Elaborating Inductive Types to Custom Data Structures

CONSTANTINE THEOCHARIS, University of St Andrews, UK

ACM Reference Format:
Constantine Theocharis. 2024. Elaborating Inductive Types to Custom Data Structures. 1, 1 (May 2024), 9 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: Constantine Theocharis, 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:

- 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.
- 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.
- 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. 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.
- The 'Nat-hack' is defined internally in λ_{REP} and shown to be coherent up to some notable assumptions.
- The standard linked tree representation is recovered in λ_{REP} for any inductive type, and shown to be coherent.
- The system is further explored, showcasing representations for other inductive types. Some desirable extensions as part of future work are discussed.

2 DATA REPRESENTATIONS, INFORMALLY

3 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. (TODO: expand!)

3.1 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 [16], 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.

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 $\mathsf{set} : [A; n] \to \mathsf{Word}_n \to A \to [A; n]$. Sequences should be thought of as unboxed arrays, living on the stack.
- A boxing type constructor $\square A : \mathcal{U}_0$ where $A : \mathcal{U}_0$, with boxing and unboxing operations $\mathsf{box}\, a : \square A$ and $\mathsf{unbox}\, b : A$. Values of type $\square A$ 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.

3.2 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:

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

The symbols in blue represent labels, which are used to uniquely identify elements of a signature. A data definition $\mathbf{data}\ \mathsf{D}\ \Delta : \Uparrow\mathcal{U}_0$ introduces a new inductive type D with a telescope Δ of arguments. A constructor definition $\mathbf{ctor}\ \mathsf{C}\ \Delta_{\mathsf{C}} : \mathsf{D}\ \Delta$ 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 $\mathbf{closed}\ \mathsf{D}\ \vec{\mathsf{C}}$ specify that the constructors $\vec{\mathsf{C}}$ are the only constructors for the inductive type D in the present signature. A definition $\mathbf{def}\ \mathsf{f} : t = t$ introduces a new symbol f of type t and value t. The reason for including function definitions in a signature is to allow for named functions in the meta-language, which can be overriden as part of data representations.

The syntax for telescopes Δ is
$$\Delta ::= \cdot \mid \Delta, x : t\,,$$
resembling the syntax for contexts, but restricted to meta-level types and well-typed 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 syntax of terms must be 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.

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.
- $\Sigma \mid \Gamma \vdash T\ \mathrm{type}_i$ — 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.

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

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

$$\textsc{Data-Item}\ \frac{\Sigma \mid \cdot \vdash \Delta\ \mathrm{tel}_1 \qquad \mathsf{D}\ \mathrm{label} \notin \Sigma}{\Sigma \vdash \textbf{data}\ \mathsf{D}\,\Delta : \Uparrow\mathcal{U}_0}$$

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

$$\textsc{Closed-Item}\ \frac{\textbf{data}\ \mathsf{D}\,\Delta : \Uparrow\mathcal{U}_0 \in \Sigma \qquad \forall i \in I.\ \textbf{ctor}\ \mathsf{C}_i\,\Delta_{\mathsf{C}_i} : \mathsf{D}\,\Delta \in \Sigma \qquad \textbf{closed}\ \mathsf{D}, \_ \notin \Sigma}{\Sigma \vdash \textbf{closed}\ \mathsf{D}, \vec{\mathsf{C}}}$$

$$\textsc{Def-Item}\ \frac{\Sigma \mid \cdot \vdash m : M \qquad \mathsf{f}\ \mathrm{label} \notin \Sigma}{\Sigma \vdash \textbf{def}\ \mathsf{f} : M = m}$$

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

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

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

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

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

Fig. 3. Typing rules for items in λ_{REP}.

3.3 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 meta-language, 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 represent a given meta-level 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}:

$$Z ::= \ldots \mid \mathbf{repr}\ \mathsf{L}\,\Delta\ \mathbf{as}\ t$$

A representation $\mathbf{repr}\ \mathsf{L}\,\Delta\ \mathbf{as}\ t$ 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 will require 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. For this 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 shown in fig. 4. With that, we can now define the additional typing rules for λ_{REP}, in fig. 5

The correctness of a representation of some data type D is ensured by the correctness of the representations of each of its constructors C_i and the case analysis function $\mathsf{case}_{\mathsf{D}}$. Therefore there is no special correctness condition for the data representation type itself. The constructor representations $\mathbf{repr}\ \mathsf{C}_i\ \mathcal{R}\Delta_{\mathsf{C}_i}\ \mathbf{as}\ t_i$ form an *algebra* over the endofunctor $[\![\mathsf{D}]\!]$ associated with the data type D, by packaging all the representation values $\pi_1 t_i$, where the carrier

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

Fig. 4. Definition of $\mathcal{R}$, where Σ is a concrete signature.

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

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

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

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

Fig. 5. Typing rules for data representations in λ_{REP}.

object is the defined representation type A. Moreover, this algebra is *injective* in the sense that the representation values are unique for each constructor, ensured by the equality constraints $\pi_1 t_i \neq \pi_1 t_j$ for $i \neq j$.

It is known that algebras over indexed inductive types can be interpreted as *ornaments* [11]; 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 is thus a *section* of $\tilde{\mathsf{D}}$ by the represented type A. In other words, it must ensure that the subject η of the case analysis is used to index into the ornamented type $\tilde{\mathsf{D}}$, or put another way, that the propositional equality $\eta = \pi_1 t_i$ holds when the branch m_i is invoked. Overall, this yields an isomorphism between the inductive type D and the represented type A, in the Kleisli category of the code-generation monad of the object-level language (TODO: expand on this!).

In fig. 5, the symbols $\mathcal{R}$ and $\mathcal{R}$ *lower* the given atoms from the meta level to the object level, through the defined representations in Σ. To define them, we 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 $\mathbf{data}\ \mathsf{D}\ \Delta : \Uparrow\mathcal{U}_0 \in \Sigma$, then $\exists A\ \vec{x}.\ \mathbf{repr}\ \mathsf{D}\ \vec{x}\ \mathbf{as}\ A \in \Sigma$,
- if $\mathbf{ctor}\ \mathsf{C}\ \Delta_{\mathsf{C}} : \mathsf{D}\ \Delta \in \Sigma$, then $\exists t\ \vec{z}.\ \mathbf{repr}\ \mathsf{C}\ \vec{z}\ \mathbf{as}\ t \in \Sigma$,
- if $\mathbf{closed}\ \mathsf{D}\ \vec{\mathsf{C}} \in \Sigma$, then $\exists T\ \vec{a}\ \eta\ \vec{m}.\ \mathbf{repr}\ \mathsf{case}_{\mathsf{D}}\ T\ \vec{a}\ \eta\ \vec{m}\ \mathbf{as}\ e \in \Sigma$, and

- if **def** f : $M = m \in \Sigma$, then $\exists a.$ **repr** f **as** $a \in \Sigma$.

We can now define the lowering functions $\mathcal{R}$ and $\mathcal{R}$ as follows, where all the signatures Σ are assumed to be concrete:

3.4 Default representations

Default-Data

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

Repr-Ctor

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

Repr-Case

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

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.

Todo:

- rephrase kleisli category thing - denotational semantics? - restrict image of R to lambda prim. - correctness : initiality of algebras is preserved from lambda rep to lambda prim

Extensions:

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

4 ELABORATION INTO A CORE STAGED LANGUAGE

5 BEYOND NATURAL NUMBERS, LISTS AND TREES

6 CONCLUSIONS AND FUTURE WORK

REFERENCES

[1] Guillaume Allais. 2023. Builtin Types Viewed as Inductive Families. In *Programming Languages and Systems*. Springer Nature Switzerland, 113–139.

[2] Guillaume Allais. 2023. Seamless, Correct, and Generic Programming over Serialised Data. (Oct. 2023). arXiv:2310.13441 [cs.PL]

[3] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor Mcbride, and Peter Morris. 2015. Indexed containers. *J. Funct. Programming* 25 (Jan. 2015), e5.

[4] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. 2017. Two-Level Type Theory and Applications. (May 2017). arXiv:1705.03307 [cs.LO]

[5] Steve Awodey. 2014. Natural models of homotopy type theory. (June 2014). arXiv:1406.3219 [math.CT]

[6] Thaïs Baudon, Gabriel Radanne, and Laure Gonnord. 2023. Bit-Stealing Made Legal: Compilation for Custom Memory Representations of Algebraic Data Types. *Proc. ACM Program. Lang.* 7, ICFP (Aug. 2023), 813–846.

[7] S Boulier. 2018. Extending type theory with syntactic models. (Nov. 2018).

[8] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The next 700 syntactical models of type theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs* (Paris, France) *(CPP 2017)*. Association for Computing Machinery, New York, NY, USA, 182–194.

[9] Edwin C Brady and Kevin Hammond. 2010. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. *SIGPLAN Not.* 45, 9 (Sept. 2010), 297–308.

[10] Simon Castellan, Pierre Clairambault, and Peter Dybjer. 2019. Categories with Families: Unityped, Simply Typed, and Dependently Typed. (April 2019). arXiv:1904.00827 [cs.LO]

[11] 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

[12] Peter Dybjer. 1994. Inductive families. *Form. Asp. Comput.* 6, 4 (Jan. 1994), 440–465.

[13] Brandon Hewer and Graham Hutton. 2024. Quotient Haskell: Lightweight Quotient Types for All. *Proc. ACM Program. Lang.* 8, POPL (Jan. 2024), 785–815.

[14] Ralf Hinze. 2011. Type Fusion. In *Algebraic Methodology and Software Technology*. Springer Berlin Heidelberg, 92–110.

[15] Ambrus Kaposi and Jakob von Raumer. 2020. A syntax for mutual inductive families.

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

[17] M Sato, Takafumi Sakurai, and Yukiyoshi Kameyama. 2001. A simply typed context calculus with first-class environments. *J. Funct. Log. Prog.* (March 2001), 359–374.

[18] Zhong Shao, John H Reppy, and Andrew W Appel. 1994. Unrolling lists. In *Proceedings of the 1994 ACM conference on LISP and functional programming* (Orlando, Florida, USA) *(LFP '94)*. Association for Computing Machinery, New York, NY, USA, 185–195.

[19] Walid Taha and Tim Sheard. 1997. Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (Amsterdam, The Netherlands) *(PEPM '97)*. Association for Computing Machinery, New York, NY, USA, 203–217.

[20] Taichi Uemura. 2019. A General Framework for the Semantics of Type Theory. *arXiv [math.CT]* (April 2019).

[21] Marcos Viera and Alberto Pardo. 2006. A multi-stage language with intensional analysis. In *GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering*. unknown.

[22] Jeremy Yallop. 2017. Staged generic programming. *Proc. ACM Program. Lang.* 1, ICFP (Aug. 2017), 1–29.

[23] Robert Atkey Sam Lindley Yallop. [n. d.]. Unembedding Domain-Specific Languages. https://bentnib.org/unembedding.pdf. Accessed: 2024-2-22.