Elaborating Inductive Types to Optimised Data Structures

CONSTANTINE THEOCHARIS, University of St Andrews, UK

ACM Reference Format:
Constantine Theocharis. 2024. Elaborating Inductive Types to Optimised Data Structures. 1, 1 (May 2024), 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Dependently-typed functional languages offer high-level expressiveness, but unlike imperative languages such as C, they limit control in behaviour of programs when it comes to runtime execution. Indeed, this is intentional; by letting go of low-level control, programs become easier to write, understand, and modify. However, relinquishing this control often leads to unoptimised programs that do not take full advantage of the architecture they are running on. The default memory representation for inductive types is a linked tree, where each constructor is a node and the recursive data it holds are the children of that node. This representation is very flexible, because it can represent any inductive data type, but it is not always the most efficient.

1.1 The 'Nat-hack'

Introduce Nat-hack and detail how it works in Idris, Lean, Coq, etc.

Provide perf tests for Nat hack vs others

1.2 Beyond natural numbers

Identify other data structures that would benefit (list-like things, trees)

1.3 A staging-based approach

Staging and why is it needed?

1.4 Contributions

- A type system $\lambda_{2+\text{Rep}}$ for defining data representations based in a staged language, in a correct-by-construction manner.
- An elaboration procedure from $\lambda_{2+\text{Rep}}$ to λ_2, which translates inductive data into the format defined by the representations.
- Application of this type system to define the 'Nat-hack' in a principled way and show that it is correct.
- A further exploration of the type system, showing how to represent other inductive types such as lists and trees.

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

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

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

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 $\textbf{let}_i$ 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, constructor, and function definitions:

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

$$Z ::= \underline{\mathbf{data}}\ \mathsf{D}\, \Delta : \mathcal{U}_1 \mid \underline{\mathbf{ctor}}\ \mathsf{C}\, \Delta : \mathsf{D}\, \Delta \mid \underline{\mathbf{closed}}\ \mathsf{D}\, \vec{\mathsf{C}} \mid \underline{\mathbf{fun}}\ \mathsf{f} : t = t$$

The symbols in blue represent labels, which are used to uniquely identify elements of a signature. A data definition $\underline{\mathbf{data}}\ \mathsf{D}\, \Delta : \mathcal{U}_1$ introduces a new inductive type D with a telescope Δ of arguments. A constructor definition $\underline{\mathbf{ctor}}\ \mathsf{C}\, \Delta_{\mathsf{C}} : \mathsf{D}\, \Delta$ 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 $\underline{\mathbf{closed}}\ \mathsf{D}\, \vec{\mathsf{C}}$ specify that the constructors $\vec{\mathsf{C}}$ are the only constructors for the inductive type D in the present signature. A function definition $\underline{\mathbf{fun}}\ \mathsf{f} : t = t$ introduces a new named function 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 function definitions). For example, given a function

$$\underline{\mathbf{fun}}\ \mathsf{id} : (A : \mathcal{U}_1) \to A \to A = \_ \mapsto x \mapsto x$$

we can write $\mathsf{id}\, A\, a$ to denote the full application of the identity function to a type A and a term $a : A$. Partial applications are also allowed as syntactic syntactic sugar, so that $\mathsf{id}\, A$ is shorthand Each inductive definition $\underline{\mathbf{data}}\ \mathsf{D}\, \Delta : \mathcal{U}_1$ exposes an additional function label $\mathsf{case}_{\mathsf{D}}$ which is used to perform case analysis on terms of the inductive type D. for $x \mapsto y \mapsto \mathsf{id}\, A\, x\, y$. 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:

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

TODO: typing rules

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 function 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 functions 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 functions.

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

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

A representation $\underline{\textbf{repr}}\ \mathsf{L}\ \Delta\ \underline{\textbf{as}}\ t$ asks for a definition L with parameters Δ to be represented by a term t. Data, constructor and function 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 $\textbf{repr}\,A$. 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 extend the raw syntax of terms to

$$t ::= \dots \mid \textbf{repr}\ t \mid \textbf{Repr}\ t$$

where $\textbf{Repr}\,A$ is the representation of a type A and $\textbf{repr}\,a$ is the representation of a term a, whose type is $\textbf{Repr}\,A$. With that, we can now define the additional typing rules for λ_{REP}, in fig. 1

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 $\underline{\textbf{repr}}\ \mathsf{C}_i \Downarrow\Delta_{\mathsf{C}_i}\ \underline{\textbf{as}}\ t_i$ 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 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.

Repr-Ctor

$$\frac{\begin{array}{c}\underline{\textbf{data}}\ \mathsf{D}\ \Delta : \mathcal{U}_1 \in \Sigma \qquad \forall i \in I.\ \underline{\textbf{ctor}}\ \mathsf{C}_i\ \Delta_{\mathsf{C}_i} : \mathsf{D}\ \Delta \in \Sigma \\ \underline{\textbf{repr}}\ \mathsf{D}\ {\Downarrow}\Delta\ \underline{\textbf{as}}\ A \in \Sigma \qquad \forall j < i.\ \underline{\textbf{repr}}\ \mathsf{C}_j\ {\Downarrow}\Delta_{\mathsf{C}_j}\ \underline{\textbf{as}}\ t_j \in \Sigma \qquad \Sigma \mid {\Downarrow}\Delta, {\Downarrow}\Delta_{\mathsf{C}} \vdash t_i : (a : A) \times \prod_{j<i}(a \neq \pi_1 t_j)\end{array}}{\Sigma \vdash \underline{\textbf{repr}}\ \mathsf{C}_i\ {\Downarrow}\Delta_{\mathsf{C}_i}\ \underline{\textbf{as}}\ t_i}$$

Repr-Case

$$\frac{\begin{array}{c}\underline{\textbf{data}}\ \mathsf{D}\ \Delta : \mathcal{U}_1 \in \Sigma \\ \underline{\textbf{repr}}\ \mathsf{D}\ {\Downarrow}\Delta\ \underline{\textbf{as}}\ A \in \Sigma \qquad \forall i \in I.\ \underline{\textbf{ctor}}\ \mathsf{C}_i\ \Delta_{\mathsf{C}_i} : \mathsf{D}\ \Delta \in \Sigma \qquad \forall i \in I.\ \underline{\textbf{repr}}\ \mathsf{C}_i\ {\Downarrow}\Delta_{\mathsf{C}_i}\ \underline{\textbf{as}}\ t_i \in \Sigma \qquad \underline{\textbf{closed}}\ \mathsf{D}\ \vec{\mathsf{C}} \in \Sigma \\ \Sigma \mid T : (\vec{x} : {\Downarrow}\Delta) \to A[\vec{x}] \to \mathcal{U}_0,\ \vec{a} : {\Downarrow}\Delta,\ \eta : A[\vec{a}],\ \left(m_i : (\vec{y} : {\Downarrow}\Delta_{\mathsf{C}_i}) \to (\eta =_{A[\vec{a}]} \pi_1 t_i[\vec{a},\vec{y}]) \to T(\vec{a}, \pi_1 t_i[\vec{a},\vec{y}]) \mid i \in I\right) \vdash e : T(\vec{a}, \eta)\end{array}}{\Sigma \vdash \underline{\textbf{repr}}\ \mathsf{case}_{\mathsf{D}}\ T\ \vec{a}\ \eta\ \vec{m}\ \underline{\textbf{as}}\ e}$$

Repr-Fun

$$\frac{\underline{\textbf{fun}}\ \mathsf{f} : M = m \in \Sigma \qquad \Sigma \vdash a : (m' : {\Downarrow}M) \times ({\downarrow}m =_{{\Downarrow}M} m')}{\Sigma \vdash \underline{\textbf{repr}}\ \mathsf{f}\ \underline{\textbf{as}}\ a}$$

Fig. 1. Typing rules for data representations in $\lambda_{\textsc{rep}}$.

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. 1, the symbols $\Downarrow$ and $\downarrow$ *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

- if $\underline{\textbf{data}}\ \mathsf{D}\ \Delta : \mathcal{U}_1 \in \Sigma$, then $\exists A\ \widehat{\Delta}.\ \underline{\textbf{repr}}\ \mathsf{D}\ \widehat{\Delta}\ \underline{\textbf{as}}\ A \in \Sigma$,
- if $\underline{\textbf{ctor}}\ \mathsf{C}\ \Delta_{\mathsf{C}} : \mathsf{D}\ \Delta \in \Sigma$, then $\exists t\ \widehat{\Delta_{\mathsf{C}}}.\ \underline{\textbf{repr}}\ \mathsf{C}\ \widehat{\Delta_{\mathsf{C}}}\ \underline{\textbf{as}}\ t \in \Sigma$,
- if $\underline{\textbf{closed}}\ \mathsf{D}\ \vec{\mathsf{C}} \in \Sigma$, then $\exists T\ \vec{a}\ \eta\ \vec{m}.\ \underline{\textbf{repr}}\ \mathsf{case}_{\mathsf{D}}\ T\ \vec{a}\ \eta\ \vec{m}\ \underline{\textbf{as}}\ e \in \Sigma$, and
- if $\underline{\textbf{fun}}\ \mathsf{f} : M = m \in \Sigma$, then $\exists a.\ \underline{\textbf{repr}}\ \mathsf{f}\ \underline{\textbf{as}}\ a \in \Sigma$.

We can now define the lowering functions $\Downarrow$ and $\downarrow$ as follows, where all the signatures Σ are assumed to be concrete:

$$\begin{aligned} &{\Downarrow} : \mathsf{Con}\ \Sigma \to \mathsf{Con}\ \Sigma \\ &{\Downarrow}(\cdot) = \cdot \\ &{\Downarrow}(\Delta, T) = {\Downarrow}\Delta, {\Downarrow}T \end{aligned} \qquad \begin{aligned} &{\Downarrow} : \mathsf{Ty}_1\ \Sigma\ \Gamma \to \mathsf{Ty}_0\ \Sigma\ {\Downarrow}\Gamma \\ &{\Downarrow}(\mathsf{D}\ \vec{t}) = A[{\downarrow}\vec{t}] \\ &{\Downarrow}(A \to B) = {\Downarrow}A \to {\Downarrow}B \\ &{\Downarrow}(A \times B) = {\Downarrow}A \times {\Downarrow}B \\ &{\Downarrow}\mathcal{U}_1 = \mathcal{U}_0 \\ &{\Downarrow}\mathsf{El}\,t = {\downarrow}t \end{aligned}$$

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.