Elaborating Inductive Types to Custom Data Structures

AUTHOR(S), University of St Andrews, UK

ACM Reference Format:

Author(s). 2024. Elaborating Inductive Types to Custom Data Structures. 1, 1 (May 2024), 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Functional languages offer a high degree of expressiveness using a very small set of primitives: most employ a variant of the typed lambda calculus, commonly with schemas for defining recursive or inductive types. Such constructs allow programmers to define data and logic in a natural, algebraic way that is amenable to abstraction. Data structures such as lists, trees, and even natural numbers are the archetypical examples of inductively defined data, which offer pattern-matching as a principled way to introduce 'complete' branching points in a program. Despite these advantages, functional languages generally do not provide programmers with many tools to specify how their programs should be represented on physical computers. For inductively-defined data, there is a fixed representation that the majority of functional languages utilise, which is to represent each constructor as a heap cell, and link chains of constructors together using pointer indirection. As a result, a list as an inductively defined data structure is stored as a linked list, and a natural number is stored as a unary number where each digit is an 'empty' heap cell. To get around this issue, most real-world implementations expose underlying machine primitive types such as contiguous arrays and bitvectors, and programmers are able to utilise these instead of the 'algebraic' inductively-defined types to make their functional programs more performant. In Haskell, the `array` package comes to mind, as well as the `Integer` and `Natural` primitives which utilise a GMP-style big-integer implementation internally.

1.1 The 'Nat-hack'

In functional languages with dependent types, such as Idris or Lean, the issue of type representation is further complicated by the fact that inductive proofs come into the mix. Inductively defined types enjoy induction principles which can prove facts about them on a case-by-case basis. The standard example is the induction over the natural numbers `data Nat = Z | S Nat` given by

```
induction : (P : Nat -> Type) -> P Z -> ((m : Nat) -> P m -> P (S m)) -> (n : Nat) -> P n
```

This can be proven trivially by well-founded recursion and case analysis over `Z` and `S`. However, if natural numbers are not defined inductively, but rather opaquely as an intricate data structure like Haskell's `Natural`, induction is no longer given 'for free', and must be manually proven internally in the language. On the other hand, if natural numbers are defined inductively, then many operations become inexcusably slow, such as multiplication

Author's address: Author(s), University of St Andrews, St Andrews, UK, kt81@st-andrews.ac.uk.

```
mul : Nat -> Nat -> Nat
mul Z b = Z
mul (S a) b = add (mul a b) b
```

taking $a \times b$ steps to compute for input numbers a and b. To solve this problem, Idris, Agda and Lean 'short-circuit' the default inductive type representation for natural numbers, to use arbitrarily-sized big integers for calculations whose digits are bitvectors, rather than unary numbers.

To describe how this trick works, assume we have access to a type `BigUInt` for arbitrarily-sized big integers, along with some primitive operations

```
bigZero, bigOne : BigUint
bigAdd, bigMul, bigSub : BigUInt -> BigUInt -> BigUInt
bigIsZero : BigUInt -> Bool
```

In a well-typed input program, all occurences of the zero constructor `Z : Nat` should be replaced with the constant `bigZero`, and all occurences of the successor constructor `S : Nat -> Nat` should be replaced with the expression `\x -> bigAdd x bigOne`. For case analysis, each pattern matching expression `case x of Z -> b | S n -> r n` should be replaced with the conditional expression `if bigIsZero x then b else r (bigSub x bigOne)`. Additionally, some basic functions on natural numbers should be replaced with more performant variants. The recursively-defined addition function `add : Nat -> Nat -> Nat` should be replaced with `bigAdd` and similarly `mul` should be replaced with `bigMul`. This way, the high-level program still appears to be using the inductive definition `Nat`, but upon compilation it uses `BigUInt` for efficient execution.

1.2 Beyond natural numbers

There are arguably many inductive types which could admit a more optimised representation than the default linked tree. The first which comes to mind is the type of lists `data List t = Nil | Cons t (List t)`. There exist many representations of lists in memory, including flattened contiguous heap-backed arrays with dynamic resizing, singly-linked lists, doubly-linked lists, their circular variants, tree-based representations like binary search trees, their balanced variants, B and B+ trees, and segment trees. Each representation offers different performance characteristics for common operations such as appending, insertion, deletion, splicing, concatenating, and so on. Nevertheless, all of them are `List` in spirit; there is a 'canonical' bijection between a list and any of the afforementioned data structures. A functional program using the algebraic `List` type could potentially benefit from a different representation depending on the exact operations it performs and in what proportion. Not only lists, but structures such as trees, finite sets, as well as refinement predicates on types such as element proofs or parity proofs, could all be subject to more optimal representations. Since, at first glance, such an alteration could be done purely mechanically in a similar way to the 'Nat-hack', it is natural to wonder if this technique readily generalises to user-defined inductive types such that the transformation itself is specified in the same language.

1.3 Contributions

This paper develops an extension of dependently-typed lambda calculus with inductive constructions, in order to support the definition of custom representations of the inductive constructions, along with the specialisation of functions for fine tuning to the chosen representations. This system features correct-by-construction representations, awarding the programmer with a bijection proof between an inductive type and its representation, as well as an elaboration into a lower language with a guarantee that no inductive constructors or eliminators remain. The standard linked tree representation of inductive types that is the hardcoded default in most implementations is incarnated as 'just another representation' in this system, special only because it can apply to *any* inductive type. The order of these developments in the paper are as follows:

- In section 2.1, the language λ_{PRIM} is introduced, which is a staged language with dependent types, whose object-level fragment contains some machine primitives that act as building blocks of data representations.
- In section 2.2, the language λ_{IND} is introduced as an extension of λ_{PRIM}, which allows the familiar definition of inductive types, living in a universe of 'codes' for object-level types.
- In section 2.3 The language λ_{REP} is introduced as the completion of λ_{IND}, containing a 'representation' construct that assigns concrete object-level codes to inductive types.
- In section 3, an elaboration procedure $\mathcal{R} : \lambda_{\text{REP}} \to \lambda_{\text{PRIM}}$ is formulated which eliminates inductive constructs through the defined representations, yielding the final program that can be staged and compiled.
- In section 4 The standard linked tree representation is recovered in λ_{REP} for any inductive type, and shown to be coherent.
- In section 5, the 'Nat-hack' is defined internally in λ_{REP} and shown to be coherent up to some notable assumptions. The system is further explored, showcasing representations for other inductive types.
- Finally, in section 6, some desirable extensions as part of future work are discussed.

2 A TYPE SYSTEM FOR DATA REPRESENTATIONS

In this section, we describe a type system for data representations in a staged language. We start by defining a core staged language λ_{PRIM} with Σ/Π types, identity types, and a universe of types $\mathcal{U}_i$ for each stage i, as well as a set of object-level machine primitives. We then introduce inductive constructions in the meta-fragment of λ_{PRIM} to form λ_{IND}. Finally, we introduce data representations in λ_{IND} and extend the system with rules for data representations in λ_{REP}. Since staged dependent type systems and inductive constructions are well understood, we will focus on the novel aspects of data representations. The languages above form an inclusion hierarchy

$$\lambda_{\text{PRIM}} \subset \lambda_{\text{IND}} \subset \lambda_{\text{REP}}$$

and our goal is to describe a transformation from λ_{REP} to λ_{PRIM}, elaborating away inductive definitions with the help of data representations. Finally, the resulting program in λ_{PRIM} can be staged to the purely object-level language that represents the target architecture. When defining these languages, we will use a BNF-like syntax for terms and contexts, natural deduction-style typing rules, as well as CWF-style notation for contexts, types, and terms.

2.1 A core staged language with machine primitives, λ_{PRIM}

As a first step towards a type system for data representations, we informally describe a staged dependent type system λ_{PRIM} with Σ/Π types, identity types, and a universe of types $\mathcal{U}_i$ for each stage $i \in \{0, 1\}$. This serves as a background system on which we introduce inductive constructions and data representations in. The raw syntax of λ_{PRIM} is given by the BNF grammar

$$t ::= \mathcal{U}_i \mid x \mapsto t \mid t\ t \mid x \mid (x : t) \to t \mid (x : t) \times t \mid a =_A b \mid \pi_1\, t \mid \pi_2\, t \mid \mathbf{refl} \mid \Uparrow t \mid \langle t \rangle \mid \sim t$$

This follows the standard typing rules of 2LTT with Σ, Π and identity types [3], without regard for the universe hierarchy, as this is orthogonal to the main focus of this work. Notably, $\mathcal{U}_0 : \mathcal{U}_0$ is the object-level universe, and $\mathcal{U}_1 : \mathcal{U}_1$ is the meta-level universe. We will use $\mathbf{let}_i$ notation for binding, though this is just syntactic sugar for redexes.

We need to add explicit substitutions here!

On top of this base syntax, we assume a certain set of primitives that exist in the object language. These will be used to form data representations. They are provided by the target architecture, which is represented by the object language:

- A type of booleans, $\mathsf{Bool} : \mathcal{U}_0$, with constants $\mathsf{true} : \mathsf{Bool}$ and $\mathsf{false} : \mathsf{Bool}$, and operations and, or, and not. Elimination for booleans is also provided, in the form of

$$\mathsf{ifthenelse} : (b : \mathsf{Bool}) \to ((b =_{\mathsf{Bool}} \mathsf{true}) \to A) \to ((b =_{\mathsf{Bool}} \mathsf{false}) \to A) \to A.$$

- A type of machine words, $\mathsf{Word} : \mathcal{U}_0$ with constants $0, 1, 2, \ldots$ and standard binary numeric operations add, sub, mul, as well as Bool-valued comparison operations eq, lt, and gt. We will use the notation Word_n to denote the type $(w : \mathsf{Word}) \times \mathsf{lt}\, w\, n =_{\mathsf{Bool}} \mathsf{true}$.
- A type of n-sized sequences of type A, $[A; n] : \mathcal{U}_0$, where $n : \mathsf{Word}$ and $A : \mathcal{U}_0$. Sequences come with indexing operations $\mathsf{get} : [A; n] \to \mathsf{Word}_n \to A$ and $\mathsf{set} : [A; n] \to \mathsf{Word}_n \to A \to [A; n]$, as well as an initialisation operation $\mathsf{repeat} : (a : A) \to (n : \mathsf{Word}) \to [A; n]$. Sequences should be thought of as unboxed arrays, living on the stack.
- A boxing type constructor $\Box A : \mathcal{U}_0$ where $A : \mathcal{U}_0$, with boxing and unboxing operations $\mathsf{box}\, a : \Box A$ and $\mathsf{unbox}\, b : A$. Values of type $\Box A$ represent explicitly heap-allocated values of type A.

These primitives do not necessarily form an exhaustive list; indeed, we will sometimes expect further properties of these primitives to hold propositionally in the form of additional primitive lemmas, such as $(n : \mathsf{Word}) \to \mathsf{add}\, 0\, n =_{\mathsf{Word}} n$. Precise details are only necessary when implementing such a system, and the definitions above are sufficient for the present discussion.

2.2 Extending λ_{PRIM} with inductive constructions

Building on top of λ_{PRIM}, we introduce an extension of the system for named inductive constructions in the meta-language, called λ_{IND}. First, we present a raw syntax for signatures Σ containing items Z consisting of data declarations, constructor declarations, and definitions:

$$\Sigma ::= \cdot \mid \Sigma, Z$$

$$Z ::= \textbf{data}\ \mathsf{D}\,\Delta : \Uparrow\mathcal{U}_0 \mid \textbf{ctor}\ \mathsf{C}\,\Delta : \mathsf{D}\,\Delta \mid \textbf{closed}\ \mathsf{D}\,\vec{\mathsf{C}} \mid \textbf{def}\ \mathsf{f} : t = t$$

The symbols in blue represent labels, which are used to uniquely identify elements of a signature. A data definition $\textbf{data}\ \mathsf{D}\,\Delta : \Uparrow\mathcal{U}_0$ introduces a new inductive type D with a telescope Δ of arguments. Inductive types should be thought of as *codes* for object-level types, eventually to be replaced by the defined representations. A constructor definition $\textbf{ctor}\ \mathsf{C}\,\Delta_{\mathsf{C}} : \mathsf{D}\,\Delta$ introduces a new constructor C for the inductive type D, with a telescope Δ_{C} of parameters which may depend on D's parameters. Closed declarations $\textbf{closed}\ \mathsf{D}\,\vec{\mathsf{C}}$ specify that the constructors $\vec{\mathsf{C}}$ are the only constructors for the inductive type D in the present signature. A definition $\textbf{def}\ \mathsf{f} : t = t$ introduces a new symbol f of type t and value t. The reason for including named definitions in a signature is to allow them to be overriden as part of data representations.

The system allows for the definition of inductive types as well as inductive families; the telescope Δ in a data declaration defines the parameters of the inductive type, and index refinement can occur in the constructor telescopes Δ_{C} through the usage of equality types, paired with the ability to reference the parameters of the inductive type Δ. Such an approach is syntactically simpler than the standard approach of constructor signatures as Π types common in most proof assistants, but is equivalent in expressive power [1]. For example, to define the type of vectors indexed by their length, we can write

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Vec}\ (A : {\Uparrow}\mathcal{U}_0,\ n : \mathsf{Nat}) : {\Uparrow}\mathcal{U}_0 \\
&\textbf{ctor}\ \mathsf{nil}\ (n =_{\mathsf{Nat}} \mathsf{z}) : \mathsf{Vec}\ A\ n \\
&\textbf{ctor}\ \mathsf{cons}\ (a : A,\ m : \mathsf{Nat},\ v : \mathsf{Vec}\ A\ m,\ m =_{\mathsf{Nat}} \mathsf{s}\ n) : \mathsf{Vec}\ A\ n \\
&\textbf{closed}\ \mathsf{Vec}\ (\mathsf{nil}, \mathsf{cons})
\end{aligned}
$$

In a real implementation, these definitions can be elaborated from a more familiar syntax involving index refinement.

The syntax of terms is extended accordingly with labels applied to arguments

$$t ::= \ldots \mid \mathsf{L}\,\vec{t}$$

where $\vec{t}$ denotes a sequence of terms and L refers to any valid label in a signature (data, constructor, or definition). For example, given a constructor

$$\textbf{ctor}\ \mathsf{cons}\ (x : T,\ xs : \mathsf{List}\ T) : \mathsf{List}\ T$$

we can write $\mathsf{cons}\,a\,l$ to denote the full application of the 'cons' constructor to an element a and a list l. Partial applications are also allowed as syntactic syntactic sugar, so that $\mathsf{cons}\,a$ is shorthand for $l \mapsto \mathsf{cons}\,a\,l$. Each inductive definition $\textbf{data}\ \mathsf{D}\,\Delta : {\Uparrow}\mathcal{U}_0$ exposes an additional label $\mathsf{case}_{\mathsf{D}}$ which is used to perform case analysis on terms of the inductive type D. As such, valid labelled application terms in a signature are given through the typing rules of λ_{IND} and do not correspond exactly to the labels *present* in the signature's items.

In λ_{IND}, contexts are fibered over signatures, so that a context Γ is well formed with respect to a signature Σ. The syntax for telescopes Γ and contexts Δ is

$$
\begin{aligned}
\Gamma &::= \cdot \mid \Gamma, x : t \\
\Delta &::= \cdot \mid \Delta, x : t\,.
\end{aligned}
$$

As is standard in 2LTT, contexts can be extended with types from any stage. Telescopes are very similar to contexts, but restricted to types from a single stage and well formed with respect to a context Γ, meaning that telescopes can contain open terms. We use the notation $\Delta \to t$ to denote a repeated function type with parameters from Δ and codomain t which may depend on the parameters. Additionally, we will sometimes explicitly bind the names of a telescope such as $(\vec{x} : \Delta) \to t[\vec{x}]$. Similar syntax is used to extend contexts with telescopes: Γ, Δ or $\Gamma, \vec{x} : \Delta$.

The language λ_{IND} is equipped with the following judgment forms:

- Σ sig — The signature Σ is well-formed.
- $\Sigma \vdash \Gamma$ con — In signature Σ the context Γ is well-formed.
- $\Sigma \mid \Gamma \vdash \Delta\ \mathsf{tel}_i$ — In signature Σ and context Γ, the telescope Δ is well-formed in stage i.
- $\Sigma \mid \Gamma \vdash T\ \mathsf{type}_i$ — In signature Σ and context Γ, T is a well-formed type in stage i.
- $\Sigma \mid \Gamma \vdash a : T$ — In signature Σ and context Γ, a is a well-formed term of type T.
- $Z \in \Sigma$ — The item Z is present in the signature Σ.
- $\Sigma \vdash Z$ — The item Z is well-formed in the signature Σ.
- L label $\notin \Sigma$ — The label L does not appear in the signature Σ.

Need to add the equality judgement for types and terms.

First, we present the rules for well formed signatures, contexts and telescopes in fig. 1.

Next, the rules for well-formed items in signatures are given in fig. 2.

Awkward spacing!

$$\textsc{Sig-Empty}\ \frac{}{\cdot\ \mathsf{sig}} \qquad \textsc{Sig-Extend}\ \frac{\Sigma\ \mathsf{sig} \qquad \Sigma \vdash Z}{\Sigma, Z\ \mathsf{sig}} \qquad \textsc{Con-Empty}\ \frac{\Sigma\ \mathsf{sig}}{\Sigma \vdash \cdot\ \mathsf{con}} \qquad \textsc{Con-Extend}\ \frac{\Sigma \vdash \Gamma\ \mathsf{con} \qquad \Sigma \mid \Gamma \vdash T\ \mathsf{type}_i}{\Sigma \vdash \Gamma, T\ \mathsf{con}} \qquad \textsc{Tel-Empty}\ \frac{\Sigma \vdash \Gamma\ \mathsf{con}}{\Sigma \mid \Gamma \vdash \cdot\ \mathsf{tel}_i}$$

$$\textsc{Tel-Extend}\ \frac{\Sigma \mid \Gamma \vdash \Delta\ \mathsf{tel}_i \qquad \Sigma \mid \Gamma \vdash T\ \mathsf{type}_i}{\Sigma \mid \Gamma \vdash \Gamma, T\ \mathsf{tel}_i}$$

Fig. 1. Rules for signatures, contexts and telescopes in λ_{IND}.

$$\textsc{Data-Item}\ \frac{\Sigma \mid \cdot \vdash \Delta\ \mathsf{tel}_1 \qquad \mathsf{D}\ \mathsf{label} \notin \Sigma}{\Sigma \vdash \mathbf{data}\ \mathsf{D}\ \Delta : \Uparrow\mathcal{U}_0} \qquad \textsc{Ctor-Item}\ \frac{\mathbf{data}\ \mathsf{D}\ \Delta : \Uparrow\mathcal{U}_0 \in \Sigma \qquad \mathbf{closed}\ \mathsf{D}, \_ \notin \Sigma \qquad \mathsf{C}\ \mathsf{label} \notin \Sigma \qquad \Sigma \mid \Delta \vdash \Delta_{\mathsf{C}}\ \mathsf{tel}_1}{\Sigma \vdash \mathbf{ctor}\ \mathsf{C}\ \Delta_{\mathsf{C}} : \mathsf{D}\ \Delta}$$

$$\textsc{Closed-Item}\ \frac{\mathbf{data}\ \mathsf{D}\ \Delta : \Uparrow\mathcal{U}_0 \in \Sigma \qquad \forall i \in I.\ \mathbf{ctor}\ \mathsf{C}_i\ \Delta_{\mathsf{C}_i} : \mathsf{D}\ \Delta \in \Sigma \qquad \mathbf{closed}\ \mathsf{D}, \_ \notin \Sigma}{\Sigma \vdash \mathbf{closed}\ \mathsf{D}, \vec{\mathsf{C}}} \qquad \textsc{Def-Item}\ \frac{\Sigma \mid \cdot \vdash m : M \qquad \mathsf{f}\ \mathsf{label} \notin \Sigma}{\Sigma \vdash \mathbf{def}\ \mathsf{f} : M = m}$$

Fig. 2. Rules for items in signatures in λ_{IND}.

$$\textsc{Data-Form}\ \frac{\mathbf{data}\ \mathsf{D}\ \Delta : \Uparrow\mathcal{U}_0 \in \Sigma \qquad \Sigma \mid \Gamma \vdash \vec{t} : \Delta}{\Sigma \mid \Gamma \vdash \mathsf{D}\ \vec{t} : \Uparrow\mathcal{U}_0}$$

$$\textsc{Data-Intro}\ \frac{\mathbf{data}\ \mathsf{D}\ \Delta : \Uparrow\mathcal{U}_0 \in \Sigma \qquad \mathbf{ctor}\ \mathsf{C}\ \Delta_{\mathsf{C}} : \mathsf{D}\ \Delta \in \Sigma \qquad \Sigma \mid \Gamma \vdash \vec{t} : \Delta \qquad \Sigma \mid \Gamma \vdash \vec{u} : \Delta_{\mathsf{C}}[\vec{t}]}{\Sigma \mid \Gamma \vdash \mathsf{C}\ \vec{u} : \mathsf{D}\ \vec{t}}$$

$$\textsc{Data-Elim}\ \frac{\begin{array}{c}\mathbf{data}\ \mathsf{D}\ \Delta : \Uparrow\mathcal{U}_0 \in \Sigma \qquad \forall i \in I.\ \mathbf{ctor}\ \mathsf{C}_i\ \Delta_{\mathsf{C}_i} : \mathsf{D}\ \Delta \in \Sigma \qquad \mathbf{closed}\ \mathsf{D}\ \vec{\mathsf{C}} \in \Sigma \qquad \Sigma \mid \Gamma \vdash \vec{t} : \Delta \\ \Sigma \mid \Gamma \vdash \eta : \mathsf{D}\ \vec{t} \qquad \Sigma \mid \Gamma,\ \vec{x} : \Delta,\ h : \mathsf{D}\ \vec{x} \vdash T : \Uparrow\mathcal{U}_0 \qquad \forall i \in I.\ \Sigma \mid \Gamma,\ \vec{x} : \Delta,\ \vec{u} : \Delta_{\mathsf{C}_i}[\vec{x}] \vdash m_i : T[\vec{x}, \mathsf{C}_i \vec{u}]\end{array}}{\Sigma \mid \Gamma \vdash \mathsf{case}_{\mathsf{D}}\ \eta\ \vec{m} : T[\vec{t}, \eta]}$$

$$\textsc{Def-Intro}\ \frac{\mathbf{def}\ \mathsf{f} : M = m \in \Sigma}{\Sigma \mid \Gamma \vdash \mathsf{f} : M}$$

Fig. 3. Typing rules for terms and types related to items in λ_{REP}.

There is a difference betwen an item being well-formed in a signature, and being *present* in a signature. The former describes items which are candidates to be added to a signature, which often relies on the absence of certain items in the signature, such as duplicate labels not being present, or a data type not being closed. We do not explicitly require that defined data types are well-founded or strictly positive, though this is a desirable property for a real implementation.

Actually, I think this should be a requirement, because otherwise case elimination is not quite valid...

Having defined well-formed signatures, it is now time to define the rules of well-formed types and terms in λ_{IND}, shown in fig. 3. For brevity, only the rules that relate to items in signatures are shown, that is, rules regarding the introduction and elimination of data types, constructors, and definitions. The rest of the rules are standard and can be found in the literature on 2LTT [3], with the modification that everything is now fibered over signatures. This completes a description of a staged language with dependent types and inductive constructions, λ_{IND}. In the next section, we move on to the main focus of this work, data representations.

Here we must define the equality rules for constructors

2.3 Data representations in λ_{REP}

So far, we have described λ_{IND}, a dependently-typed staged language with object-level machine primitives as well as named inductive constructions and definitions. We now introduce data representations in the syntax for items, forming the language λ_{REP}. The goal of data representations is to provide a way to represent data types, constructions, and definitions in a more efficient manner, by transforming them into more suitable data structures for the target architecture. This is achieved by defining a new kind of item in signatures, called a representation, which specifies how to interpret a given item in the object language. With no further restrictions, a user would be able to arbitrarily change the semantics of the meta-level items through representations, which would be undesirable. Instead, we restrict data representations to preserve the intended semantics of the original items, allowing for a correct-by-construction transformation from λ_{REP} to λ_{PRIM}. This is done in different ways for data types, constructors, and definitions.

First, we extend the raw syntax of items Z with representations, forming λ_{REP}:

$$Z ::= \dots \mid \textbf{repr}\ \mathsf{L}\ \Delta\ \textbf{as}\ t$$

A representation $\textbf{repr}\ \mathsf{L}\ \Delta\ \textbf{as}\ t$ asks for a definition L with parameters Δ to be represented by a term t. Data, constructor and definition representations all share the same raw syntax and are distinguished by the subject L. In the typing rules we expect that every meta-level type A has a defined representation $\mathcal{R}A$. Extensions to this system could support a sub-class of types with representations, retaining the ability to have purely meta-level types with no defined representations. We define $\mathcal{R}A$ to be the representation of a type A and $\mathcal{R}a$ the representation of a term a, whose type is $\mathcal{R}A$. These compute definitionally according to rules developed in section 3, and indeed constitute the elaboration process into λ_{PRIM}. For now however, the definition of $\mathcal{R}$ can remain opaque. We only need to know that $\mathcal{R}$ preserves the structure of the base type theory λ_{PRIM}, that is, it preserves Σ/Π types, identity types, and universes. With that, we can now define the additional typing rules for λ_{REP}, in fig. 4

A data definition is represented by an object-level type A which is allowed to depend on the *represented* parameters of the data type. For example, if D is the type of natural numbers, then the type A could be an arbitrary precision big integer type. The actual representation of the inhabitants of the data type is determined by the representation of the constructors. Each constructor C_i is represented by a term t_i which can reference the parameters of the data type and the parameters of the constructor. The term t_i is an inhabitant of the representation type A, but with an extra set of proofs attached to it through a Σ type. These are *inequality* constraints which ensure that the representation values of different constructors are distinct. More specifically, the i-th constructor representation t_i must not be equal to the representation value of any other constructor t_j for $j \neq i$. It suffices to check this for the constructors that have been defined so far, relaxing the proof obligations to only consider *previous* constructors. In effect, the mapping from the data type D to the representation type A is injective, and given by the ensemble of constructor representations.

An injective mapping is halfway to an isomorphism, but we need to ensure that it is also surjective. This is done by defining a representation for the case analysis $\mathsf{case}_{\mathsf{D}}$ which takes a term $\eta : A$ and a set of branches $\vec{m}$, one for

$$\textsc{Repr-Data}\quad \frac{\textbf{data}\ \mathsf{D}\,\Delta : \Uparrow\mathcal{U}_0 \in \Sigma \qquad \Sigma \mid \vec{x} : \mathcal{R}\Delta \vdash A : \mathcal{U}_0}{\Sigma \vdash \textbf{repr}\ \mathsf{D}\,\vec{x}\ \textbf{as}\ A}$$

$$\textsc{Repr-Ctor}\quad \frac{\begin{array}{c}\textbf{data}\ \mathsf{D}\,\Delta : \Uparrow\mathcal{U}_0 \in \Sigma \qquad \forall i \in I.\ \textbf{ctor}\ \mathsf{C}_i\,\Delta_{\mathsf{C}_i} : \mathsf{D}\,\Delta \in \Sigma \qquad \textbf{repr}\ \mathsf{D}\,\mathcal{R}\Delta\ \textbf{as}\ A \in \Sigma \\ \forall j < i.\ \textbf{repr}\ \mathsf{C}_j\,\mathcal{R}\Delta_{\mathsf{C}_j}\ \textbf{as}\ t_j \in \Sigma \qquad \Sigma \mid \vec{x} : \mathcal{R}\Delta,\ \vec{z} : \mathcal{R}\Delta_{\mathsf{C}_i} \vdash t_i : (a : A) \times \prod_{j<i}((\vec{y} : \mathcal{R}\Delta_{\mathsf{C}_j}) \to a \neq \pi_1 t_j[\vec{x}, \vec{y}])\end{array}}{\Sigma \vdash \textbf{repr}\ \mathsf{C}_i\,\vec{z}\ \textbf{as}\ t_i}$$

$$\textsc{Repr-Case}\quad \frac{\begin{array}{c}\textbf{data}\ \mathsf{D}\,\Delta : \Uparrow\mathcal{U}_0 \in \Sigma \qquad \textbf{repr}\ \mathsf{D}\,\mathcal{R}\Delta\ \textbf{as}\ A \in \Sigma \qquad \forall i \in I.\ \textbf{ctor}\ \mathsf{C}_i\,\Delta_{\mathsf{C}_i} : \mathsf{D}\,\Delta \in \Sigma \qquad \forall i \in I.\ \textbf{repr}\ \mathsf{C}_i\,\mathcal{R}\Delta_{\mathsf{C}_i}\ \textbf{as}\ t_i \in \Sigma \\ \textbf{closed}\ \mathsf{D}\,\vec{\mathsf{C}} \in \Sigma \qquad \Sigma \mid T : (\vec{x} : \mathcal{R}\Delta) \to \Uparrow A[\vec{x}] \to \mathcal{U}_0,\ \vec{a} : \mathcal{R}\Delta,\ \eta : A[\vec{a}],\ \vec{m} : \mathrm{Cases}(T, \vec{a}, \eta) \vdash e : T(\vec{a}, \eta) \\ \text{where } \mathrm{Cases}(T, \vec{a}, \eta) = (m_i : (\vec{y} : \mathcal{R}\Delta_{\mathsf{C}_i}) \to (\eta =_{A[\vec{a}]} \pi_1 t_i[\vec{a}, \vec{y}]) \to T(\vec{a}, \pi_1 t_i[\vec{a}, \vec{y}]) \mid i \in I)\end{array}}{\Sigma \vdash \textbf{repr}\ \mathsf{case}_{\mathsf{D}}\ T\,\vec{a}\,\eta\,\vec{m}\ \textbf{as}\ e}$$

$$\textsc{Repr-Def}\quad \frac{\textbf{def}\ \mathsf{f} : M = m \in \Sigma \qquad \Sigma \mid \cdot \vdash a : (m' : \mathcal{R}M) \times (\mathcal{R}m =_{\mathcal{R}M} m')}{\Sigma \vdash \textbf{repr}\ \mathsf{f}\ \textbf{as}\ a}$$

Fig. 4. Rules for data representations in λ_{REP}.

each constructor, and produces some elimination term of type T. A term e is used to represent the result of the case analysis, which can reference η and the branches $\vec{m}$. In practice, e will be a certain kind of branching construct, which reproduces the behaviour of the case analysis function $\mathsf{case}_{\mathsf{D}}$ but on the object-level type A instead of the data type D. Each branch m_i is a function which takes the parameters of the constructor C_i, but also a proof that the subject η indeed reduces to the representation value of the constructor t_i. This proof must be provided by e when invoking the m_is. There is no other way to construct a term of type T than through the constructor representations, and e *must* map η to something of type T. As a result, e *must* provide the appropriate proofs for each branch, and must handle every branch. If such a representation for case analysis can be constructed, it must mean that the mapping is surjective, and thus an isomorphism between the data type D and the representation type A is established.

This argument should be made more formal

There is another way to view data representations that might elucidate the situation. It is known that algebras over inductive families can be interpreted as *ornaments* [2]; an inductive type is decorated with the values of the algebra at each node. We can apply the constructor representation algebra $(t_i \mid i \in I)$ to the inductive type D to obtain the ornamented type $\tilde{\mathsf{D}}$. The case analysis $\mathsf{case}_{\mathsf{D}}$ representation can then be viewed as a *section* of $\tilde{\mathsf{D}}$ by the represented type A, that is, a dependent function

$$(a : A) \to \tilde{\mathsf{D}}(a)$$

omitting all the indices for brevity. An injective function with a section is another name for an isomorphism.

Finally, we define the representation of definitions. A definition $\textbf{def}\ \mathsf{f} : M = m$ is represented by a term a which is a pair of an inhabitant m' of $\mathcal{R}M$ and a proof that the representation of the term m is equal to this inhabitant. Most of the time, definitions are expected to be functions, for example the addition function on natural numbers. In that case, since $\mathcal{R}M$ preserves Π types, the given m' must also to be a function, and the proof must be a proof of equality of functions. In order to make this possible, we expect the type theory to support function extensionality, so that equality of functions is defined pointwise.

We really need to use the proofs to show some formal correctness argument..

3 ELABORATION INTO A CORE STAGED LANGUAGE

The elaboration procedure $\mathcal{R} : \lambda_{\text{REP}} \longrightarrow \lambda_{\text{PRIM}}$ is still left to be defined. The goal of this procedure is to eliminate inductive constructs through the defined representations, yielding the final program that can be staged and compiled. To define $\mathcal{R}$, we first need to introduce a concept of *concrete signatures*. A concrete signature is a signature where all items are accompanied by representations. In other words, Σ is a concrete signature when

- if **data** $\mathsf{D}\,\Delta : \Uparrow\mathcal{U}_0 \in \Sigma$, then $\exists A\,\vec{x}.$ **repr** $\mathsf{D}\,\vec{x}$ **as** $A \in \Sigma$,
- if **ctor** $\mathsf{C}\,\Delta_{\mathsf{C}} : \mathsf{D}\,\Delta \in \Sigma$, then $\exists t\,\vec{z}.$ **repr** $\mathsf{C}\,\vec{z}$ **as** $t \in \Sigma$,
- if **closed** $\mathsf{D}\,\vec{\mathsf{C}} \in \Sigma$, then $\exists T\,\vec{a}\,\eta\,\vec{m}.$ **repr** $\mathsf{case}_{\mathsf{D}}\,T\,\vec{a}\,\eta\,\vec{m}$ **as** $e \in \Sigma$, and
- if **def** $\mathsf{f} : M = m \in \Sigma$, then $\exists a.$ **repr** f **as** $a \in \Sigma$.

A term or type can only be elaborated if the signature it is well formed in is concrete. This is a restriction that ensures that the elaboration procedure can always look up the representation of a data item, constructor or definition. We can now define the elaboration $\mathcal{R}$ for types and terms in a concrete signature. The definition of $\mathcal{R}$ is given in a mutually recursive CWF-style presentation, mapping constructs in λ_{REP} to λ_{PRIM}. More specifically, we define $\mathcal{R}$ as mapping contexts, types, and terms in λ_{REP}, to contexts, types, and terms in λ_{PRIM}, respectively. And substitutions!

$$
\begin{array}{lll}
\mathcal{R} : \mathrm{Con}_{\text{REP}}\,\Sigma \longrightarrow \mathrm{Con}_{\text{PRIM}} & \mathcal{R} : \mathrm{Ty}_{\text{REP},i}\,\Sigma\,\Gamma \longrightarrow \mathrm{Ty}_{\text{PRIM},i}\,\mathcal{R}\Gamma & \mathcal{R} : \mathrm{Tm}_{\text{REP},i}\,\Sigma\,\Gamma\,T \longrightarrow \mathrm{Tm}_{\text{PRIM},i}\,\mathcal{R}\Gamma\,\mathcal{R}T \\
\mathcal{R}(\cdot) = \cdot & \mathcal{R}(\mathsf{D}\,\vec{t}) = \Uparrow\mathcal{R}A_{\mathsf{D}}[\mathcal{R}\vec{t}] & \mathcal{R}(\mathsf{C}\,\vec{t}) = \langle \mathcal{R}(\pi_1\,t_{\mathsf{C}})[\mathcal{R}\vec{t}]\rangle \\
\mathcal{R}(\Gamma, T) = \mathcal{R}\Gamma, \mathcal{R}T & \text{else recurse with } \mathcal{R} & \mathcal{R}(\mathsf{case}_{\mathsf{D}}\,\eta\,\vec{m}) = \langle \mathcal{R}e_{\mathsf{C}}[\_,\_,\mathcal{R}\eta,\mathcal{R}\vec{m}]\rangle \\
 & & \mathcal{R}\mathsf{f} = \pi_1\,a_{\mathsf{f}} \\
 & & \text{else recurse with } \mathcal{R}
\end{array}
$$

Fig. 5. Definition of $\mathcal{R}$, where Σ is a concrete signature.

The elaboration of a type $\mathsf{D}\,\vec{t}$ is given by its defined representation A_{D} applied to the elaborations of its arguments. The elaboration of a constructor term $\mathsf{C}\,\vec{t}$ is given by its defined representation t_{C} applied to the elaborations of its arguments. Similarly, the elaboration of a closed eliminator term $\mathsf{case}_{\mathsf{D}}\,\eta\,\vec{m}$ is given by its defined representation e_{C} applied to the elaborations of its arguments. Finally, the elaboration of a definition f is given by its defined representation a_{f}. Notice that for data types, constructors, and case expressions, the representation terms themselves must be elaborated, since they are allowed to reference other data types, constructors defined before them. The elaboration procedure is type-preserving by construction, which means that if $\Sigma \mid \Gamma \vdash_{\text{REP}} t : T$, then $\mathcal{R}\Gamma \vdash_{\text{PRIM}} \mathcal{R}t : \mathcal{R}T$. We must also show that elaboration preserves the equality rules, which can be done through the extra proofs given in the representations!

4 RECOVERING THE DEFAULT LINKED REPRESENTATION

Default-Data

$$\frac{\begin{array}{c}\Sigma \text{ concrete} \\ \Sigma \vdash Z := \textbf{data}\ \mathsf{D}\ \Delta : \Uparrow\mathcal{U}_0 \qquad \forall i \in I.\ \Sigma, Z, (X_j)_{j<i} \vdash X_i := \textbf{ctor}\ \mathsf{C}_i\ \Delta_{\mathsf{C}_i} : \mathsf{D}\ \Delta \qquad \Sigma, Z, (X_i)_{i\in I} \vdash L := \textbf{closed}\ \mathsf{D}\ \vec{\mathsf{C}}\end{array}}{\begin{array}{c}\Sigma, Z, X, L \vdash \textbf{repr}\ \mathsf{D}\ \vec{x}\ \textbf{as}\ (\mu F.\vec{x} \mapsto (c : \text{Word}_{[\![I]\!]}) \times \square(\text{switch}\ c\ (\text{Ctors}\ \vec{x}\ F))))\ \vec{x} \\ \text{where Ctors}\ \vec{x}\ F = ((d : \text{Data}_i\ \vec{x}) \times ((r : \text{Rec}_i\ \vec{x}\ d) \to F(\text{Idx}_i\ r)))_{i\in I}\end{array}}$$

Repr-Ctor

$$\frac{\begin{array}{c}\textbf{data}\ \mathsf{D}\ \Delta : \Uparrow\mathcal{U}_0 \in \Sigma \qquad \forall i \in I.\ \textbf{ctor}\ \mathsf{C}_i\ \Delta_{\mathsf{C}_i} : \mathsf{D}\ \Delta \in \Sigma \qquad \textbf{repr}\ \mathsf{D}\ \mathcal{R}\Delta\ \textbf{as}\ A \in \Sigma \\ \forall j < i.\ \textbf{repr}\ \mathsf{C}_j\ \mathcal{R}\Delta_{\mathsf{C}_j}\ \textbf{as}\ t_j \in \Sigma \qquad \Sigma \mid \vec{x} : \mathcal{R}\Delta,\ \vec{z} : \mathcal{R}\Delta_{\mathsf{C}_i} \vdash t_i : (a : A) \times \prod_{j<i}((\vec{y} : \mathcal{R}\Delta_{\mathsf{C}_j}) \to a \neq \pi_1 t_j[\vec{x}, \vec{y}])\end{array}}{\Sigma \vdash \textbf{repr}\ \mathsf{C}_i\ \vec{z}\ \textbf{as}\ t_i}$$

Repr-Case

$$\frac{\begin{array}{c}\textbf{data}\ \mathsf{D}\ \Delta : \Uparrow\mathcal{U}_0 \in \Sigma \qquad \textbf{repr}\ \mathsf{D}\ \mathcal{R}\Delta\ \textbf{as}\ A \in \Sigma \qquad \forall i \in I.\ \textbf{ctor}\ \mathsf{C}_i\ \Delta_{\mathsf{C}_i} : \mathsf{D}\ \Delta \in \Sigma \qquad \forall i \in I.\ \textbf{repr}\ \mathsf{C}_i\ \mathcal{R}\Delta_{\mathsf{C}_i}\ \textbf{as}\ t_i \in \Sigma \\ \textbf{closed}\ \mathsf{D}\ \vec{\mathsf{C}} \in \Sigma \qquad \Sigma \mid T : (\vec{x} : \mathcal{R}\Delta) \to \Uparrow A[\vec{x}] \to \mathcal{U}_0,\ \vec{a} : \mathcal{R}\Delta,\ \eta : A[\vec{a}],\ \vec{m} : \text{Cases}\ T\ \vec{a}\ \eta \vdash e : T(\vec{a}, \eta) \\ \text{where Cases}\ T\ \vec{a}\ \eta = (m_i : (\vec{y} : \mathcal{R}\Delta_{\mathsf{C}_i}) \to (\eta =_{A[\vec{a}]} \pi_1 t_i[\vec{a}, \vec{y}]) \to T(\vec{a}, \pi_1 t_i[\vec{a}, \vec{y}]) \mid i \in I)\end{array}}{\Sigma \vdash \textbf{repr}\ \mathsf{case}_{\mathsf{D}}\ T\ \vec{a}\ \eta\ \vec{m}\ \textbf{as}\ e}$$

Default-Def

$$\frac{\Sigma \text{ concrete} \qquad \Sigma \vdash F := \textbf{def}\ \mathsf{f} : M = m}{\Sigma, F \vdash \textbf{repr}\ \mathsf{f}\ \textbf{as}\ (\mathcal{R}m, \textbf{refl})}$$

Fig. 6. Default data representations in λ_{REP}. These are not rules, but derivations.

5 NATURAL NUMBERS, AND OTHER EXAMPLES

6 CONCLUSIONS AND FUTURE WORK

Extensions:

- ps data - multiple reprs - class of "compile-time" inductive types - multiple pattern clauses - quotients

REFERENCES

[1] Jesper Cockx and Andreas Abel. 2018. Elaborating dependent (co)pattern matching. *Proc. ACM Program. Lang.* 2, ICFP (July 2018), 1–30. https://doi.org/10.1145/3236770

[2] Pierre-Evariste Dagand. 2017. The essence of ornaments. *J. Funct. Programming* 27 (Jan. 2017), e9. https://www.cambridge.org/core/services/aop-cambridge-core/content/view/4D2DF6F4FE23599C8C1FEA6C921A3748/S0956796816000356a.pdf/div-class-title-the-essence-of-ornaments-div.pdf

[3] András Kovács. 2022. Staged compilation with two-level type theory. (Sept. 2022). arXiv:2209.09729 [cs.PL]