[\text{cons ::= } | \overline{p} \Rightarrow c \Delta \quad \text{pattern matching constructor}\]
\[| c \Delta \quad \text{constructor}\]
\[\text{cls ::= } | \overline{p} \Rightarrow u \quad \text{pattern matching clause}\]
\[p, q ::= c \overline{p} \quad \text{constructor patterns}\]
\[| x \quad \text{catch-all patterns}\]
\[| \text{impossible} \quad \text{absurd patterns}\]
\[\Sigma ::= \text{decl}\quad \text{signature}\]

**Figure 2. Syntax of signature and declarations**

We will borrow some notational convention from [9, §3.2]²; \[u[x/x]\] for substituting occurrences of \(x\) with \(v\) in term \(u\).

We use \[u[\overline{x}/\overline{y}]\] to denote a list of substitutions applied sequentially to the term \(u\). Substitution objects are denoted as \(\sigma\). We will assume the substitution operation defined on terms, patterns, and substitutions.

In the typing rules in §2.3, we will omit the vertical bars in \textit{cons} and \textit{cls} which are intended to separate the clauses and constructors.

### 2.2 Operations on Terms

We also need some operations on terms and patterns. All of them are defined by induction on the syntax. We define \textit{vars}(\Delta) to compute the list of variables in \(\Delta\):

\[\text{vars}(\emptyset) := \emptyset\]
\[\text{vars}(x : A, \Delta) := x, \text{vars}(\Delta)\]

We define \textit{vars}(\(p\)) to compute the list of bindings in pattern \(p\) and \textit{vars}(\(\overline{p}\)) to gather all the bindings in the patterns \(\overline{p}\). This operation requires the well-typedness of the patterns because we need the types of the bindings. We store these types into the patterns to allow accessing them in this operation:

\[\text{vars}(x : A) := x : A\]
\[\text{vars}(\text{impossible}) := \emptyset\]
\[\text{vars}(c \overline{p}) := \text{vars}(\overline{p})\]
\[\text{vars}(\emptyset) := \emptyset\]
\[\text{vars}(x : A, \overline{p}) := x : A, \text{vars}(\overline{p})\]

We define \textit{term}(\(p\)) to compute a term that matches exactly the pattern \(p\) and \textit{term}(\(\overline{p}\)) to compute a list of terms matching exactly the patterns \(\overline{p}\). This requires \(p\) to contain no impossible sub-patterns:

\[\text{term}(x : A) := x\]
\[\text{term}(c \overline{p}) := c \text{ term}(\overline{p})\]
\[\text{term}(\emptyset) := \emptyset\]
\[\text{term}(q, \overline{p}) := \text{term}(q), \text{term}(\overline{p})\]

We define \textit{matches}(\(u, p\)) \(\mapsto \sigma\) to perform pattern matching, similar to the \textsc{match} and \textsc{matches} operations in [19]. It computes a substitution when the pattern matching success positively and produces \(\perp\) when the pattern matching success negatively. We will also define a version of this operation to match a list of terms with a list of patterns \(\text{matches}(\overline{u}, \overline{p}) \mapsto \sigma\), similar to \textsc{vars}(\(\overline{p}\)) and \textit{term}(\(\overline{p}\)).

\[\text{matches}(u, x) \mapsto [u/x]\]
\[\text{matches}(\emptyset, \emptyset) \mapsto []\]
\[\text{matches}(\overline{u}, \overline{p}) \mapsto \sigma\]
\[\text{matches}(c \overline{u}, c \overline{p}) \mapsto \sigma\]
\[\text{matches}(\overline{u}, \overline{p}) \mapsto \perp\]
\[\text{matches}(\overline{c_1 u}, c_2 \overline{p}) \mapsto \perp\]
\[\text{matches}(\overline{c_1 u}, c_2 \overline{p}) \mapsto \perp\]
\[\text{matches}(\overline{c_1 u}, \overline{c_2 p}) \mapsto \perp\]
\[\text{matches}(\overline{c_1 u}, \overline{c_2 p}) \mapsto \perp\]
\[\text{matches}(\overline{c_1 u}, \overline{c_2 p}) \mapsto \perp\]
\[\text{matches}(\overline{c_1 u}, \overline{c_2 p}) \mapsto \perp\]
\[\text{matches}(\overline{c_1 u}, \overline{c_2 p}) \mapsto \perp\]
\[\text{matches}(\overline{c_1 u}, \overline{c_2 p}) \mapsto \perp\]
\[\text{matches}(\overline{c_1 u}, \overline{c_2 p}) \mapsto \perp\]
\[\text{matches}(\overline{c_1 u}, \overline{c_2 p}) \mapsto \perp\]

**Figure 3. Pattern matching operation**

**Lemma 2.1.** For all pattern \(p\), \textit{matches}(\textit{term}(\(p\)), \(p\)) \(\mapsto \sigma\) and for all list of patterns \(\overline{p}\), \textit{matches}(\textit{term}(\(\overline{p}\)), \(\overline{p}\)) \(\mapsto \sigma\). In both formulae, the substitution \(\sigma\) is an identity substitution.

**Proof.** By induction on \(p\). \(\square\)

### 2.3 Typing Rules for Terms

Well-typed terms are formed under the following type checking judgments:

- \[\Sigma, \Gamma \vdash \Delta\] \(\Delta\) is a well-formed context under \(\Sigma, \Gamma\).
- \[\Sigma, \Gamma \vdash u : A\] term \(u\) has type \(A\) under \(\Sigma, \Gamma\).
- \[\Sigma, \Gamma \vdash \overline{u} : \Delta\] terms \(\overline{u}\) instantiate context \(\Delta\) under \(\Sigma, \Gamma\).
- \[\Sigma, \Gamma \vdash u = v : A\] terms \(u\) and \(v\) are equal inhabitants of type \(A\) under \(\Sigma, \Gamma\).

Typing rules for types and terms are defined in Fig. 4. They are grouped by the relevant type formation. For simplicity, we will omit several things:

²Other styles of substitution include \(u[x \mapsto v]\), \(u[x/v]\), etc. (there is a relevant online discussion [21])
• We assume the conversion check between terms – the
  problem is the same as other dependent type systems,
  and the strategies used by other systems will apply to
  ours as well.
• We also have the type-in-type rule to simplify the
  universe types, and in practical implementations, we
  could integrate polymorphic universe levels to make
  the system consistent.
• In the implementation of Aya and Arend, we also have
  the sigma type and records, but we omit them here for
  simplicity.

Rules related to the \( \Pi \)-type.

\[
\frac{\Sigma; \Gamma \vdash A : \mathcal{U}}{\Sigma; \Gamma \vdash (x : A) \rightarrow B : \mathcal{U}}
\]

\[
\text{func } \Delta : A \stackrel{\text{rules}}{\vdash} \Sigma; \Gamma \vdash \bar{\sigma} : \Delta
\]

\[
\frac{\Sigma; \Gamma \vdash \bar{\sigma} : \Delta[\bar{\theta}]}{\Sigma; \Gamma \vdash \alpha[\bar{\theta}]} \quad (\text{matches}(\bar{\sigma}, \bar{\theta}) \mapsto \sigma)
\]

\[
\frac{\Delta \vdash \bar{\sigma} \in \Sigma}{\Sigma; \Gamma \vdash \bar{\sigma} : \Delta[\bar{\theta}]} \quad \text{IXCall}
\]

\[
\frac{\Sigma; \Gamma \vdash \bar{\sigma} \in \Sigma}{\Sigma; \Gamma \vdash \bar{\sigma} : \Delta[\bar{\theta}]} \quad \text{CONCall}
\]

Figure 4. Typing rules for types and terms

In the IXCall rule, we perform a pattern matching
between the type arguments and the patterns in the construc-
tor to make sure the availability of the selected constructor,
and apply the resulting substitution to the parameters of
the constructor as they can access the patterns according to
the rules in Fig. 5. In contrast, the CONCall rule does not
perform pattern matching and the constructor is directly
available. The differences between IXCall and CONCall
include a successful matches \((u, p)\) operation and an extra
substitution applied on \(\Delta\).

2.4 Signature Well-formedness

A well-formed signature consists of a list of well-typed declara-
tions. We can think of the whole type checking algorithm
as a signature formation process.

To check function definitions and pattern matching con-
structors, we first need to type check the patterns and elabo-
rate the pattern matching clauses. The rules of pattern type
checking, similar to the operations in §2.2, have two versions:

\[
\Sigma; \Gamma \vdash p : A \mapsto \Theta \quad \text{type-checking a pattern } p \text{ against a type } A.
\]

\[
\Sigma; \Gamma \vdash \bar{p} : \Delta \mapsto \Theta \quad \text{type-checking patterns } \bar{p} \text{ against a context } \Delta.
\]

These rules are defined in Fig. 5. They produce a context
\(\Theta\) containing all of the bindings in the give pattern(s).

Rules for one pattern.

\[
\Sigma; \Gamma \vdash x : A \mapsto x : A
\]

\[
\text{data } \Delta \overline{c \sigma} \in \Sigma \quad \text{data } \Delta \overline{c \sigma} \in \Sigma \quad \Sigma; \Gamma \vdash \bar{\sigma} : \Delta \mapsto \Theta
\]

\[
\Sigma; \Gamma \vdash \bar{p} : D \bar{u} \mapsto \Theta
\]

Rules for a list of patterns.

\[
\Sigma; \Gamma \vdash q : A \mapsto \Theta \quad \Sigma; \Gamma \vdash \bar{p} : \Delta[\text{term}(q)/x] \mapsto \Theta'
\]

\[
\Sigma; \Gamma \vdash \emptyset : \emptyset \mapsto \emptyset
\]

Figure 5. Type checking of patterns

Lemma 2.2. \(\Sigma; \Gamma \vdash \Delta \mapsto \Theta \implies \Sigma; \Gamma \vdash \text{term}(\bar{p}) : \Delta \text{ and } \Sigma; \Gamma \vdash p : A \mapsto \Theta \implies \Sigma; \Gamma \vdash \text{term}(p) : A.

Proof. By induction on \(p\).  \(\square\)

Then, we define the rules for type checking pattern match-
ing structures as in Fig. 6 using the operation defined in Fig. 5.

With them, we could define the type checking of function
definitions and simpler indexed types, and form signature by
the rules \(\Sigma; \Gamma \vdash \bar{p}\) in Fig. 7. The declarations are checked one
after another so latter functions can depend on former ones.
A Simpler Encoding of Indexed Types

By that, we will lose induction-recursion [17] and induction-induction [18], and we consider it a potential future work.

3 Metatheory of Simpler Indexed Types

In this section, we discuss the limitations of simpler indexed types by examples and provide a complete and sound translation from simpler indexed types to general indexed types.

3.1 Limitations and Workarounds

Many useful indexed types cannot be written as simpler indexed types. For instance, as a common illustration of the convenience brought by indexed types, the normalizer and the syntax tree for an expression language with natural numbers and booleans can be defined as a general indexed type in Agda [24].

The syntax looks like this. We first define the type \textsf{Term} as a type indexed by another type, which is the type of the evaluation result of each syntax variant. In each constructor, we specialize this type.

\begin{verbatim}
data Term : Type₀ → Type₀ where
  nat : ℕ → Term ℕ
  succ : Term ℕ → Term ℕ
  bool : Bool → Term Bool
  inv : Term Bool → Term Bool
  case : Term Bool → (x y : Term A) → Term A

Then, we define its \textsf{normalize} function, which takes an instance of \textsf{Term A} and return an instance of type \textsf{A}. The type guarantees that there will never be ill-typed terms like \textsf{succ (bool x)}.

\textsf{normalize} : Term A → A
\end{verbatim}

We cannot encode \textsf{Term} as a simpler indexed type because we cannot pattern match on types, so the direct translation will not work – We will need an auxiliary type to help us encoding them:

\begin{verbatim}
data TermTy : \mathcal{U} | \textit{natT} | \textit{boolT}
  func TermTy (x : TermTy) : \mathcal{U}
  | \textit{natT} ⇒ \textit{ℕ}
  | \textit{boolT} ⇒ \textit{Bool}
\end{verbatim}

Then, we define the type for terms and the normalize function:

\begin{verbatim}
data Term (n : TermTy) : \mathcal{U}
  | \textit{natT} ⇒ \textit{ℕ}
  | \textit{natT} ⇒ \textit{succ} (Term \textit{natT})
  | \textit{boolT} ⇒ \textit{bool} \textit{boolT}
  | \textit{boolT} ⇒ \textit{inv} (Term \textit{boolT})
  | A ⇒ case (Term \textit{boolT}) (Term A) (Term A)
  func normalize (t : TermTy) (x : Term t) : \textit{termTy} t
  | \textit{natT}, \textit{nat} n ⇒ n
  | \textit{natT}, \textit{succ} n ⇒ suc (normalize \textit{natT} n)
  | \textit{boolT}, \textit{bool} b ⇒ b
  | \textit{boolT}, \textit{inv} b ⇒ \textit{not} (normalize \textit{boolT} b)
  | t, case b x y ⇒ ifElse (normalize \textit{boolT} b)
    (normalize t x) (normalize t y)
\end{verbatim}

In the general case, only when the indices are in canonical constructor form (say, generated by references to parameters of the constructor and applications to constructors) can we translate them into simpler index types. Even though we could use auxiliary types to help us encoding them, there is still one case where this encoding will fail, where the indices contain references to the \textit{parameters} of the indexed type. The simplest case is the identity type:

\begin{verbatim}
data Id (A : Type) (x : A) : A → Type \textit{\ell} where
  idp : Id A x x
\end{verbatim}

The index being x, a reference to the parameter of \textit{Id}, is the essential reason why a general term unification needs to be performed during the pattern matching over \textit{idp}. Pattern matching is a mechanism to match terms by patterns, not by terms.

Simpler indexed type essentially simplifies the problem of constructor selection just by turning the term-match-term problem to a term-match-pattern problem, which rules out numerous complication but also loses the benefit of general indexed types. A potential way to bring general indexed
types back without introducing them directly is discussed as future work in §4.1, requiring the presence of a built-in identity type.

3.2 Translation to Indexed Types

We could translate simpler indexed types back to general indexed types. To describe the translation, we define the syntax of general indexed types, which is the output of the translation, in Fig. 8. We do not have type parameters as they are just special cases of indices.

Now we can start the translation. First, we unify pattern matching constructors with simple constructors. For constructor \( c_\Delta \), in simpler indexed type \( D_\Delta \), we translate it into a pattern matching constructor \( \text{vars}(\Delta) \Rightarrow c_\Delta \).

After that, we could perform the translation of constructors. In other words, we need to construct the type ("\( A \)" in Fig. 8) of the translated constructor.

Definition 3.1. For pattern matching constructor \( \bar{p} \Rightarrow c_\Delta \), we construct the type of the translated constructor as \( \text{vars}(\bar{p}) \Delta_\rightarrow \rightarrow \cdot \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \right
not all (exceptions include the identity type in Agda [24] and the examples in §1.1).

4.1 Future Work

We could translate simpler indexed types into an even simpler type theory with only products and coproducts, just like in [12]. This translation requires an algorithm to classify the pattern matching clauses with overlapping parts. This is done in [13], but in Aya we have a better implementation. We decide to describe such translation after the overlapping pattern classification algorithm is formalized.

We could also have a built-in identity type in the type theory and encode the indexed types with the identity type. The Image type in [12] is a great example:

\[
\text{data Image } (A : \text{Type } \ell) (f : A \to B) : B \to \text{Type } \ell \text{ where image} \forall x \to \text{Image } A \, f \, f \, x
\]

It can be encoded as an inductive type without indices:

\[
\text{data Image } (A : \mathcal{U}) (f : A \to B) (b : B) : \mathcal{U} \mid \text{image } (x : A) (p : f \, x = b)
\]

We might be able to define a translation from general indexed types into simpler indexed types with a built-in identity type, and during pattern matching over the encoded types, we perform a rewriting over the identity proofs that we used to encode the indices. By that, we will have a different treatment of the index unification problem, and we could study how it compares to the general indexed types.

This idea (encoding the unification of type indices as a rewriting performed during pattern matching) is similar to the transpX operation discussed in [28, §3.2.4, §4] and the “index-fixing” fcoe operation discussed in [6, §4.2], but we are working in a general type theory with any definition of the identity type as long as they support the J operation, including the path type in homotopy type theory [26], the path type in cubical type theories [6, 14, 27], the identity type in intuitionistic type theory [23] (either homogeneous or heterogeneous), and others.

The cubical path type is a preferred choice as it does not depend on any fancy unification mechanism. This means we can develop a type theory expressive enough to discuss indexed types without dependent pattern matching. Apart from that, we could seek integration with induction-recursion [17] and induction-induction [18] as mentioned in §2.4.

The compilation technique could be investigated to address the limitation discussed in §3.3.

4.2 Related Work

Type families in dependent types can be regarded as an encoding of GADTs [15]. This idea was then put into a simpler type system (H-M) in [2], and was developed further as first-class phantom types in [30] and guarded recursive type constructors in [8], [25] used Leibniz-style encoding of equality to reason over the equality among types for building well-typed and well-scoped syntax trees. GADTs are integrated into GHC Haskell in [29].

Indexed types [31] are the generalization of inductive types with type-equality, where values are also allowed to appear as parameters of inductive types. Agda [24] and Idris [4] have a more ergonomic design of indexed types where the equality relations are made implicit.

In [12], the type-family encoding of the sized vector type is discussed and is directly related to simpler indexed types. However, there are several key advantages of simpler indexed types over the record encoding given in [12]:

- Simpler indexed types have names for the types and constructors. The record encoding anonymizes the type and the constructors, so the error messages are harder to understand.
- The pattern matching in simpler indexed type does not need to be covering. For instance, the simpler indexed type Fin zero is implicitly an empty type, while encoding it as a function requires writing an explicit pattern matching clause zero \(_\bot_\).
- Similar to coverage, pattern matching in simpler indexed types does not need to be structurally recursive. The record encoding uses functions so we need to respect the rules for functions, including persuading the termination checker.

Another work related to the encoding of indexed types is [7, §5], where they propose an encoding similar to the image example proposed in §4.1 and discuss a potential optimization to indexed types similar to [12]. The advantages of simpler indexed types over [12] still applies to the encoding in [7]. A notable application of indexed types based on [7] is ornaments [16, 22].

The proposed feature has been implemented in two systems individually:

- The Arend [20] proof assistant, an implementation of homotopy type theory with a cubical-flavored interval type.
- The Aya [3] proof assistant, an experimental implementation of a type theory similar to Arend’s, but with other features such as overlapping and order-independent patterns [13].

All of the operations (except vars(Δ) – it is too simple to be a class) in §2.2 have a corresponding class in the package org.aya.core.pat in the source code of Aya: vars(p) corresponds to PatTyper, term(p) corresponds to PatTerm, matches(u, p) corresponds to PatMatcher.

Apart from that, the type checking of terms in Fig. 4 corresponds to ExprTycker, the type checking of patterns in Fig. 5.
corresponds to PatTycker, and the type checking of declarations in Fig. 2 corresponds to StmtTycker. The source code of Aya could be retrieved from the link in the corresponding reference entry. The complete normalizer example in §3.1 is available at https://github.com/aya-prover/aya-dev/blob/main/base/src/test/resources/success/type-safe-norm.aya as a test-case of the Aya type checker.

Acknowledgments

We are grateful to Yu Zhang and Yiming Zhu for their suggestions on the draft version of this paper and Guillaume Allais for their review on an earlier version of this paper. We would also like to thank the anonymous reviewers from the ICFP TyDe workshop for their valuable suggestions on improving the paper.

References


