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

The techinque of program staging 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 object-level fragments inside the meta language. ...

1.1 Contributions

TODO

- We present a formalism for the expression of a choice of representation for inductive data types.
- 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.
- We show semantic preservation of the entire transformation modulo its preservation by each chosen representation.

2 Examples and technique

The type of natural numbers is an example of a ubiquitous inductive data type that is used extensively in theorem proving and general functional programming, 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

... 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

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

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. 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):

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

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

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

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.

2.1 Representations of inductive types

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

$$\begin{tikzcd} F(\mu F) \arrow[r, "F(\textbf{fold}\ a)"] \arrow[d, shift left, "\textbf{fix} F"] & F(A) \arrow[d, "a"] \\ \mu F \arrow[u, bend left, "\textbf{unfix}\ F"] \arrow[r, "\textbf{fold}\ a"] & A \end{tikzcd} .$$

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.

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

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].

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 $\mathcal{U}_{\textbf{Meta},i}$ of meta-level types, and a single universe of values $\mathcal{U}_{\textbf{Val}}$. Additionally, the universe $\mathcal{U}_{\textbf{Meta},0}$ has a subuniverse $\mathcal{U}_{\textbf{Repr}}$ of "representable" meta-level types. We have

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

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.

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

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 **Val**.

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] ...

The sort for meta inductive types is $\mathcal{U}_{\mathbf{Ind}}$, which is a subuniverse of $\mathcal{U}_{\mathbf{Meta}}$. This sort interprets signatures $(A : \mathsf{Ty}_s, \mathsf{Con}_s\, A)$.

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

...

3.3 Choice of representations

Representable meta types, or inhabitants of $\mathcal{U}_{\mathbf{Repr}}$, are meta types which can be staged into the object level, but are not necessarily just lifted object types. Indeed, lifting an object type is one way to acquire a representable meta type. However, the more interesting way to do so is to attach a representation to a meta-level type. This is captured by the rule

$$\textsc{Repr-Intro}\ \frac{\Gamma \vdash A : \mathcal{U}_{\mathbf{Ind}} \quad \Gamma \vdash R : \mathtt{Repr}\, A}{\Gamma \vdash A_R : \mathcal{U}_{\mathbf{Repr}}}\ .$$

In other words, any inductive type A paired with a representation R is a representable meta type. The type family `Repr` is defined as

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

It defines a representation to be a triple of an object type R_A, a *collapsing* function c_A, and an *inspecting* function i_A.

The notation $A[N]$ is shorthand for $\mathtt{Syntax}\, A\, N$, where the `Syntax` type family is defined by

$$\textbf{data}\ \mathtt{Syntax}\,(A : \mathcal{U}_{\mathbf{Ind}})\,(N : \mathcal{U}_{\mathbf{Meta}}) = \mathtt{known}\,(A\,(\mathtt{Syntax}\, A\, N)) \mid \mathtt{opaque}\, N\,.$$

where $A(B)$ is the endofunctor associated with the inductive type A applied to the meta type B (need rules for that too).

The collapsing function is used to convert a partial syntactical representation of a term of the inductive type μA into a value of a chosen representation type R_A.

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

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

The inspecting function is used to convert a value of the chosen representation type R_A into a partial syntactical representation of a term of the inductive type A.

The subuniverse $\mathcal{U}_{\mathbf{Repr}}$ can be reflected into the metatheory as a structure over the base category $(\mathsf{Sub}_{\mathbb{G}}, \mathsf{Con}_{\mathbb{G}})$. The set $\mathsf{Ty}_{\mathbb{G},\mathbf{Repr}}\,\Gamma$ is the set of representable types in representable context Γ, defined as $\mathsf{Tm}_{\mathbb{G},\mathbf{Meta}}\,\Gamma\,\mathcal{U}_{\mathbf{Repr}}$. Similarly, the set $\mathsf{Tm}_{\mathbb{G},\mathbf{Repr}}\,\Gamma\,A$ is the set of terms of representable type A in context Γ, defined as $\mathsf{Tm}_{\mathbb{G},\mathbf{Meta}}\,\Gamma$ restricted to $\mathsf{Ty}_{\mathbb{G},\mathbf{Repr}}\,\Gamma$.

3.4 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 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 on the object fragment 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 an object program is done through a syntactical CWF morphism

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

We exploit the initiality of the CWF $\mathbb{G}_{\mathbf{Repr}}$ to define the translation inductively over its syntax:

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

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

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

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

$$\begin{aligned}
&\mathsf{S} : \mathsf{Tm}_{\mathbb{G},\mathbf{Repr}}\ \Gamma\ A_R \to \mathsf{Tm}_{\mathbb{G},\mathbf{Repr}}\ \mathsf{T}\Gamma\ A[\Uparrow R.\mathtt{R}]\\
&\mathsf{S}(c_{A_R}\ \hat{@}\ \vec{a}\ @\ \vec{r}) = \texttt{known}\ (c\ \hat{@}\ \mathsf{S}\vec{a}\ @\ \mathsf{S}\vec{r})\\
&\mathsf{S}(a) = \texttt{opaque}\ (\mathsf{T}a)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{V} : \mathsf{Cases}_{\mathbb{G},\mathbf{Repr}}\ \Gamma\ A_R\ C \to \mathsf{Cases}_{\mathbb{G},\mathbf{Repr}}\ \Gamma\ A[\Uparrow R.\mathtt{R}]\ \mathsf{T}C\\
&\mathsf{V}((c_i\ \hat{@}\ \vec{x}_i\ @\ \vec{y}_i \mapsto r_i)_{i\in I}) = (\texttt{known}\ (c_i\ \hat{@}\ \vec{x}_i\ @\ \vec{y}_i \mapsto r_i) \mapsto \mathsf{T}r_i)\ (\texttt{opaque}\ z \mapsto)
\end{aligned}$$

Issues with the above:

- Need to clarify order of execution; The $\mathsf{T}(a)$ in `opaque` $\langle \mathsf{T}(a)\rangle$ should not go back to the constructor case, even if the type is A_R.
- Unclear what happens to the constructor data. [14] leaves non-recursive parameters as metatheoretic sets, which is not good enough! We want these to be actual types in the language!

Coalgebra should be phrased in terms of views for semantic correctness [1,2] ?

4 Properties

5 Related work

6 Conclusion

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.