Submitted to MFPS 2024

Staging Inductive Types to Optimised Data Structures

Constantine Theocharis Christopher Brown

University of St Andrews
St Andrews, Fife, UK

1 Introduction

Need to start with the general idea first. Something like functional languages offer high-level expressiveness, but, unlike imperative languages such as C, limit control in the behaviour - in fact FP is designed to be this way on purpose, so also need to be careful about the argument here. Then the argument should be built up from here... what if programmers want to offer the high-level flexibility and also the control... etc. Would be nice to have some specific examples here where the control is really necessary. Perhaps an implementation of a particular algorithm needs to fit into the memory constrains of an embedded device. The more tangible the better.

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.

Languages such as Haskell, Idris, and OCaml make heavy use of *inductive data types*. An inhabitant of an inductive data type is one of a defined set of *constructors*, each of which can hold arbitrary data. Constructors are also allowed to be *recursive*, meaning that a constructor of a data type can hold an inhabitant of the same data type. Such data types turn out to be very expressive, able to encode natural numbers, lists, trees, and many other data structures in a very elegant and mathematical way. Similarly, functions that operate on inductive data types can utilise *pattern matching*, which is a branching operation depending on which constructor is given as input.

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.

What is a good example to introduce this idea? Some ideas are: programming with matrices/arrays, numeric computation, snoc lists. Perhaps it is useful to highlight the fact that different representations are convenient for programming vs theorem proving.

How much stuff do we need to say about inductive data types?

1.1 Preliminaries

The technique of program staging enables writing programs that produce other programs as output. There are many use cases for this ability; for the purposes of this work, we are interested in being able to write a program in a high-level language, and then translating certain parts of it into a *quoted* form consisting of

low-level constructs, and finally *staging* the resultant high-level program with quotations into a low-level program.

aims to separate the high-level structure of a program in a way that is convenient for abstraction and manipulation, from the low-level eventual representation of the program that is efficient for machine execution. This is done by separating a language into two parts: the *meta* fragment and the *object* fragment. The meta fragment is the site in which the program is synthesised, and the object fragment is the output of the synthesis process. This is made possible by the ability to manipulate

I think this is more preliminaries and belongs in the preliminary section, not the introduction. Need to give examples here of what a meta fragment is, and what an object fragment is.

- Introduce 2LTT, CWFs, give an overview about how the translation is defiend

1.2 Contributions

TODO

- We present a formalism for the expression of a choice of representation for inductive data types.
  Formalism using what? why?
- We develop a transformation procedure from inductive data types to their chosen representation.
- We extend the transformation to allow for an intermediate staging of inductive constructors to further refine the staging output, emulating a kind of intensional analysis.
  doesn't this come under contribution 2?
- We show semantic preservation of the entire transformation modulo its preservation by each chosen representation.
  For what language? Presume you also need to define the semantics?

2 Examples and technique

- Theorem proving using matrices, to show why it is useful to have induction
- Matrix multiplication, where matrices are a representable inductive data type, to show that the same matrix type can also be used for efficient computations.
- Snoc lists, and a transition function from cons to snoc, showing that under a given representation it is zero-cost.
- Natural numbers using big integers; a principled way to do the Idris hack.

The type of natural numbers is an example of a ubiquitous

better to say common or typical

inductive data type that is used extensively in theorem proving and general functional programming

more specifically DT languages

, defined as

$$\textbf{data}\ \texttt{Nat} = \texttt{Z} \mid \texttt{S}\ \texttt{Nat}\,. \tag{1}$$

Such a definition in a language such as Haskell [CITE] would be represented as a linked list at runtime. That is, a memory representation of the form

but in Idris it wouldn't, and is represented by an actual number...

... memory layout thing from SPLS talk

Performing arithmetic operations on this data structure would involve traversing the linked list. On the other hand, computers allow the direct manipulation of bitvectors and offer native operations for arithmetic on them. Therefore, if we care about performance we should instead represent natural numbers

as

$$\textbf{data}\ \texttt{Nat} = \texttt{MkNat}\ [\texttt{Word}]\,. \tag{2}$$

need to explain what this code does

Unfortunately, even though arithmetic can be defined more efficiently on this representation, it is harder to work with, because its constructor structure diverges from the typical mathematical definition of natural numbers

what do you mean by "harder" - that is a very subjective thing. I'd argue both are equivalent in expressiveness

. More concretely, to define a function or predicate on the natural numbers, it suffices to define it on 0, and define it for $n+1$ given the result for n. This strategy can be achieved in a concise and readable way using pattern matching on `Nat` if it is defined as in (1):

$$\begin{aligned} &f : \texttt{Nat} \to A \\ &f\ \texttt{Z} = \ldots \\ &f\ (\texttt{S}\ n) = \ldots \end{aligned}$$

However, if `Nat` is defined as in (2), then the definition of f becomes more cumbersome:

$$\begin{aligned} &f : \texttt{Nat} \to A \\ &f\ \texttt{MkNat}\ [0] = \ldots \\ &f\ n' = \textbf{let}\ n = n' - 1\ \textbf{in}\ \ldots \end{aligned}$$

these need explained and described

The technique we present here allows the programmer to define the natural numbers as in (1), and then automatically transform the definition to the representation in (2) for runtime performance reasons

the technique is more general than that: it allows any inductive type to be defined as in 1, but transformed into a variant of 2

.

2.1 Representations of inductive types

This seems to come from nowhere and it's not easy to understand what it is for or how it connects to anything that came before

Yes, I think this doesn't really fit in here; It should instead appear in the next section as motivation for why the `Repr` struct is defined as it is.

In **Set**-based semantics of inductive types, we interpret an inductive data type `F` as the initial algebra of the associated endofunctor F. This takes a set X to the set of constructors of `F`, replacing each recursive parameter with X. The carrier of the initial algebra of F is the least fixpoint of F, denoted μF, where μF is equivalent to the actual data type `F`. We have an isomorphism between $F(\mu F)$ and μF, denoted $(\textbf{fix}\ F, \textbf{unfix}\ F)$. Furthermore, by initiality of the algebra, we have a unique algebra morphism from the initial algebra to any other algebra of F, which materialises as folding in the programming language. These can be assembled into the diagram

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{F(\textbf{fold}\ a)} & F(A) \\ {\scriptstyle \textbf{unfix}\ F}\ \uparrow \downarrow\ {\scriptstyle \textbf{fix} F} & & \downarrow {\scriptstyle a} \\ \mu F & \xrightarrow{\textbf{fold}\ a} & A \end{array}\,.$$

If we are to interpret inductive data types, we must be able to interpret this diagram, including the fixpoint maps, initiality maps, and the commutativity of the square.

why?

To do this, we will replace μF with a chosen representation R_F, the fixpoint maps with a pair of maps (**collapse** F, **inspect** F), and the initiality maps with a pair of maps (**wrap** F, **unwrap** F).

3 The transformation

The technique for transforming inductive data types into custom data structures will be phrased in the language of 2-level type theory (2LTT) [15].

why? why this particular language? what does it give you that other ways do not?

3.1 The 2-level type theory $\mathbb{G}$

We will work in a 2LTT which we denote $\mathbb{G}$. The meta fragment of $\mathbb{G}$ contains a universe hierarchy $\mathcal{U}_{\mathbf{Meta},i}$ of meta-level types, and a single universe of values $\mathcal{U}_{\mathbf{Obj}}$. Additionally, the universe $\mathcal{U}_{\mathbf{Meta},0}$ has a subuniverse $\mathcal{U}_{\mathbf{Repr}}$ of "representable" meta-level types. We have

$$\mathcal{U}_{\mathbf{Meta},i} : \mathcal{U}_{\mathbf{Meta},i+1} \qquad \mathcal{U}_{\mathbf{Repr}} : \mathcal{U}_{\mathbf{Meta},1} \qquad \mathcal{U}_{\mathbf{Obj}} : \mathcal{U}_{\mathbf{Meta},0}\,.$$

Any object type A can be lifted to the meta-level as $\Uparrow A : \mathcal{U}_{\mathbf{Repr}}$, and any object term $t : A$ can be lifted as $\langle t \rangle : \Uparrow A$, similarly to the original presentation of 2LTT [15]. Splicing also works in the same way; if $t : \Uparrow A$, then $\sim t : A$. Universe levels will be implicit in the rest of this presentation, as they are orthogonal to its main content.

Please give an example

The universes $\mathcal{U}_{\mathbf{Repr}}$ and $\mathcal{U}_{\mathbf{Obj}}$ are closed under simple Σ and Π types, and the universe $\mathcal{U}_{\mathbf{Meta}}$ is closed under dependent Σ and Π types.

why? also are you considering DTs as the meta language here? Why pi types? The languages need defined or at least the assumptions about them need to be given

The *categories with families* interpretation of $\mathbb{G}$ is given by the symbols $\mathsf{Ty}_{\mathbb{G},m}$, $\mathsf{Con}_{\mathbb{G}}$, $\mathsf{Sub}_{\mathbb{G}}$, and $\mathsf{Tm}_{\mathbb{G},m}$, where m ranges over the two stages **Meta** and **Obj**.

Is it fine to have a sort for $\mathcal{U}_{\mathbf{Repr}}$?

3.2 Inductive data types

We allow inductive data types to be defined in the meta fragment of $\mathbb{G}$, as inductive families. These exist as first-class citizens, and we follow the syntactical approach of [11] ...

We have constructors and eliminators for inductive types in the meta fragment.

...

3.3 Global section of representations

Our goal is to translate a program that lives in the $\mathcal{U}_{\mathbf{Repr}}$ subuniverse, into an object program, i.e. a program that lives in the $\mathcal{U}_{\mathbf{Obj}}$ universe.

A type in the $\mathcal{U}_{\mathbf{Repr}}$ universe needs to have a corresponding "representation" object type. This means that we require a function of the form

$$\sigma : \mathcal{U}_{\mathbf{Repr}} \to \mathcal{U}_{\mathbf{Obj}}$$

which assigns to each meta type a corresponding object type. This is in some sense a "global section" of representations, because for each representable type, we pick an object type to represent it with, among a set of candidate object types.

Give an example of such a choice of representation

Having defined such a function σ, now we need to define how to translate a term of type A in the $\mathcal{U}_{\mathbf{Repr}}$ universe to a term of type σA in the $\mathcal{U}_{\mathbf{Obj}}$ universe. This is done through the `Repr` meta type, defined as

$$\begin{aligned}&\textbf{record}\ \mathtt{Repr}\,(\sigma : \mathcal{U}_{\mathbf{Repr}} \to \mathcal{U}_{\mathbf{Obj}})\,(A : \mathcal{U}_{\mathbf{Repr}})\ \textbf{where}\\ &\quad \mathtt{c} : A[\sigma] \to \mathtt{Gen}\Uparrow\sigma A\\ &\quad \mathtt{i} : \Uparrow\sigma A \to \mathtt{Gen}\,A(\sigma)\end{aligned}$$

which defines a representation to be a pair of functions `c` and `i`. The function `c` is used to convert a partial syntactical representation of a term of the representable type $A[\sigma]$ into a value of its representation type σA. Similarly, the function `i` is used to convert a value of the representation type σA into a partial syntactical representation of a term of the representable type $A(\sigma)$.

In order to understand the meaning of this type, we first need to define the types $A[\sigma]$ and $A(\sigma)$. These are primitive types that exist within $\mathcal{U}_{\mathbf{Meta}}$, and are defined inductively over the structure of types in $\mathcal{U}_{\mathbf{Repr}}$. The type $A[\sigma]$ is defined as

$$\begin{aligned}(A \to B)[\sigma] &= \Uparrow\sigma A \to \Uparrow\sigma B\\ (A \times B)[\sigma] &= \Uparrow\sigma A \times \Uparrow\sigma B\\ (1)[\sigma] &= 1\\ (F : \mathcal{U}_{\mathbf{Repr}}, (c_i : N_i \to (F)_{j\in J_i} \to F)_{i\in I})[\sigma] &= (F_\sigma : \mathcal{U}_{\mathbf{Repr}}, (c_i : N_i[\sigma] \to (F_\sigma)_{j\in J_i} \to F_\sigma)_{i\in I}, ? : \Uparrow\sigma F \to F_\sigma)\,,\end{aligned}$$

and the type $A(\sigma)$ is defined as

$$\begin{aligned}(A \to B)(\sigma) &= \Uparrow\sigma A \to \Uparrow\sigma B\\ (A \times B)(\sigma) &= \Uparrow\sigma A \times \Uparrow\sigma B\\ (1)(\sigma) &= 1\\ (F : \mathcal{U}_{\mathbf{Repr}}, (c_i : N_i \to (F)_{j\in J_i} \to F)_{i\in I})(\sigma) &= (F^\sigma : \mathcal{U}_{\mathbf{Repr}}, (c_i : \Uparrow\sigma N_i \to (F^\sigma)_{j\in J_i} \to F^\sigma)_{i\in I})\,.\end{aligned}$$

The return type of the collapsing and inspecting functions is over the monad `Gen`. This is the code generation monad, first described in [Kovacs unpublished], which is defined as

$$\textbf{data}\ \mathtt{Gen}\,(A : \mathcal{U}_{\mathbf{Meta}}) = \mathtt{unGen}\,(\{R : \mathcal{U}_{\mathbf{Obj}}\} \to (A \to \Uparrow R) \to \Uparrow R)$$

Explain what $A[\sigma]$ and $A(\sigma)$ are supposed to represent

Surely this means that $\mathcal{U}_{\mathbf{Repr}}$ is a closed universe so that we can define σ by induction..

We can similarly package choice of `Repr` as a global section

$$\tau : \prod_{A:\mathcal{U}_{\mathbf{Repr}}} \mathtt{Repr}\,\sigma\,A$$

3.4 Specialising functions

$$\textsc{Spec-Intro}\ \frac{\Gamma \vdash A_R : \mathcal{U}_{\mathbf{Repr}} \quad \Gamma \vdash B_S : \mathcal{U}_{\mathbf{Repr}} \quad \Gamma, x : A_R \vdash b : B_S \quad \Gamma \vdash r : \Uparrow R.\mathtt{R} \to \Uparrow S.\mathtt{R}}{\Gamma \vdash \lambda_r x.\, b : A_R \to B_R}\,.$$

3.5 Translating $\mathbb{G}_{\mathbf{Repr}}$ to $\mathbb{G}_{\mathbf{Val}}$

The CWF defined by $\mathbb{G}$ is contextual [10], which means that the contexts are inductively defined as dependent lists of types $\mathsf{Ty}_{\mathbb{G}}$. Therefore, we can define restrictions of the objects $\mathsf{Con}_{\mathbb{G}}$ and morphisms $\mathsf{Sub}_{\mathbb{G}}$ to only contain representable inductive types, which we call $\mathsf{Con}_{\mathbb{G},\mathbf{Repr}}$ and $\mathsf{Sub}_{\mathbb{G},\mathbf{Repr}}$. This makes

$$(\mathsf{Con}_{\mathbb{G},\mathbf{Repr}}, \mathsf{Sub}_{\mathbb{G},\mathbf{Repr}}, \mathsf{Ty}_{\mathbb{G},\mathbf{Repr}}, \mathsf{Tm}_{\mathbb{G},\mathbf{Repr}})$$

into a syntactical (initial) CWF which we call $\mathbb{G}_{\mathbf{Repr}}$ (Proof?). We can perform a similar restriction to no inductive types at all, to obtain

$$(\mathsf{Con}_{\mathbb{G},\mathbf{Val}}, \mathsf{Sub}_{\mathbb{G},\mathbf{Val}}, \mathsf{Ty}_{\mathbb{G},\mathbf{Val}}, \mathsf{Tm}_{\mathbb{G},\mathbf{Val}})$$

which is also a syntactical CWF that we call $\mathbb{G}_{\mathbf{Val}}$ (Proof?).

The translation of a representable meta-program into a valuated meta-program is done through a syntactical CWF morphism

$$\mathsf{T}_{\sigma,\tau} : \mathbb{G}_{\mathbf{Repr}} \longrightarrow \mathbb{G}_{\mathbf{Val}}$$

that is defined for a given representation choice σ. We exploit the initiality of the CWF $\mathbb{G}_{\mathbf{Repr}}$ to define the translation inductively over its syntax:

$$\begin{aligned}
&\mathsf{T} : \mathsf{Con}_{\mathbb{G},\mathbf{Repr}} \to \mathsf{Con}_{\mathbb{G},\mathbf{Val}} \\
&\mathsf{T}(\cdot) = \cdot \\
&\mathsf{T}(\Gamma, A) = \mathsf{T}(\Gamma), \mathsf{T}(A)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{T} : \mathsf{Sub}_{\mathbb{G},\mathbf{Repr}}\ \Gamma\ \Delta \to \mathsf{Sub}_{\mathbb{G},\mathbf{Val}}\ \mathsf{T}\Gamma\ \mathsf{T}\Delta \\
&\mathsf{T}(\mathbf{id}) = \mathbf{id} \\
&\mathsf{T}(\gamma, f) = \mathsf{T}\gamma, \mathsf{T}f
\end{aligned}$$

$$\begin{aligned}
&\mathsf{T} : \mathsf{Ty}_{\mathbb{G},\mathbf{Repr}}\ \Gamma \to \mathsf{Ty}_{\mathbb{G},\mathbf{Val}}\ \mathsf{T}\Gamma \\
&\mathsf{T}(\Pi x : A.\, B) = \Pi x : \mathsf{T}A.\mathsf{T}B \\
&\mathsf{T}(\Uparrow V) = \Uparrow V \\
&\mathsf{T}(A) = \Uparrow \sigma A
\end{aligned}$$

$$\begin{aligned}
&\mathsf{T} : \mathsf{Tm}_{\mathbb{G},\mathbf{Repr}}\ \Gamma\ A \to \mathsf{Tm}_{\mathbb{G},\mathbf{Val}}\ \mathsf{T}\Gamma\ \mathsf{T}A \\
&\mathsf{T}(\lambda x.a) = \lambda x.\mathsf{T}a \\
&\mathsf{T}(a\, b) = (\mathsf{T}a)\,(\mathsf{T}b) \\
&\mathsf{T}(x) = x \\
&\mathsf{T}(\langle p \rangle) = \langle p \rangle \\
&\mathsf{T}(c_A \mathbin{\hat{@}} \vec{a} \mathbin{@} \vec{r}) = \texttt{runGen}\ (\tau A.\texttt{c}\ (c_A \mathbin{\hat{@}} \mathsf{S}\vec{a} \mathbin{@} \mathsf{S}\vec{r}))\ \texttt{id} \\
&\mathsf{T}(\mathbf{case}_A\ m\ \vec{a}) = \texttt{runGen}\ (\tau A.\texttt{i}\ \mathsf{T}m)\ (\lambda x.\, \mathbf{case}\ x\ \mathsf{T}\vec{a})
\end{aligned}$$

$$\begin{aligned}
&\mathsf{S} : \mathsf{Tm}_{\mathbb{G},\mathbf{Repr}}\ \Gamma\ A_R \to \mathsf{Tm}_{\mathbb{G},\mathbf{Repr}}\ \mathsf{T}\Gamma\ A[\sigma] \\
&\mathsf{S}(c \mathbin{\hat{@}} \vec{a} \mathbin{@} \vec{r}) = (c \mathbin{\hat{@}} \mathsf{S}\vec{a} \mathbin{@} \mathsf{S}\vec{r}) \\
&\mathsf{S}(a) = ?(\mathsf{T}a)
\end{aligned}$$

Issues with the above:
Coalgebra could be phrased in terms of views for semantic correctness [1,2] ?

4 Properties

5 Related work

6 Conclusions and Future Work

References

[1] Guillaume Allais. Builtin types viewed as inductive families. In *Programming Languages and Systems*, pages 113–139. Springer Nature Switzerland, 2023.

[2] Guillaume Allais. Seamless, correct, and generic programming over serialised data. October 2023.

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

[4] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-Level type theory and applications. May 2017.

[5] Steve Awodey. Natural models of homotopy type theory. June 2014.

[6] Thaïs Baudon, Gabriel Radanne, and Laure Gonnord. Bit-Stealing made legal: Compilation for custom memory representations of algebraic data types. *Proc. ACM Program. Lang.*, 7(ICFP):813–846, August 2023.

[7] S Boulier. Extending type theory with syntactic models. November 2018.

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

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

[10] Simon Castellan, Pierre Clairambault, and Peter Dybjer. Categories with families: Unityped, simply typed, and dependently typed. April 2019.

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

[12] Brandon Hewer and Graham Hutton. Quotient haskell: Lightweight quotient types for all. *Proc. ACM Program. Lang.*, 8(POPL):785–815, January 2024.

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

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

[15] András Kovács. Staged compilation with two-level type theory. September 2022.

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

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

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

[19] Taichi Uemura. A general framework for the semantics of type theory. *arXiv [math.CT]*, April 2019.

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

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

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