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.

CCS Concepts • **Software and its engineering → Language features**;

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

ACM Reference Format:
Yinsen Zhang. 2021. A Simpler Encoding of Indexed Types. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe '21), August 22, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3471875.3472991

1 Introduction

Correct-by-construction data structures are pleasant to work with, such as well-typed and well-scoped syntax trees [Allais et al. 2017]. 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*[1]:

$$
\begin{array}{l}
\mathsf{data}\ \mathsf{Fin} : \mathbb{N} \to \mathsf{Type}_0\ \mathsf{where} \\
\quad \mathsf{fzero} : \forall\, \{n\} \to \mathsf{Fin}\ (\mathsf{suc}\ n) \\
\quad \mathsf{fsuc}\ \ : \forall\, \{n\} \to \mathsf{Fin}\ n \to \mathsf{Fin}\ (\mathsf{suc}\ n)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{data}\ \mathsf{Vect}\ (A : \mathsf{Type}\ \ell) : \mathbb{N} \to \mathsf{Type}\ \ell\ \mathsf{where} \\
\quad [\,]\ \ \ \ : \mathsf{Vect}\ A\ 0 \\
\quad \_{::}\_ : \forall\, \{n\} \to A \to \mathsf{Vect}\ A\ n \to \mathsf{Vect}\ A\ (\mathsf{suc}\ n)
\end{array}
$$

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 Fin specify the *index* as suc *n*, so when pattern matching over 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 [Cockx et al. 2014a, §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:

$$
\begin{array}{ll}
\mathsf{data}\ \mathbb{N} : \mathcal{U} & \qquad \mathsf{data}\ \mathsf{List}\ (A : \mathcal{U}) : \mathcal{U} \\
\mid\ \mathsf{zero} & \qquad \mid\ \mathsf{nil} \\
\mid\ \mathsf{suc}\ (x : \mathbb{N}) & \qquad \mid\ \mathsf{cons}\ (x : A)\ (xs : \mathsf{List}\ A)
\end{array}
$$

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 [Norell 2009]. The types and functions are in blue while constructors are in green. There are other pieces of code written in Aya [Aya developers 2021], where types are in green and constructors are in purple. In §2, we use the latter coloring.

TyDe '21, August 22, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8616-6/21/08...$15.00
https://doi.org/10.1145/3471875.3472991

$$\begin{aligned}&\mathsf{data}\ \mathsf{Vec}\ (A : \mathcal{U})\ (n : \mathbb{N}) : \mathcal{U}\\ &\mid\ A, \mathsf{zero}\ \Rightarrow \mathsf{vnil}\\ &\mid\ A, \mathsf{suc}\ n\ \Rightarrow \mathsf{vcons}\ (x : A)\ (xs : \mathsf{Vec}\ A\ n)\end{aligned}$$

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 of patterns and the exhaustiveness of the patterns. We exemplify the type checking of our encoding of indexed types by describing the pattern matching over $\mathsf{Vec}\ \mathbb{N}\ n$. First, it tries to match the terms $\mathbb{N}, n$ with the patterns A, zero and $A, \mathsf{suc}\ n$. The pattern matching has three potential results, similar to the term unification problem:

- Success positively – the patterns are matched, this constructor will be available (needs to be matched).
- Success negatively – the patterns do not match, this constructor is not available (does not need to be matched).
- Failure – the pattern matching gets stuck, pattern matching cannot be performed.

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:

$$\begin{aligned}&\mathsf{data}\ \mathsf{Fin}\ (n : \mathbb{N}) : \mathcal{U}\\ &\mid\ \mathsf{suc}\ n\ \Rightarrow \mathsf{fzero}\\ &\mid\ \mathsf{suc}\ n\ \Rightarrow \mathsf{fsuc}\ (x : \mathsf{Fin}\ n)\end{aligned}$$

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 Univ : Type₀ → Type₁ 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 $\mathbb{N}$ (this unification problem is related to the injectivity of type constructors, see the discussion in [Cockx et al. 2016, §1]). Similarly the unification problem may become higher order (like in Higher – neither Higher suc, Higher pred, Higher ($\lambda\ x\ \to\ 2 + x$ - 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 Figure 2.1. 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. $\Rightarrow$ is used in λ-abstractions instead of dots for consistency with function definitions and pattern matching clauses. We use $\overline{u}$ to denote a list of expressions, and $\emptyset$ when the list is empty.

x, y	::=		variable names
A, B, u, v	::=	$\mathsf{f}\ \overline{u}$	full application to functions
	$\mid$	$x\ \overline{u}$	application to references
	$\mid$	$\mathsf{D}\ \overline{u}$	fully applied inductive type
	$\mid$	$\mathsf{c}\ \overline{u}$	fully applied constructor
	$\mid$	$(x : A) \to B$	Π-type
	$\mid$	$\lambda x \Rightarrow u$	lambda abstraction

Figure 2.1. 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: $A \to B$ for $(x : A) \to B$, and $\Delta \to B$ for $(x_1 : A_1) \to (x_2 : A_2) \to \cdots \to (x_n :$

$A_n) \to B$ where $\Delta = (\overline{x_i : A_i})_{\forall i \in [1,n]}$. The latter is only used in §3.2.

The syntax for definitions, contexts and signatures is defined in Figure 2.2. 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.

Γ, Δ, Θ	$::=$	$\overline{x_i : A_i}$	context
$decl$	$::=$	$\mathsf{data}\ \mathsf{D}\ \Delta\ \overline{cons}$	simpler indexed type
	$\mid$	$\mathsf{func}\ \mathsf{f}\ \Delta : A\ \overline{cls}$	function definition
$cons$	$::=$	$\mid \overline{p} \Rightarrow \mathsf{c}\ \Delta$	pattern matching constructor
	$\mid$	$\mid \mathsf{c}\ \Delta$	constructor
cls	$::=$	$\mid \overline{p} \Rightarrow u$	pattern matching clause
p, q	$::=$	$\mathsf{c}\ \overline{p}$	constructor patterns
	$\mid$	x	catch-all patterns
	$\mid$	$\mathsf{impossible}$	absurd patterns
Σ	$::=$	$\overline{decl}$	signature

Figure 2.2. Syntax of signature and declarations

We will borrow some notational convention from [Cockx and Abel 2018, §3.2][2]: $\boxed{u[v/x]}$ for substituting occurrences of x with v in term u. We use $\boxed{u[\overline{v}/\overline{x}]}$ to denote a list of substitutions applied sequentially to the term u. Substitution objects are denoted as σ. 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 *cons* and *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 $\mathsf{vars}(\Delta)$ to compute the list of variables in Δ:

$$\begin{aligned}\mathsf{vars}(\emptyset) &:= \emptyset \\ \mathsf{vars}(x : A, \Delta) &:= x, \mathsf{vars}(\Delta)\end{aligned}$$

We define $\mathsf{vars}(p)$ to compute the list of bindings in pattern p and $\mathsf{vars}(\overline{p})$ 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:

$$\begin{aligned}\mathsf{vars}(x : A) &:= x : A \\ \mathsf{vars}(\mathsf{impossible}) &:= \emptyset \\ \mathsf{vars}(\mathsf{c}\ \overline{p}) &:= \mathsf{vars}(\overline{p}) \\ \mathsf{vars}(\emptyset) &:= \emptyset \\ \mathsf{vars}(x : A, \overline{p}) &:= x : A, \mathsf{vars}(\overline{p})\end{aligned}$$

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

$$\begin{aligned}\mathsf{term}(x : A) &:= x \\ \mathsf{term}(\mathsf{c}\ \overline{p}) &:= \mathsf{c}\ \mathsf{term}(\overline{p}) \\ \mathsf{term}(\emptyset) &:= \emptyset \\ \mathsf{term}(q, \overline{p}) &:= \mathsf{term}(q), \mathsf{term}(\overline{p})\end{aligned}$$

We define $\mathsf{matches}(u, p) \mapsto \sigma$ to perform pattern matching, similar to the MATCH and MATCHES operations in [Goguen et al. 2006]. It computes a substitution when the pattern matching success positively and produces $\bot$ 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 $\mathsf{matches}(\overline{u}, \overline{p}) \mapsto \sigma$, similar to $\mathsf{vars}(\overline{p})$ and $\mathsf{term}(\overline{p})$.

$$\mathsf{matches}(u, x) \mapsto [u/x] \qquad \mathsf{matches}(\emptyset, \emptyset) \mapsto []$$

$$\frac{\mathsf{matches}(\overline{u}, \overline{p}) \mapsto \sigma}{\mathsf{matches}(\mathsf{c}\ \overline{u}, \mathsf{c}\ \overline{p}) \mapsto \sigma} \qquad \frac{\mathsf{matches}(\overline{u}, \overline{p}) \mapsto \bot}{\mathsf{matches}(\mathsf{c}\ \overline{u}, \mathsf{c}\ \overline{p}) \mapsto \bot}$$

$$\frac{\mathsf{c}_1 \neq \mathsf{c}_2}{\mathsf{matches}(\mathsf{c}_1\ \overline{u}, \mathsf{c}_2\ \overline{p}) \mapsto \bot} \qquad \frac{\mathsf{matches}(\overline{u}, \overline{p}) \mapsto \bot}{\mathsf{matches}((v, \overline{u}), (q, \overline{p})) \mapsto \bot}$$

$$\frac{\mathsf{matches}(v, q) \mapsto \bot}{\mathsf{matches}((v, \overline{u}), (q, \overline{p})) \mapsto \bot}$$

$$\frac{\mathsf{matches}(\overline{u}, \overline{p}) \mapsto \sigma \qquad \mathsf{matches}(v, q) \mapsto \sigma'}{\mathsf{matches}((v, \overline{u}), (q, \overline{p})) \mapsto \sigma \uplus \sigma'}$$

Figure 2.3. Pattern matching operation

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

Proof. By induction on p. ◻

2.3 Typing rules for terms

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

- $\boxed{\Sigma; \Gamma \vdash \Delta}$ Δ is a well-formed context under $\Sigma; \Gamma$.

[2] Other styles of substitution include $u[x \mapsto v]$, $u[x/v]$, etc. (there is a relevant online discussion [(https://mathoverflow.net/users/7507/kaveh) 2018])

- $\boxed{\Sigma; \Gamma \vdash u : A}$ term u has type A under $\Sigma; \Gamma$.
- $\boxed{\Sigma; \Gamma \vdash \overline{u} : \Delta}$ terms $\overline{u}$ instantiate context Δ under $\Sigma; \Gamma$.
- $\boxed{\Sigma; \Gamma \vdash u = v : A}$ terms u and v are equal inhabitants of type A under $\Sigma; \Gamma$.

Typing rules for types and terms are defined in Figure 2.4. They are grouped by the relevant type formation.

For simplicity, we will omit several things:

- 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 Π-type.

$$\frac{\Sigma; \Gamma \vdash A : \mathcal{U} \qquad \Sigma; \Gamma, x : A \vdash B : \mathcal{U}}{\Sigma; \Gamma \vdash (x : A) \to B : \mathcal{U}}$$

$$\frac{\mathsf{func}\ \mathsf{f}\ \Delta : A\ \overline{cls} \in \Sigma \qquad \Sigma; \Gamma \vdash \overline{v} : \Delta}{\Sigma; \Gamma \vdash \mathsf{f}\ \overline{v} : A[\overline{v}/\mathsf{vars}(\Delta)]}$$

$$\frac{\Sigma; \Gamma, x : A \vdash b : B[x/y]}{\Sigma; \Gamma \vdash \lambda x \Rightarrow b : (y : A) \to B}$$

$$\frac{\Sigma; \Gamma \vdash u : (x : A) \to B \qquad \Sigma; \Gamma \vdash v : A}{\Sigma; \Gamma \vdash u\ v : B[v/x]}$$

Rules related to indexed types.

$$\frac{\mathsf{data}\ \mathsf{D}\ \Delta\ \overline{cons} \in \Sigma \qquad \Sigma; \Gamma \vdash \overline{u} : \Delta}{\Sigma; \Gamma \vdash \mathsf{D}\ \overline{u} : \mathcal{U}}$$

$$\frac{\begin{array}{cc}\mathsf{data}\ \mathsf{D}\ \Delta\ \overline{cons} \in \Sigma & \Sigma; \Gamma \vdash \overline{u} : \Delta \\ \mathsf{c}\ \Delta_{\mathsf{c}} \in \overline{cons} & \Sigma; \Gamma \vdash \overline{v} : \Delta_{\mathsf{c}}[\overline{u}/\mathsf{vars}(\Delta)]\end{array}}{\Sigma; \Gamma \vdash \mathsf{c}\ \overline{v} : \mathsf{D}\ \overline{u}}\textsc{ConCall}$$

$$\frac{\begin{array}{cc}\mathsf{data}\ \mathsf{D}\ \Delta\ \overline{cons} \in \Sigma & \Sigma; \Gamma \vdash \overline{u} : \Delta \\ \overline{p} \Rightarrow \mathsf{c}\ \Delta_{\mathsf{c}} \in \overline{cons} & \mathsf{matches}(u, p) \mapsto \sigma \\ \multicolumn{2}{c}{\Sigma; \Gamma \vdash \overline{v} : \Delta_{\mathsf{c}}\sigma[\overline{u}/\mathsf{vars}(\Delta)]}\end{array}}{\Sigma; \Gamma \vdash \mathsf{c}\ \overline{v} : \mathsf{D}\ \overline{u}}\textsc{IxCall}$$

Rule for convertible types and type-in-type.

$$\frac{\Sigma; \Gamma \vdash a : A \qquad \Sigma; \Gamma \vdash A = B : \mathcal{U}}{\Sigma; \Gamma \vdash a : B} \qquad \Sigma; \Gamma \vdash \mathcal{U} : \mathcal{U}$$

Figure 2.4. Typing rules for types and terms

In the IxCall rule, we perform a pattern matching between the type arguments and the patterns in the constructor 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 Figure 2.5. 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 $\mathsf{matches}(u, p)$ operation and an extra substitution applied on Δ_{c}.

2.4 Signature well-formedness

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

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

- $\boxed{\Sigma; \Gamma \vdash p : A \mapsto \Theta}$ type-checking a pattern p against a type A.
- $\boxed{\Sigma; \Gamma \vdash \overline{p} : \Delta \mapsto \Theta}$ type-checking patterns $\overline{p}$ against a context Δ.

These rules are defined in Figure 2.5. They produce a context Θ containing all of the bindings in the give pattern(s).

Rules for one pattern.

$$\Sigma; \Gamma \vdash x : A \mapsto x : A$$

$$\frac{\mathsf{data}\ \mathsf{D}\ \Delta\ \overline{cons} \in \Sigma \qquad \mathsf{c}\ \Delta_{\mathsf{c}} \in \overline{cons} \qquad \Sigma; \Gamma \vdash \overline{p} : \Delta_{\mathsf{c}} \mapsto \Theta}{\Sigma; \Gamma \vdash \mathsf{c}\ \overline{p} : \mathsf{D}\ \overline{u} \mapsto \Theta}$$

$$\frac{\begin{array}{cc}\mathsf{data}\ \mathsf{D}\ \Delta\ \overline{cons} \in \Sigma & \overline{q} \Rightarrow \mathsf{c}\ \Delta_{\mathsf{c}} \in \overline{cons} \\ \mathsf{matches}(\overline{u}, \overline{q}) \mapsto \sigma & \Sigma; \Gamma \vdash \overline{p} : \Delta_{\mathsf{c}}\sigma \mapsto \Theta\end{array}}{\Sigma; \Gamma \vdash \mathsf{c}\ \overline{p} : \mathsf{D}\ \overline{u} \mapsto \Theta}$$

$$\frac{\begin{array}{cc}\multicolumn{2}{c}{\mathsf{data}\ \mathsf{D}\ \Delta\ \overline{cons} \in \Sigma} \\ (\mathsf{matches}(\overline{u}, \overline{p}) \mapsto \bot)_{\forall \overline{p} \Rightarrow \mathsf{c}\ \Delta_{\mathsf{c}} \in \overline{cons}} & \mathsf{c}\ \Delta_{\mathsf{c}} \notin \overline{cons}\end{array}}{\Sigma; \Gamma \vdash \mathsf{impossible} : \mathsf{D}\ \overline{u} \mapsto \Theta}$$

Rules for a list of patterns.

$$\frac{\Sigma; \Gamma \vdash q : A \mapsto \Theta \qquad \Sigma; \Gamma \vdash \overline{p} : \Delta[\mathsf{term}(q)/x] \mapsto \Theta'}{\Sigma; \Gamma \vdash q, \overline{p} : (x : A, \Delta) \mapsto \Theta \uplus \Theta'}$$

$$\Sigma; \Gamma \vdash \emptyset : \emptyset \mapsto \emptyset$$

Figure 2.5. Type checking of patterns

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

Proof. By induction on p. ∎

$$\frac{\Sigma;\Gamma \vdash \overline{p} : \Delta \mapsto \Theta \qquad \Sigma;\Gamma,\Delta,\Theta \vdash u : A[\mathsf{term}(\overline{p})/\mathsf{vars}(\Delta)]}{\Sigma;\Gamma \vdash \mathsf{clause}(\overline{p} : \Delta, u : A)}$$

$$\frac{\Sigma;\Gamma \vdash \overline{p} : \Delta \mapsto \Theta \qquad \Sigma;\Gamma,\Delta,\Theta \vdash \Delta_{\mathsf{c}}}{\Sigma;\Gamma \vdash \mathsf{clause}(\overline{p} : \Delta, \mathsf{c}\ \Delta_{\mathsf{c}})}$$

Figure 2.6. Pattern matching structures

$$\frac{\Sigma;\Gamma \vdash \qquad \Sigma;\Gamma \vdash \Delta \qquad (\Sigma;\Gamma,\Delta \vdash \Delta_{\mathsf{c}})_{\forall(\mathsf{c}\ \Delta_{\mathsf{c}})\in\overline{cons}} \qquad (\Sigma;\Gamma,\Delta \vdash \Delta_{\mathsf{c}} \qquad \mathsf{clause}(\overline{p} : \Delta, \mathsf{c}\ \Delta_{\mathsf{c}}))_{\forall(\overline{p}\Rightarrow\mathsf{c}\ \Delta_{\mathsf{c}})\in\overline{cons}}}{\Sigma, \mathsf{data}\ \mathsf{D}\ \Delta\ \overline{cons};\Gamma \vdash}$$

$$\frac{\Sigma;\Gamma \vdash \qquad \Sigma;\Gamma \vdash \Delta \qquad (\mathsf{clause}(\overline{p} : \Delta, u : A))_{\forall\overline{p}\Rightarrow u\in\overline{cls}}}{\Sigma, \mathsf{func}\ \mathsf{f}\ \Delta\ : A\ \overline{cls};\Gamma \vdash}$$

Figure 2.7. Well-formedness of signature Σ

Then, we define the rules for type checking pattern matching structures as in Figure 2.6 using the operation defined in Figure 2.5.

With them, we could define the type checking of function definitions and simpler indexed types, and form signature by the rules $\boxed{\Sigma;\Gamma \vdash}$ in Figure 2.7. The declarations are checked one after another so latter functions can depend on former ones. By that, we will lose induction-recursion [Dybjer 2003] and induction-induction [Forsberg and Setzer 2010], 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 [Norell 2009].

The syntax looks like this. We first define the type 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.

```
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 normalize function, which takes an instance of Term A and return an instance of type A. The type guarantees that there will never be ill-typed terms like succ (bool x).

```
normalize : Term A → A
normalize (nat x)        = x
normalize (bool x)       = x
normalize (succ x)       = suc (normalize x)
normalize (inv t)        = not (normalize t)
normalize (case b x y) = if (normalize b)
  then (normalize x) else (normalize y)
```

We cannot encode 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:

```
data TermTy : 𝓤 | natT | boolT
func termTy (x : TermTy) : 𝓤
 | natT ⇒ ℕ
 | boolT ⇒ Bool
```

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

```
data Term (n : TermTy) : 𝓤
 | natT  ⇒ nat ℕ
 | natT  ⇒ succ (Term natT)
 | boolT ⇒ bool boolT
 | boolT ⇒ inv (Term boolT)
 | A  ⇒ case (Term boolT) (Term A) (Term A)
func normalize (t : TermTy) (x : Term t) : termTy t
 | natT, nat n ⇒ n
 | natT, succ n ⇒ suc (normalize natT n)
 | boolT, bool b ⇒ b
 | boolT, inv b ⇒ not (normalize boolT b)
 | t, case b x y ⇒ ifElse (normalize boolT b)
   (normalize t x) (normalize t y)
```

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 *parameters* of the indexed type. The simplest case is the identity type:

```
data Id (A : Type ℓ) (x : A) : A → Type ℓ where
  idp : Id A x x
```

The index being x, a reference to the parameter of Id, is the essential reason why a general term unification needs to

$decl'$	::=	data D Δ $\overline{cons'}$	indexed type
$cons'$	::=	$\mid$ c : A	constructor

Figure 3.1. Syntax of general indexed types

be performed during the pattern matching over 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 Figure 3.1. 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 Δ_c in simpler indexed type D Δ, we translate it into a pattern matching constructor $\mathrm{vars}(\Delta) \Rightarrow$ c Δ_c.

After that, we could perform the translation of constructors. In other words, we need to construct the type ("A" in Figure 3.1) of the translated constructor.

Definition 3.1. For pattern matching constructor $\overline{p} \Rightarrow$ c Δ_c, we construct the type of the translated constructor as $\mathrm{vars}(\overline{p})\ \Delta_c \to$ D $\mathrm{term}(\overline{p})$.

This type is a pi type consisting of the following major components:

1. $\mathrm{vars}(\overline{p})$: the bindings in the patterns. We turn these bindings into parameters of the translated type.
2. Δ_c: the constructor parameters. They are typed under the bindings in $\mathrm{vars}(\overline{p})$, so we append the original parameters to the tail of these required bindings.
3. D $\mathrm{term}(\overline{p})$: the return type. We specialize the indices of D with the terms correspond to $\overline{p}$. These terms are typed under $\mathrm{vars}(\overline{p})$, which is available in the domain of this pi type.

Theorem (Completeness)**.** *Every simpler indexed type can be translated into general indexed types.*

Proof. This translation is defined for all simpler indexed types, so we get the completeness theorem for free. ◻

Theorem (Well-typedness)**.** *The type of the translated constructor is well-scoped and well-typed.*

Proof. First, it indeed returns a specialization of the type D.

According to Figure 2.6, types in Δ_c are well-typed with references to the bindings in $\overline{p}$, but not in Δ. These bindings are available in $\mathrm{vars}(\overline{p})$.

According to Figure 2.4, the well-typedness of D $\mathrm{term}(\overline{p})$ requires $\mathrm{term}(\overline{p}) : \Delta$, and we know it is true by lemma 2.2. ◻

Theorem (Soundness)**.** *The translated constructor needs to be matched if and only if the original constructor needs to be matched.*

The translated constructor does not need to be matched if and only if the original constructor does not need to be matched.

The translated constructor cannot be matched if and only if the original constructor cannot be matched.

Proof. This theorem actually requires a bit more information to be well-defined – we have not given the general indexed types typing rules and semantics.

However, since the result of $\mathrm{term}(\overline{p})$ is only generated by applications to constructors and references (by definition), we only need to deal with the unification of these terms, which are quite simple. They should be structurally equivalent to the rules in Figure 2.5.

Restricting the unification problem to this smaller subset makes the soundness theorem provable by induction on the patterns $\overline{p}$. ◻

Remark 3.2. This translation is also useful in the type checking of the constructors of simpler indexed types. When we have a reference to such constructor without any argument supplied, we could synthesize a type for this reference – and we use the type in definition 3.1.

3.3 Compilation and erasure

The compilation of simpler indexed types has an advantage over normal indexed types. In [Brady et al. 2004], they used detagging and forcing optimizations to erase the indices during compilation. These methods are directly expressible in our syntax as the indices are not even quantified in the constructors (see the examples in §1, where the implicit argument n in the Agda version of fzero, fsuc are not present in the corresponding definition as simpler indexed types). In other words, simpler indexed types enjoy the benefit of index erasure without any nontrivial compilation technique.

One downside is that there will not be inaccessible patterns [Brady et al. 2004], so there will be redundant pattern matching happening at runtime. Consider the example in §3.1, the normalize function using simpler indexed type has two pattern matchings, while the normalize function using general indexed types has only one.

In conclusion, simpler indexed types are more memory efficient than general indexed types without optimizations, while they require redundant pattern matchings. We think of the latter as a potential future work.

4 Conclusion

We introduced a simpler encoding of indexed types in dependent type systems. It reuses the pattern matching for constructor selection to avoid exposing the index unification problem to the users. A number of existing indexed types such as Fin and Vect can be encoded in this simpler way, but not all (exceptions include the identity type in Agda [Norell 2009] 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 [Cockx et al. 2018]. This translation requires an algorithm to classify the pattern matching clauses with overlapping parts. This is done in [Cockx et al. 2014b], 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 [Cockx et al. 2018] is a great example:

$$\begin{array}{l}\mathsf{data}\ \mathsf{Image}\ (A\ B : \mathsf{Type}\ \ell)\ (f : A \to B) : B \to \mathsf{Type}\ \ell\ \mathsf{where}\\ \quad \mathsf{image} : \forall\ x \to \mathsf{Image}\ A\ B\ f\ (f\ x)\end{array}$$

It can be encoded as an inductive type without indices:

$$\begin{array}{l}\mathsf{data}\ \mathsf{Image}\ (A\ B : \mathcal{U})\ (f : A \to B)\ (b : B) : \mathcal{U}\\ \mid\ \mathsf{image}\ (x : A)\ (p : f\ x = b)\end{array}$$

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 [Vezzosi et al. 2020, §3.2.4, §4] and the "index-fixing" fcoe operation discussed in [Cavallo and Harper 2019, §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 [Univalent Foundations Program 2013], the path type in cubical type theories [Cavallo and Harper 2019; Cohen et al. 2015; Vezzosi et al. 2019], the identity type in intuitionistic type theory [Martin-Löf 1975] (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 [Dybjer 2003] and induction-induction [Forsberg and Setzer 2010] 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 [Coquand 1992]. This idea was then put into a simpler type system (H-M) in [Augustsson and Petersson 1994], and was developed further as *first-class phantom types* in [Xi et al. 2003] and *guarded recursive type constructors* in [Cheney and Hinze 2003]. [Sheard and Pasalic 2008] 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 [Vytiniotis et al. 2006].

Indexed types [Zenger 1997] are the generalization of inductive types with type-equality, where values are also allowed to appear as parameters of inductive types. Agda [Norell 2009] and Idris [Brady 2013] have a more ergonomic design of indexed types where the equality relations are made implicit.

In [Cockx et al. 2018], 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 [Cockx et al. 2018]:

- 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 = ⊥.
- 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 [Chapman et al. 2010, §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 [Cockx et al. 2018]. The advantages of simpler indexed types over [Cockx et al. 2018] still applies to the encoding in [Chapman et al. 2010]. A notable application of indexed types based on [Chapman et al. 2010] is *ornaments* [Dagand and McBride 2014; Ko and Gibbons 2016].

The proposed feature has been implemented in two systems individually:

- The Arend [Group for Dependent Types and HoTT 2015] proof assistant, an implementation of homotopy type theory with a cubical-flavored interval type.
- The Aya [Aya developers 2021] proof assistant, an experimental implementation of a type theory similar to Arend's, but with other features such as overlapping and order-independent patterns [Cockx et al. 2014b].

All of the operations (except $\mathrm{vars}(\Delta)$ – 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: $\mathrm{vars}(p)$ corresponds to `PatTyper`, $\mathrm{term}(p)$ corresponds to `PatToTerm`, $\mathrm{matches}(u, p)$ corresponds to `PatMatcher`.

Apart from that, the type checking of terms in Figure 2.4 corresponds to `ExprTycker`, the type checking of patterns in Figure 2.5 corresponds to `PatTycker`, and the type checking of declarations in Figure 2.2 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

Guillaume Allais, James Chapman, Conor McBride, and James McKinna. 2017. Type-and-Scope Safe Programs and Their Proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017)*. Association for Computing Machinery, New York, NY, USA, 195–207. https://doi.org/10.1145/3018610.3018613

Lennart Augustsson and Kent Petersson. 1994. Silly Type Families DRAFT. (1994).

Aya developers. 2021. The Aya Proof Assistant. https://github.com/aya-prover/aya-dev. (2021).

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23 (9 2013), 552–593. Issue 05. https://doi.org/10.1017/S095679681300018X

Edwin Brady, Conor McBride, and James McKinna. 2004. Inductive Families Need Not Store Their Indices. In *Types for Proofs and Programs*, Stefano Berardi, Mario Coppo, and Ferruccio Damiani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 115–129.

Evan Cavallo and Robert Harper. 2019. Higher Inductive Types in Cubical Computational Type Theory. *Proc. ACM Program. Lang.* 3, POPL, Article 1 (Jan. 2019), 27 pages. https://doi.org/10.1145/3290314

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. 2010. The Gentle Art of Levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/1863543.1863547

James Cheney and Ralf Hinze. 2003. *First-class phantom types.* Technical Report. Cornell University.

Jesper Cockx and Andreas Abel. 2018. Elaborating Dependent (Co)Pattern Matching. *Proc. ACM Program. Lang.* 2, ICFP, Article 75 (July 2018), 30 pages. https://doi.org/10.1145/3236770

Jesper Cockx, Dominique Devriese, and Frank Piessens. 2014a. Pattern Matching without K. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 257–268. https://doi.org/10.1145/2628136.2628139

Jesper Cockx, Dominique Devriese, and Frank Piessens. 2016. Unifiers as Equivalences: Proof-Relevant Unification of Dependently Typed Data. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 270–283. https://doi.org/10.1145/2951913.2951917

Jesper Cockx, Gaëtan Gilbert, and Nicolas Tabareau. 2018. Vectors are records, too. *TYPES 2018* (2018).

Jesper Cockx, Frank Piessens, and Dominique Devriese. 2014b. Overlapping and Order-Independent Patterns. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 87–106.

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2015. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. *FLAP* 4 (2015), 3127–3170.

Thierry Coquand. 1992. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*. Citeseer, 71–83.

Pierre-Évariste Dagand and Conor McBride. 2014. Transporting functions across ornaments. *Journal of Functional Programming* 24, 2-3 (2014), 316–383. https://doi.org/10.1017/S0956796814000069

Peter Dybjer. 2003. A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. *Journal of Symbolic Logic* 65 (06 2003). https://doi.org/10.2307/2586554

Fredrik Nordvall Forsberg and Anton Setzer. 2010. Inductive-inductive definitions. In *International Workshop on Computer Science Logic*. Springer, 454–468.

Healfdene Goguen, Conor McBride, and James McKinna. 2006. *Eliminating Dependent Pattern Matching*. Springer Berlin Heidelberg, Berlin, Heidelberg, 521–540. https://doi.org/10.1007/11780274_27

Group for Dependent Types and HoTT. 2015. The Arend Proof Assistant. https://arend-lang.github.io. (2015).

Kaveh (https://mathoverflow.net/users/7507/kaveh). 2018. History of the notation for substitution. MathOverflow. (2018). https://mathoverflow.net/q/243084

HsiangShang Ko and Jeremy Gibbons. 2016. Programming with ornaments. *Journal of Functional Programming* 27 (2016), e2. https://doi.org/10.1017/S0956796816000307

Per Martin-Löf. 1975. An intuitionistic theory of types: predicative part. In *Logic Colloquium '73, Proceedings of the Logic Colloquium*, H.E. Rose and J.C. Shepherdson (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 80. North-Holland, 73–118.

Ulf Norell. 2009. Dependently Typed Programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation (TLDI '09)*. ACM, New York, NY, USA, 1–2. https://doi.org/10.1145/1481861.1481862

Tim Sheard and Emir Pasalic. 2008. Meta-programming With Built-in Type Equality. *Electronic Notes in Theoretical Computer Science* 199 (2008), 49–65. https://doi.org/10.1016/j.entcs.2007.11.012 Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004).

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Proc. ACM Program. Lang.* 3, ICFP, Article 87 (July 2019), 29 pages. https://doi.org/10.1145/3341691

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2020. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. (2020). Preprint available at https://staff.math.su.se/anders.mortberg/papers/cubicalagda2.pdf.

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2006. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming (ICFP'06)*. ACM SIGPLAN. https://www.microsoft.com/en-us/research/publication/simple-unification-based-type-inference-for-gadts/ 2016 ACM SIGPLAN Most Influential ICFP Paper Award.

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. Association for Computing Machinery, New York, NY, USA, 224–235. https://doi.org/10.1145/604131.604150

Christoph Zenger. 1997. Indexed Types. *Theor. Comput. Sci.* 187, 1–2 (Nov. 1997), 147–165. https://doi.org/10.1016/S0304-3975(97)00062-5