Representing Inductive Families

Constantine Theocharis[0000−1111−2222−3333] and
Edwin Brady[0000−1111−2222−3333]

University of St Andrews, UK
{kt81,ecb10}@st-andrews.ac.uk

1 Introduction

Inductive data types allow programmers to define data structures in a declarative manner by specifying the all the constructors which can build up the data. Every inductive definition is equipped with an induction principle that captures the notion of mathematical induction over the data, and in particular, enables structural recursion over the data. This is a powerful tool for programming as well as theorem proving. However, this abstraction often comes at a cost; the data representation of inductive types is a linked tree structure. By default, 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. This representation is not always the most efficient for all operations, and often forces users to rely on 'more efficient' machine primitives to achieve good performance. The 'Nat-hack' in languages with dependent types is a prime example of this, where natural numbers are represented as big integers for efficient computation.

In this paper, we propose an extension to a core language with dependent types and inductive families which allows users to define custom representations for inductive types. This is done through a translation of the constructors and eliminators of the inductive type to a concrete implementation, which forms a 'representation'. Representations are defined in the language itself, and come with coherence properties that ensure that the representation is a bijection between the inductive type and its implementation.

1.1 The 'Nat-hack'

In languages with dependent types, such as Agda, Coq, Idris and 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

```
Nat-ind : (P : Nat -> Type) -> P Z -> ((m : Nat) -> P m -> P (S m)) -> (n : Nat) -> P n
```

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

```
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:

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

References

Cockx and AbelCockx and Abel2018. Jesper Cockx and Andreas Abel. 2018. Elaborating dependent (co)pattern matching. *Proc. ACM Program. Lang.* 2, ICFP (July 2018), 1–30. `https://doi.org/10.1145/3236770`

DagandDagand2017. 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`

KovácsKovács2022. András Kovács. 2022. Staged compilation with two-level type theory. (Sept. 2022). arXiv:2209.09729 [cs.PL]