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

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

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

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

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):

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

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

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

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 $(\text{fix}\ F, \text{unfix}\ F)$. 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

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

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.

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

3 The transformation

An n-level type theory (nLTT), a straightforward generalisation of 2LTT [11], viewed from the perspective of *categories with families* [3], splits the terms and types at each context into distinct fragments corresonding to each of the stages.

More precisely, a model of an n-level type theory consists of the following data:

- A category $\mathbb{C}$ with a terminal object, with objects denoted $\text{Con}_{\mathbb{C}}$ and morphisms denoted $\text{Sub}_{\mathbb{C}}$.
- Two families of n presheaves, $\text{Ty}_i : \mathbb{C} \longrightarrow \mathbf{Set}$ and $\text{Tm}_i : \mathbb{C} \longrightarrow \mathbf{Set}$.
- A family of n representable natural transformations $\text{typeof}_i : \text{Tm}_i \longrightarrow \text{Ty}_i$.
- A family of $n-1$ natural transformations $\Uparrow_i : \text{Ty}_i \longrightarrow \text{Ty}_{i+1}$ and $(n-1)$ invertible natural transformations $\langle - \rangle_i : \text{Tm}_{i+1} \longrightarrow \text{Tm}_i$ with inverses $\sim_i (-)$. This creates a lifting structure, where terms and types from level i can be lifted to level $i+1$.

The above data can also be formulated using *categories with families* [8], where context comprehension is given by the fact that typeof_i is representable [15].

4 Properties

5 Related work

6 Conclusion

References

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

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

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

[4] 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.

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

[6] 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.

[7] 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.

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

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

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

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

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

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