# Custom Representations of Inductive Families

Constantine Theocharis[0009−0001−0198−2750] and
Edwin Brady[0000−0002−9734−367X]

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

**Abstract.** Inductive families provide a convenient way of programming with dependent types. Yet, when it comes to compilation, their default linked-tree runtime representations, as well as the need to convert between different indexed views of the same data when programming with dependent types, can lead to unsatisfactory runtime performance. In this paper, we aim to introduce a language with dependent types, and inductive families with custom representations. Representations are a version of Wadler's views [28], refined to inductive families like in Epigram [26], but with compilation guarantees: a represented inductive family will not leave any runtime traces behind, without relying on heuristics such as deforestation. This way, we can build a library of convenient inductive families based on a minimal set of primitives, whose re-indexing and conversion functions are erased at compile-time. We show how we can express inductive data optimisation techniques, such as representing Nat-like types as GMP-style big integers, without special casing in the compiler. With dependent types, reasoning about data representations is also possible: we get computationally irrelevant isomorphisms between the original and represented data.

**Keywords:** Dependent types · Memory representation · Inductive families

## 1 Introduction

Inductive families are a generalisation of inductive data types found in some programming languages with dependent types. Every inductive definition is equipped with an eliminator 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 has a cost when it comes to compilation: the runtime representation of inductive types is a linked tree structure. This representation is not always the most efficient for all operations, and often forces users to rely on more efficient machine primitives to achieve desirable performance, at the cost of structural recursion and dependent pattern matching. This is the first problem we aim to address in this paper.

Despite advances in the erasure of irrelevant indices in inductive families [14] and the use of theories with irrelevant fragments [10,27], there is still a need to convert between different indexed views of the same data. For example, the function to convert from $\mathsf{BinTree}\ T$ to $\mathsf{BinTreeOfHeight}\ T\ n$ by forgetting the size index $n$ is *not* erased by any current language with dependent types, unless sized binary trees are defined as a refinement of binary trees with an erased height field (which hinders dependent pattern matching due to the presence of non-structural witnesses), or a Church encoding is used in a Curry-style context [20] (which restricts the flexibility of data representation). This is the second problem we aim to address in this paper.

Wadler's views [28] provide a way to abstract over inductive interfaces, so that different views of the same data can be defined and converted between seamlessly. In the context of inductive families, views have been used in Epigram [26] that utilise the index refinement machinery of dependent pattern matching to avoid certain proof obligations with eliminator-like constructs. While views exhibit a nice way to transport across a bijection between the original data and the viewed data, they do not utilise this bijection to erase the view from the program.

In this paper, we propose an extension $\lambda_{\text{REP}}$ to a core language with dependent types and inductive families $\lambda_{\text{IND}}$, which allows programmers to define custom, correct-by-construction data representations. This is done through user-defined translations of the constructors and eliminators of an inductive type to a concrete implementation, which form a bijective view of the original data called a 'representation'. Representations are defined internally to the language, and require coherence properties that ensure a representation is faithful to its the original inductive family. We contribute the following:

- A formulation of common optimisations such as the 'Nat-hack', and similarly for other inductive types, as well as zero-cost data reuse when reindexing, using representations (section 2).
- A dependent type system with inductive families $\lambda_{\text{IND}}$ and its extension by representations $\lambda_{\text{REP}}$ formulated as inductive algebras for theories, along with a translation from $\lambda_{\text{IND}}$ that erases all represented data types from the program (section 3).
- A prototype implementation of this system in Superfluid, a programming language with inductive types and dependent pattern matching (section 4).

## 2 A tour of data representations

A common optimisation done by programming languages with dependent types such as Idris 2 and Lean is to represent natural numbers as GMP-style big integers. The definition of natural numbers looks like

$$\textbf{data}\ \mathsf{Nat} \begin{Bmatrix} \mathsf{0} : \mathsf{Nat} \\ \mathsf{1+} : \mathsf{Nat} \to \mathsf{Nat} \end{Bmatrix} \tag{1}$$

and generates a Peano-style induction principle $\mathsf{elim}_{\mathsf{Nat}}$ of type[1]

$$(P : \mathsf{Nat} \to \mathcal{U}) \to P\ 0 \to ((n : \mathsf{Nat}) \to \overline{P\ n} \to P\ (1{+}\ n)) \to (s : \mathsf{Nat}) \to P\ s\,.$$

Without further intervention, the Nat type is represented in unary form, where each digit becomes an empty heap cell at runtime. This is inefficient for a lot of the basic operations on natural numbers, especially since computers are particularly well-equipped to deal with numbers natively, so many real-world implementations will treat Nat specially, swapping the default inductive type representation with one based on GMP integers. This is done by performing the replacements

$$|0| = 0 \tag{2}$$

$$|1{+}| = 1\ + \tag{3}$$

$$|\mathsf{elim}_{\mathsf{Nat}}\ P\ m_0\ m_{1+}\ s| = \mathsf{ubig\text{-}elim}\ |s|\ |m_0|\ |m_{1+}| \tag{4}$$

where $|\cdot|$ denotes a source translation into a compilation target language with primitives $\mathsf{ubig\text{-}}*$.[2]

In addition to the constructors and eliminators, the compiler might also define translations for commonly used definitions which have a more efficient counterpart in the target, such as recursively-defined addition, multiplication, etc. The recursively-defined functions are well-suited to proofs and reasoning, while the GMP primitives are more efficient for computation.

The issue with this approach is that it only works for the data types which the compiler can recognise as special. Particularly in the presence of dependent types, other data types might end up being equivalent to Nat or another 'nicely-representable' type, but in a non-trivial way that the compiler cannot recognise. Hence, one of our goals is to extend this optimisation to work for any data type. To achieve this this, our framework requires that representations are fully typed in a way that ensures the behaviour of the representation of a data type matches the behaviour of the data type itself.

### 2.1 The well-typed Nat-hack

A representation definition looks like

$$\textbf{repr}\ \mathsf{Nat}\ \textbf{as}\ \mathsf{UBig} \left\{ \begin{array}{r l} 0\ \textbf{as} & 0 \\ 1{+}\ n\ \textbf{as} & 1 + n \\ \mathsf{elim}_{\mathsf{Nat}}\ \textbf{as} & \mathsf{ubig\text{-}elim} \\ \textbf{by} & \mathsf{ubig\text{-}elim\text{-}zero\text{-}id}, \\ & \mathsf{ubig\text{-}elim\text{-}add\text{-}one\text{-}id} \end{array} \right\}$$

[1] Recursive parameters like $\overline{P\ n}$ are lazy, which makes the eliminator more efficient when they are not used.

[2] Idris 2 will in fact look for any 'Nat-like' types and apply this optimisation. A Nat-like type is any type with two constructors, one with arity zero and the other with arity one. A similar optimisation is also done with list-like and boolean-like types because they have a canonical representation in the target runtime, Chez Scheme.

$\mathsf{Nat}$ is represented as the type $\mathsf{UBig}$ of GMP-style unlimited-size unsigned integers, with translations for the constructors $\mathsf{0}$ and $\mathsf{1+}$, and the eliminator $\mathsf{elim}_{\mathsf{Nat}}$. Additionally, the eliminator satisfies the expected computation rules of the $\mathsf{Nat}$ eliminator, which are postulated as propositional equalities. This representation is valid in a context containing the primitives

$$\begin{aligned} \mathsf{0}, \mathsf{1} : \mathsf{UBig} \qquad &+, \times : \mathsf{UBig} \to \mathsf{UBig} \to \mathsf{UBig} \\ \mathsf{ubig\text{-}elim} : (P : \mathsf{UBig} \to \mathcal{U}) &\to P\ \mathsf{0} \to ((n : \mathsf{UBig}) \to \overline{P\ n} \to P\ (\mathsf{1} + n)) \\ &\to (s : \mathsf{UBig}) \to P\ s \end{aligned}$$

and propositional equalities

$$\begin{aligned} &\mathsf{ubig\text{-}elim\text{-}zero\text{-}id} :_{\forall P b r}\ \mathsf{ubig\text{-}elim}\ P\ b\ r\ \mathsf{0} = b \\ &\mathsf{ubig\text{-}elim\text{-}add\text{-}one\text{-}id} :_{\forall P b r n}\ \mathsf{ubig\text{-}elim}\ P\ b\ r\ (\mathsf{1} + n) = r\ n\ (\lambda\_.\ \mathsf{ubig\text{-}elim}\ P\ b\ r\ n)\,. \end{aligned}$$

Representations can also be defined for functions on $\mathsf{Nat}$, such as addition, multiplication, and other numeric operations, in terms of $\mathsf{UBig}$ primitives.

$$\textbf{repr}\ \mathsf{add}\ \textbf{as}\ +\ \textbf{by}\ +\text{-}\mathsf{id} \qquad \textbf{repr}\ \mathsf{mul}\ \textbf{as}\ \times\ \textbf{by}\ \times\text{-}\mathsf{id}$$

These will be replaced during a translation process back to $\lambda_{\text{IND}}$, like rewriting rules [17], given that we have the appropriate lemmas to justify them in the context.

This will effectively erase the $\mathsf{Nat}$ type from the compiled program, replacing all occurrences with the $\mathsf{UBig}$ type and its primitives. In a way, the hard work is done by the postulates above; we expect that the underlying implementation of $\mathsf{UBig}$ indeed satisfies them, which is a separate concern from the correctness of the representation itself. However, postulates are only needed when the representation target is a primitive; the next examples use defined types as targets, so that the coherence of the target eliminator follows from the coherence of other eliminators used in its implementation.

### 2.2 Vectors are just certain lists

In addition to representing inductive types as primitives, we can use representations to share the underlying data when converting between indexed views of the same data. For example, we can define a representation of $\mathsf{Vec}$ in terms of $\mathsf{List}$, so that the conversion from one to the other is 'compiled away'. We can do this by representing the indexed type as a refinement of the unindexed type by an appropriate relation. For the case of $\mathsf{Vec}$, we know intuitively that

$$\mathsf{Vec}\ T\ n \simeq \{l : \mathsf{List}\ T \mid \mathsf{length}\ l = n\} := \mathsf{List'}\ T\ n$$

so we can start by choosing $\mathsf{List'}\ T\ n$ as the representation of $\mathsf{Vec}\ T\ n$.[3] We are then tasked with providing terms that correspond to the constructors of $\mathsf{Vec}$ but

[3] We will take the subset $\{x : A \mid P\ x\}$ to mean a $\Sigma$-type $(x : A) \times P\ x$ where the right component is irrelevant and erased at runtime.

for $\mathsf{List}'$. These can be defined as

$$\begin{aligned} &\mathsf{nil} : \mathsf{List}'\ T\ 0 && \mathsf{cons} : T \to \mathsf{List}'\ T\ n \to \mathsf{List}'\ T\ (\mathsf{1+}\ n) \\ &\mathsf{nil} = (\mathsf{nil}, \mathsf{refl}) && \mathsf{cons}\ x\ (xs, p) = (\mathsf{cons}\ x\ xs, \mathsf{cong}\ (\mathsf{1+})\ p) \end{aligned}$$

Next we need to define the eliminator for $\mathsf{List}'$, which should have the form

$$\begin{aligned} &\mathsf{elim\text{-}List}' : (E : (n : \mathsf{Nat}) \to \mathsf{List}'\ T\ n \to \mathsf{Type}) \\ &\quad \to E\ 0\ \mathsf{nil} \\ &\quad \to ((x : T) \to (n : \mathsf{Nat}) \to (xs : \mathsf{List}'\ T\ n) \to \overline{E\ n\ xs} \to E\ (\mathsf{1+}\ n)\ (\mathsf{cons}\ x\ xs)) \\ &\quad \to (n : \mathsf{Nat}) \to (v : \mathsf{List}'\ T\ n) \to E\ n\ v \end{aligned}$$

Dependent pattern matching does a lot of the heavy lifting by refining the length index and equality proof by matching on the underlying list. However we still need to substitute the lemma $\mathsf{cong}\ (\mathsf{1+})\ (\mathsf{1+\text{-}inj}\ p) = p$ in the recursive case.

$$\begin{aligned} &\mathsf{elim\text{-}List}'\ P\ b\ r\ 0\ (\mathsf{nil}, \mathsf{refl}) = b \\ &\mathsf{elim\text{-}List}'\ P\ b\ r\ (\mathsf{1+}\ m)\ (\mathsf{cons}\ x\ xs, e) = \mathsf{subst}\ (\lambda p.\ P\ (\mathsf{1+}\ m)\ (\mathsf{cons}\ x\ xs, p)) \\ &\qquad\qquad (\mathsf{1+\text{-}cong\text{-}id}\ e)\ (r\ x\ (xs, \mathsf{1+\text{-}inj}\ e) \\ &\qquad\qquad (\lambda\_.\ \mathsf{elim\text{-}List}'\ P\ b\ r\ m\ (xs, \mathsf{1+\text{-}inj}\ e))) \end{aligned}$$

Finally, we need to prove that the eliminator satisfies the expected computation rules propositionally. These are

$$\mathsf{elim\text{-}List'\text{-}nil\text{-}id} : \mathsf{elim\text{-}List}'\ P\ b\ r\ 0\ (\mathsf{nil}, \mathsf{refl}) = b$$

$$\begin{aligned} \mathsf{elim\text{-}List'\text{-}cons\text{-}id} : {} & \mathsf{elim\text{-}List}'\ P\ b\ r\ (\mathsf{1+}\ m)\ (\mathsf{cons}\ x\ xs, \mathsf{cong}\ (\mathsf{1+})\ p) \\ & = r\ x\ (xs, p)\ (\lambda\_.\ \mathsf{elim\text{-}List}'\ P\ b\ r\ m\ (xs, p)) \end{aligned}$$

which we leave as an exercise, though they are evident from the definition of $\mathsf{elim\text{-}List}'$. This completes the definition of the representation of $\mathsf{Vec}$ as $\mathsf{List}'$, which would be written as

$$\textbf{repr}\ \mathsf{Vec}\ T\ n\ \textbf{as}\ \mathsf{List}'\ T\ n \left\{ \begin{array}{r l} \mathsf{nil}\ \textbf{as} & \mathsf{nil} \\ \mathsf{cons}\ \textbf{as} & \mathsf{cons} \\ \mathsf{elim}_{\mathsf{Vec}}\ \textbf{as} & \mathsf{elim\text{-}List}' \\ \textbf{by} & \mathsf{elim\text{-}List'\text{-}nil\text{-}id}, \\ & \mathsf{elim\text{-}List'\text{-}cons\text{-}id} \end{array} \right\}$$

Now the hard work is done; Every time we are working with a $v : \mathsf{Vec}\ T\ n$, its form will be $(l, p)$ at runtime, where $l$ is the underlying list and $p$ is the proof that the length of $l$ is $n$. Under the assumption that the $\Sigma$-type's right component is irrelevant and erased at runtime, every vector is simply a list at runtime, where the length proof has been erased. In practice, this erasure is achieved in SUPERFLUID using quantitative type theory [10]. In section 3.8 we show how to formally identify computationally irrelevant conversion functions.

We can utilise this representation to convert between $\mathsf{Vec}$ and $\mathsf{List}$ at zero runtime cost, by using the **repr** and **unrepr** operators of the language (defined in section 3). Specifically, we can define the functions

$$\begin{aligned}
&\mathsf{forget\text{-}length} : \mathsf{Vec}\ T\ n \to \mathsf{List}\ T\\
&\mathsf{forget\text{-}length}\ v = \mathbf{let}\ (l, _) = \mathbf{repr}\ v\ \mathbf{in}\ l\\[1em]
&\mathsf{recall\text{-}length} : (l : \mathsf{List}\ T) \to \mathsf{Vec}\ T\ (\mathsf{length}\ l)\\
&\mathsf{recall\text{-}length}\ l = \mathbf{unrepr}\ (l, \mathsf{refl})
\end{aligned}$$

and in section 3.8 we will show that it holds by reflexivity that such functions are inverses of one another.

### 2.3 General reindexing

The idea from the previous example can be generalised to any data type. In general, suppose that we have two inductive families

$$\mathsf{F} : P \to \mathcal{U} \qquad \mathsf{G} : P \to X\ p \to \mathcal{U}$$

for some index family $X : P \to \mathcal{U}$. If we hope to represent $\mathsf{G}$ as some refinement of $\mathsf{F}$ then we must be able to provide a way to compute $\mathsf{G}$'s extra indices $X$ from $\mathsf{F}$, like we computed $\mathsf{Vec}$'s extra $\mathsf{Nat}$ index from $\mathsf{List}$ with $\mathsf{length}$ in the previous example. This means that we need to provide a function $\mathsf{comp} :_{\forall p} \mathsf{F}\ p \to X\ p$ which can then be used to form the family

$$\mathsf{F}^{\mathsf{comp}}\ p\ x := \{f : \mathsf{F}\ p \mid \mathsf{comp}\ f = x\}.$$

If $\mathsf{G}$ is 'equivalent' to the algebraic ornament of $\mathsf{F}$ by the algebra defining $\mathsf{comp}$ (given by an isomorphism between the underlying polynomial functors), then it is also equivalent to the $\Sigma$-type above. The 'recomputation lemma' of algebraic ornaments [19] then arises from its projections. Our system allows us to *set* the representation of $\mathsf{G}$ as $\mathsf{F}^{\mathsf{comp}}$, so that the forgetful map from $\mathsf{G}$ to $\mathsf{F}$ is the identity at runtime.

### 2.4 Zero-copy deserialisation

The machinery of representations can be used to implement zero-copy deserialisation of data formats into inductive types. For example, consider the following record for a player in a game:

$$\mathbf{data}\ \mathsf{Player}\left\{\begin{aligned}
\mathsf{player} : &\ (\mathsf{position} : \mathsf{Position})\\
&\to (\mathsf{direction} : \mathsf{Direction})\\
&\to (\mathsf{items} : \mathsf{Fin}\ \mathsf{MAX_INVENTORY})\\
&\to (\mathsf{inventory} : \mathsf{Inventory}\ (\mathsf{fin\text{-}to\text{-}nat}\ \mathsf{items})) \to \mathsf{Player}
\end{aligned}\right\}$$

We can use the $\mathsf{Fin}$ type to maintain the invariant that the inventory has a maximum size. Additionally, we can index the $\mathsf{Inventory}$ type by the number of items it contains, which might be defined similarly to $\mathsf{Vec}$:

$$\textbf{data}\ \mathsf{Inventory}\ (n : \mathsf{Nat}) \left\{ \begin{array}{r} \mathsf{empty} : \mathsf{Inventory}\ 0 \\ \mathsf{add} : \mathsf{Item} \to \mathsf{Inventory}\ n \to \mathsf{Inventory}\ (1+\ n) \end{array} \right\}$$

We can use the full power of inductive families to model the domain of our problem in the way that is most convenient for us. If we were writing this in a lower-level language, we might choose to use the serialised format directly when manipulating the data, relying on the appropriate pointer arithmetic to access the fields of the serialised data, to avoid copying overhead. Representations allow us to do this while still being able to work with the high-level inductive type.

We can define a representation for $\mathsf{Player}$ as a pair of a byte buffer and a proof that the byte buffer contents correspond to a player record. Similarly, we can define a representation for $\mathsf{Inventory}$ as a pair of a byte buffer and a proof that the byte buffer contents correspond to an inventory record of a certain size. By the implementation of the eliminator in the representation, the projection $\mathsf{inventory} : (p : \mathsf{Player}) \to \mathsf{Inventory}\ p.\mathsf{items}$ is compiled into some code to slice into the inventory part of the player's byte buffer. We assume that the standard library already represents $\mathsf{Fin}$ in the same way as $\mathsf{Nat}$, so that reading the $\mathsf{items}$ field is a constant-time operation (we do not need to build a unary numeral). We can thus define the representation of $\mathsf{Player}$ as

$$\textbf{repr}\ \mathsf{Player}\ \textbf{as}\ \{\mathsf{Buf} \mid \mathsf{IsPlayer}\} \left\{ \begin{array}{r} \mathsf{player}\ \textbf{as}\ \mathsf{buf\text{-}is\text{-}player} \\ \mathsf{elim}_{\mathsf{Player}}\ \textbf{as}\ \mathsf{elim\text{-}buf\text{-}is\text{-}player} \\ \textbf{by}\ \mathsf{elim\text{-}buf\text{-}is\text{-}player\text{-}id} \end{array} \right\}$$

with an appropriate definition of $\mathsf{IsPlayer}$ which refines a byte buffer. The refinement would have to match the expected structure of the byte buffer, so that all the required fields can be extracted. Allais [7] explores how data descriptions that index into a flat buffer could be defined.

### 2.5 Transitivity

Representations are transitive, so in the previous example, the 'terminal' representation of $\mathsf{Vec}$ also depends on the representation of $\mathsf{List}$. It is possible to define a custom representation for $\mathsf{List}$ itself, for example a heap-backed array or a finger tree, and $\mathsf{Vec}$ would inherit this representation. However it will still be the case that $\textbf{Repr}\ (\mathsf{Vec}\ T\ n) \equiv \mathsf{List}\ T$, which means the $\textbf{repr}/\textbf{Repr}$ operators only look at the immediate representation of a term, not its terminal representation. Regardless, we can construct predicates that find types which satisfy a certain 'eventual' representation. For example, given a $\mathsf{Buf}$ type of byte buffers, we can consider the set of all types which are eventually represented as a $\mathsf{Buf}$:

$$\textbf{data}\ \mathsf{ReprBuf}\ (T : \mathcal{U}) \left\{ \begin{array}{r} \mathsf{buf} : \mathsf{ReprBuf}\ \mathsf{Buf} \\ \mathsf{from} : \mathsf{ReprBuf}\ (\textbf{Repr}\ T) \to \mathsf{ReprBuf}\ T \\ \mathsf{refined} : \mathsf{ReprBuf}\ T \to \mathsf{ReprBuf}\ \{t : T \mid P\ t\} \end{array} \right\}$$

Every such type comes with a projection function to the $\mathsf{Buf}$ type

$$
\begin{aligned}
&\mathsf{as\text{-}buf} : \{r : \mathsf{ReprBuf}\ T\} \to T \to \mathsf{Buf}\\
&\mathsf{as\text{-}buf}\ \{r = \mathsf{buf}\}\ x = x\\
&\mathsf{as\text{-}buf}\ \{r = \mathsf{from}\ t\}\ x = \mathsf{as\text{-}buf}\ t\ (\mathbf{repr}\ x)\\
&\mathsf{as\text{-}buf}\ \{r = \mathsf{refined}\ t\}\ (x, \_) = \mathsf{as\text{-}buf}\ t\ x
\end{aligned}
$$

which eventually computes to the identity function after applying **repr** the appropriate amount of times. Upon compilation, every type is converted to its terminal representation, and all **repr** calls are erased, so the as-buf function is effectively the identity function at runtime.[4]

# 3 A type system for data representations

In this section, we will develop an extension of dependent type theory with inductive families and custom data representations. We start in section 3.2 by exploring the semantics of data representations in terms of algebras for signatures. In section 3.4 we define a core language with inductive families $\lambda_{\text{IND}}$. In section 3.5, we extend this language with data representations to form $\lambda_{\text{REP}}$. All of the examples in the paper are written in a surface language that elaborates to $\lambda_{\text{REP}}$. The dependent pattern matching can be elaborated to internal eliminators by standard techniques [21,16]. The $\lambda_{\text{IND}}/\lambda_{\text{REP}}$ languages use eliminators instead of pattern matching.

We work in an extensional metatheory with a small universe **Set**, $(a : A) \times B$ for dependent pairs, $(a : A) \to B$ for dependent products, and $=$ for equality, and an Agda-like syntax. The metatheory also supports quotient-inductive-inductive definitions, which are used to define the syntaxes of the languages presented in this paper in the style of Kaposi and Altenkirch [9]. Weakening of terms in defined object theories is generally also left implicit to reduce syntactic noise, and often higher-order abstract syntax notation is used when de-Brujin indices are implied. We use @ for 'partial' object applications as the inverse of $\lambda$: applying $f$ to $x$ is $(f@)[\langle x\rangle]$. We use $\Gamma : \mathsf{Con}$ for contexts, $A : \mathsf{Ty}\ \Gamma$ for types, $a : \mathsf{Tm}\ \Gamma\ A$ for terms, $\Delta : \mathsf{Tel}\ \Gamma$ for telescopes and $(a, b, \ldots) : \mathsf{Tms}\ \Gamma\ \Delta$ for lists of terms. Empty contexts and telescopes are $\bullet$, and extension is $\triangleright$.

## 3.1 Algebraic signatures

A representation of a data type must be able to emulate the behaviour of the original data type. In turn, the behaviour of the original data type is determined by its elimination, or induction principle. This means that a representation of a data type should provide an implementation of induction of the same 'shape' as the original. Induction can be characterised in terms of algebras and displayed algebras of algebraic signatures.

[4] Given that the $r$ argument is known at compile-time and monomorphised.

Algebraic signatures [5,24] consist of a list of operations, each with a specified arity. There are many flavours of algebraic signatures with varying degrees of expressiveness. For this paper, we are interested in algebraic signatures which can be used as a syntax for defining inductive families in a object theory. Thus, we define

$$
\begin{array}{ll}
\mathsf{Sig} : (\Gamma : \mathsf{Con}) \to \mathsf{Tel}\ \Gamma \to \mathbf{Set} & \mathsf{Op} : (\Gamma : \mathsf{Con}) \to \mathsf{Tel}\ \Gamma \to \mathbf{Set} \\
\bullet : \mathsf{Sig}\ \Gamma\ P & \Pi : (A : \mathsf{Ty}\ \Gamma) \to \mathsf{Op}\ (\Gamma \rhd A)\ P \to \mathsf{Op}\ \Gamma\ P \\
\rhd : (T : \mathsf{Sig}\ P) \to \mathsf{Op}\ P \to \mathsf{Sig}\ P & \Pi\iota : \mathsf{Tms}\ \Gamma\ P \to \mathsf{Op}\ \Gamma\ P \to \mathsf{Op}\ \Gamma\ P \\
 & \iota : \mathsf{Tms}\ \Gamma\ P \to \mathsf{Op}\ \Gamma\ P
\end{array}
$$

The Sig sort represents a simple class of algebraic signatures with values in some object theory $(\mathsf{Con}, \mathsf{Ty}, \mathsf{Tm}, \ldots)$. Each signature is described by an associated telescope of indices $P$ as a telescope of indices, a *finite list* of operations:

- $\Pi\ A\ B$, a (dependent) abstraction over some type $A$ from the object theory, of another operation $B$.
- $\Pi\iota\ p\ B$, an abstraction over a recursive occurence of the object being defined, with indices $p$, of another operation $B$.
- $\iota\ p$, a constructor of the object being defined, with indices $p$.

For example, the signature of natural numbers lives in the empty context has an empty telescope of indices. It is defined by $\mathsf{NatTh} : \mathsf{Sig}\ \bullet\ \bullet\ =\ \bullet \rhd \iota\ (\,) \rhd \Pi\iota\ (\,)\ (\iota\ (\,))$. We can add labels to aid readability, omitting parameters if they are empty:

$$\mathsf{NatTh} := \bullet \rhd \mathit{zero} : \iota \rhd \mathit{succ} : \Pi\iota\ \iota\,.$$

Notice that this syntax only allows occurrences of $\iota$ in positive positions, which is a requirement for inductive types. Different classes of theories and quantification are explored in detail by Kovács [24].

### 3.2 Algebras, displayed algebras and inductive algebras

In order to make use of our definition for theories, we would like to be able to interpret the structure into a semantic universe. An algebra $(X, a) : \mathsf{Alg}\ T$ for a signature $T$ and carrier $X$ defines a way to interpret the structure of $T$ in terms of an object type $X : \mathsf{Ty}\ \Gamma$. This produces a type which matches the structure of $T$ but replaces each occurrence of $\iota$ with $X$. The function arrows in $T$ are interpreted as function arrows in the target universe. Algebras for the signature of natural numbers might look like

$$\mathsf{Alg}\ \mathsf{NatTh} \simeq (X : \mathsf{Ty}\ \Gamma) \times (\mathit{zero} : \mathsf{Tm}\ \Gamma\ X) \times (\mathit{succ} : \mathsf{Tm}\ \Gamma\ (\Pi\ X\ X))$$

We have a choice in terms of how much we want to interpret $T$ in the object theory, and how much we want to interpret it in the *metatheory*. Here we have chosen to interpret a theory as a metatheoretical iterated pair type, but an operation as a term in the object theory.

Very special classes of algebras support *induction.* To formulate induction, we first need to define displayed algebras. A displayed algebra $(M, m)$ over an algebra $(X, a)$ for a signature $T$ with carrier $M$ mirrors the shape of $T$ like an algebra does, but each recursive occurrence $\iota$ is now replaced by $M$ applied to the corresponding value of the algebra. The displayed algebras for natural numbers are

$$\begin{aligned}\mathsf{DispAlg}\ (X, \mathit{zero}, \mathit{succ}) \simeq\ & (M : \mathsf{Ty}\ (\Gamma \rhd X)) \\ & \times (\mathit{zero'} : \mathsf{Tm}\ \Gamma\ M[\langle \mathit{zero} \rangle]) \\ & \times (\mathit{succ'} : \mathsf{Tm}\ \Gamma\ (\Pi\ (x : X)\ \Pi\ M[\langle x \rangle]\ M[\langle \mathit{succ}\ x \rangle]))\end{aligned}$$

The type $M$ is often called the *motive*, and $m$ the *methods.* A section is a dependent function from $X$ to $M$ which takes its values from the displayed algebra. For natural numebrs,

$$\begin{aligned}&\mathsf{Section}\ \{(X, \mathit{zero}, \mathit{succ})\}\ (M, \mathit{zero'}, \mathit{succ'}) \\ &\qquad \simeq (f : \mathsf{Tm}\ \Gamma\ (\Pi\ (x : X)\ M[\langle x \rangle])) \\ &\qquad\quad \times (f\ \mathit{zero} = \mathit{zero'}) \times ((x : \mathsf{Tm}\ \Gamma\ X) \to f\ (\mathit{succ}\ x) = \mathit{succ'}\ x\ (f\ x))\end{aligned}$$

**Definition 1.** *An algebra is* inductive *if every displayed algebra over it has a section.*

A displayed algebra $(M, m)$ is the input of induction: a proof of $M$ for each structural case in $X$ by $m$. A section $f$ is the output: a proof of $M$ for all $X$ constructed from $m$.

### 3.3 Internal and external constructions

For the remainder of the paper we choose a fixed representation for algebras, displayed algebras and sections. We will omit the full definitions here for space, but where missing they can be found in section 8.1 of the appendix.

We define these constructions in two ways: one that is fully internal to the object theory and the other that is partially external (using the metatheory). In particular, all the external constructions are *positive* in the syntax of the object theory (Ty, Tm, Tel, etc) so that they can be added into the syntax retroactively.

First, we define an 'external' version of algebras

$$\begin{aligned}&\mathsf{Algebra} : (T : \mathsf{Sig}\ \Gamma\ P) \to \mathsf{Ty}\ (\Gamma \rhd P) \to \mathbf{Set} \\ &\mathsf{Algebra}\ T\ X := (a : \mathsf{In}\ T\ X) \to \mathsf{Tm}\ \Gamma\ X[\langle \mathsf{out}\ a \rangle]\end{aligned}$$

$$\mathsf{Alg}\ T := (X : \mathsf{Ty}\ (\Gamma \rhd P)) \times \mathsf{Algebra}\ T\ X\,,$$

which take some arguments In and produce the output $X$ evaluated at the appropriate index out $a$ based on the arguments. This is the uncurried version of the presentation with $\Pi$ types. The type In is defined as $(v : \mathsf{Var}\ T) \times$

$\mathsf{Tms}\ \Gamma\ (\mathsf{in}\ (T\ v)\ X)$ for where $\mathsf{Var}\ T$ indexes operations in theories. The function

$$\begin{aligned}
&\mathsf{in} : \mathsf{Op}\ \Gamma\ P \to \mathsf{Ty}\ (\Gamma \rhd P) \to \mathsf{Tel}\ \Gamma\\
&\mathsf{in}\ (\Pi\ A\ B)\ X := \bullet \rhd (a : A) \rhd \mathsf{in}\ B[\langle a\rangle]\ X\\
&\mathsf{in}\ (\Pi\iota\ p\ B)\ X := \bullet \rhd X[\langle p\rangle] \rhd \mathsf{in}\ B\ X\\
&\mathsf{in}\ (\iota\ p)\ X := \bullet
\end{aligned}$$

computes a telescope in the object theory, of the input arguments to the algebra element of a given operation. For the output, there is a function to compute the indices. It is defined as $\mathsf{out}\ \{T\ v\}\ a\ X$, where

$$\begin{aligned}
&\mathsf{out} : \{O : \mathsf{Op}\ \Gamma\ P\} \to \mathsf{Tms}\ \Gamma\ (\mathsf{in}\ O\ X) \to \mathsf{Ty}\ \Gamma \to \mathsf{Tms}\ \Gamma\ P\\
&\mathsf{out}\ \{\Pi\ A\ B\}\ (a, t)\ X := \mathsf{out}\ \{B[\langle a\rangle]\}\ t\ X\\
&\mathsf{out}\ \{\Pi\iota\ p\ B\}\ (r, t)\ X := \mathsf{out}\ \{B\}\ t\ X\\
&\mathsf{out}\ \{\iota\ p\}\ (\,)\ X := p\,.
\end{aligned}$$

We can also define an internal version of algebras as telescopes of $\Pi$ types

$$\begin{aligned}
&\mathsf{algebra} : \mathsf{Sig}\ \Gamma\ P \to \mathsf{Ty}\ (\Gamma \rhd P) \to \mathsf{Tel}\ \Gamma\\
&\mathsf{algebra}\ \bullet\ X := \bullet\\
&\mathsf{algebra}\ (T \rhd O)\ X := \mathsf{algebra}\ T\ X \rhd \Pi\ (a : \mathsf{in}\ O\ X)\ X[\langle \mathsf{out}\ O\ a\ X\rangle]
\end{aligned}$$

$$\mathsf{alg}\ T := \bullet \rhd (X : \Pi\ P\ \mathcal{U}) \rhd \mathsf{algebra}\ T\ (\mathsf{El}\ X@)\,.$$

All internal constructions have ‘realisation’ functions into the metatheory

$$\ulcorner\_\urcorner : \mathsf{Tms}\ \Gamma\ (\mathsf{alg}\ T) \to \mathsf{Alg}\ T \qquad \ulcorner\_\urcorner : \mathsf{Tms}\ \Gamma\ (\mathsf{algebra}\ T\ X) \to \mathsf{Algebra}\ T\ X\,.$$

A similar construction can be performed for displayed algebras over algebras

$$\begin{aligned}
&\mathsf{dispAlgebra} : ((X, x) : \mathsf{Alg}\ T) \to \mathsf{Ty}\ (\Gamma \rhd P \rhd X) \to \mathsf{Tel}\ \Gamma\\
&\mathsf{DispAlg}\ (X, x) := (M : \mathsf{Ty}\ (\Gamma \rhd P \rhd X)) \times \mathsf{Tms}\ \Gamma\ (\mathsf{dispAlgebra}\ (X, x)\ M)
\end{aligned}$$

as well as the telescopic internal equivalent $\mathsf{dispAlg} : \mathsf{Tel}\ (\Gamma \rhd \mathsf{alg}\ T)$. Internal displayed algebras come with a realisation function $\ulcorner\_\urcorner$ too. Finally, we construct sections over displayed algebras

$$\begin{aligned}
&\mathsf{Sec}\ M := \mathsf{Tm}\ (\Gamma \rhd P \rhd X)\ M\\
&\mathsf{Coh}\ f := \forall v\ a.\ f[\langle \mathsf{dispOut}\ a\rangle] = m\ (v, \mathsf{apply}\ f\ a)\\
&\mathsf{IntCoh}\ f := \forall v\ a.\ \mathsf{Tm}\ \Gamma\ (\mathsf{Id}\ f[\langle \mathsf{dispOut}\ a\rangle])\ (m\ (v, \mathsf{apply}\ f\ a))\\
&\mathsf{Section}\ (M, m) := (f : \mathsf{Sec}\ M) \times \mathsf{Coh}\ f\,.
\end{aligned}$$

which have coherence rules using either the equality of the metatheory ($\mathsf{Coh}$) or the propositional equality of the object theory ($\mathsf{IntCoh}$). The $\mathsf{apply}$ function takes the inputs for the algebra and a section, and produces the inputs for the displayed algebra by evaluating the section. Sections also have internal analogues

$$\mathsf{section} : \mathsf{Tel}\ (\Gamma \rhd \mathsf{alg}\ T \rhd \mathsf{dispAlg})$$

which only use propositional equality. As a result, the realisation functions for sections $\ulcorner f \urcorner_0 : \mathsf{Sec}\ \ulcorner m \urcorner$ and $\ulcorner f \urcorner_1 : \mathsf{IntCoh}\ \ulcorner f \urcorner_0$ produce only internal coherence proofs.

Now we can define a synonym for internal inductive algebras as a telescope $\mathsf{indAlg}\ T := \bullet \rhd \mathsf{alg}\ T \rhd \Pi\ \mathsf{dispAlg}\ \mathsf{section}$ with realisations

$$\begin{aligned}
\ulcorner _ \urcorner_0 &: \mathsf{Tms}\ \Gamma\ \mathsf{indAlg}[\langle X \rangle] \to (m : \mathsf{DispAlg}\ X) \to \mathsf{Sec}\ m \\
\ulcorner _ \urcorner_1 &: (i : \mathsf{Tms}\ \Gamma\ \mathsf{indAlg}[\langle X \rangle] \to (m : \mathsf{DispAlg}\ X) \to \mathsf{IntCoh}\ \ulcorner i \urcorner_0\ m\,.
\end{aligned}$$

Next we will make use of the external versions of algebras, displayed algebras, and sections in order to add inductive algebras as part of the well-typed syntactical definition of a type theory in a strictly-positive, fully-applied manner. Later, we will make use of the internal versions in order to be able to package inductive algebras as a single syntactic entity that corresponds to data representations.

### 3.4 A type theory with inductive families, $\boldsymbol{\lambda_{\text{IND}}}$

The languge $\lambda_{\text{IND}}$, is a dependent type theory with $\Pi$, $\mathsf{Id}$ with $\mathsf{uip}$, and a universe $\mathcal{U} : \mathcal{U}$. We will not concern ourselves with a universe hierarchy but our results should be readily extensible to such a type theory. This language also has inductive families and global definitions. We follow a similar approach to Cockx and Abel [16] by packaging named inductive constructions and function definitions into a global context $\Sigma : \mathsf{Glob}$, and indexing local contexts by global contexts. The contexts $\mathsf{Con}$ in the resulting theory are pairs $(\Sigma : \mathsf{Glob}) \times \mathsf{Loc}\ \Sigma$ where $\mathsf{Loc}\ \Sigma$ are local contexts given by a closed telescope of types as usual. Substitutions only occur between local contexts. Items in a signature $\Sigma$ can be either

- function definitions $\mathsf{def}\ P\ A\ t$ for some parameters $P : \mathsf{Loc}\ \Sigma$, return type $A : \mathsf{Ty}\ (\Sigma, P)$ and implementation $t : \mathsf{Tm}\ A\ (\Sigma, P)$,
- postulates $\mathsf{post}\ P\ A$ for some parameters $P : \mathsf{Loc}\ \Sigma$ and return type $A : \mathsf{Ty}\ (\Sigma, P)$, or
- inductive type definitions $\mathsf{data}\ P\ T$ for some indices $P : \mathsf{Loc}\ \Sigma$ and signature $T : \mathsf{Sig}\ (\Sigma, \bullet)\ P$.

This allows us to define data types such as vectors

$$\begin{aligned}
&\mathit{Vect} : \mathsf{data}\ (\bullet \rhd \mathcal{U} \rhd \mathit{Nat}) \\
&\quad (\bullet \rhd \mathit{nil} : \Pi\ (T : \mathcal{U})\ (\iota\ (T, \mathit{zero})) \\
&\quad\quad \rhd \mathit{cons} : \Pi\ (T : \mathcal{U})\ \Pi\ (n : \mathit{Nat})\ \Pi\iota\ (T, n)\ \Pi\ T\ (\iota\ (T, \mathit{succ}\ n))\,.
\end{aligned}$$

We do not make a distinction between parameters and indices for data types, but our system is extensible to one with additional uniform parameters and we leave its formulation as an implementation detail.

In order to actually construct inductive types in $\lambda_{\text{IND}}$, we need to extend the syntax with some term and type formers. First, we add a type former

$$\mathsf{D} : \mathsf{data}\ P\ T \in \Sigma \to \mathsf{Ty}\ (\Sigma, \Delta \rhd P)$$

which, given a data definition $i$ in $\Sigma$, and terms for its indices $p$, constructs the data type $(\mathsf{D}\ i)[p]$. The relation $I \in \Sigma$ finds items in a global context, analogously to how $\mathsf{Var}\ \Gamma\ A$ defines variables for type $A$ in a local context $\Gamma$. It is a decidable relation and stable under substitutions and global weakening.

Additionally, we add a constructor term

$$\mathsf{C} :\ \forall i.\ \mathsf{Algebra}\ T\ (\mathsf{D}\ i)$$

which fully applied, defines the data constructor $\mathsf{C}\ a$ of type $(\mathsf{D}\ i)[\langle p\rangle]$ for arguments $a$, by 'freely' extending the syntax with an algebra for the type family $\mathsf{D}\ i$. The (strictly positive) occurrences of $\mathsf{Tm}$ in $\mathsf{Algebra}$ are part of the inductive syntax of the type theory. We can now construct, for example, natural numbers as

$$\mathsf{C}_{Nat}\ (succ, (\mathsf{C}_{Nat}\ (zero, (\,)))) : \mathsf{Tm}\ \Gamma\ (\mathsf{D}\ Nat)\,.$$

Next, we add an eliminator term

$$\mathsf{E} :\ \forall i.\ (m : \mathsf{DispAlg}\ \mathsf{C}_i) \to \mathsf{Sec}\ m$$

which given a data definition $i$ in $\Sigma$, a motive and methods for $i$, eliminates each $d : (\mathsf{D}\ i)[\langle p\rangle]$ into $M[\langle p; d\rangle]$. This captures the induction principle of the data type. The coherence part of the section is captured by an equality constructor in the syntax

$$\mathsf{E\text{-}id} :\ \forall i.\ (m : \mathsf{DispAlg}\ \mathsf{C}_i) \to \mathsf{Coh}\ (\mathsf{E}_i\ m)\,.$$

**Lemma 1.** *The constructor algebra $\mathsf{C}_i$ of a data type $i$ is inductive.*

*Proof. For every displayed algebra $m$ over $\mathsf{C}$ we get a section $(\mathsf{E}\ m, \mathsf{E\text{-}id}\ m)$.*

Finally, we add terms to the language for function definitions and postulates

$$\mathsf{F} : \mathsf{def}\ P\ A\ t \in \Sigma \to \mathsf{Tm}\ (\Sigma, \Delta \rhd P)\ A \qquad \mathsf{P} : \mathsf{post}\ P\ A \in \Sigma \to \mathsf{Tm}\ (\Sigma, \Delta \rhd P)\ A$$

along with an equality constructor for function definitions $\mathsf{F\text{-}id} : \forall i.\ \mathsf{F}\ i = t$. The language we have defined thus far is comparable to the core language of a proof assistant.

### 3.5 Extending $\boldsymbol{\lambda}_{\text{IND}}$ with representations

We extend the language $\lambda_{\text{IND}}$ to form $\lambda_{\text{REP}}$, which allows the definition of custom representations for data types and global functions. The machinery of algebras that we have developed in section 3.2 allows for a very direct definition of representations of data types: A representation for a $\mathsf{data}\ P\ T$ is an inductive algebra for $T$. Representations live alongside items in a global context, and each item only corresponds to at most one representation. We achieve this by adding a constructor to global contexts $\trianglerighteq : (\Sigma : \mathsf{Glob}) \to (I : \mathsf{Item}\ \Sigma) \to \mathsf{Rep}\ \Sigma\ I \to \mathsf{Glob}$. Representations are defined inductively by

$$\begin{aligned}
&\mathsf{datarep} : \mathsf{Tms}\ (\Sigma, \epsilon)\ (\mathsf{indAlg}\ T) \to \mathsf{Rep}\ \Sigma\ (\mathsf{data}\ P\ T)\\
&\mathsf{defrep} : (x : \mathsf{Tm}\ (\Sigma, P)\ A) \to \mathsf{Tm}\ (\Sigma, P)\ (\mathsf{Id}\ x\ t) \to \mathsf{Rep}\ \Sigma\ (\mathsf{def}\ P\ A\ t)
\end{aligned}$$

We will write $\mathsf{datarep}\ (R, r, Q)$ to unpack the telescope of an inductive algebra with carrier $R$, algebra $r$ and induction $Q$. Representations for definitions are also included, where a definition can be represented by a term propositionally equal to original definition, but perhaps with better computational properties.

### 3.6 Reasoning about representations

To allow reasoning about representations internally to $\lambda_{\text{REP}}$ we add a type former $\mathsf{Repr} : \mathsf{Ty}\ \Gamma \to \mathsf{Ty}\ \Gamma$ along with two new terms in the syntax, forming an isomorphism

$$\mathsf{repr} : \mathsf{Tm}\ \Gamma\ T \simeq \mathsf{Tm}\ \Gamma\ (\mathsf{Repr}\ T) : \mathsf{unrepr} \tag{5}$$

which preserves $\Pi/\mathsf{Id}/\mathcal{U}$. The type $\mathsf{Repr}\ T$ is the defined representation of the type $T$. The term $\mathsf{repr}$ takes a term of type $T$ to its representation of type $\mathsf{Repr}\ T$, and $\mathsf{unrepr}$ undoes the effect of $\mathsf{repr}$, treating a represented term as an inhabitant of its original type. These new constructs come with equality constructors in the syntax shown in fig. 1.

| | |
|---|---|
| $\mathsf{reprr} : \mathsf{unrepr}\ (\mathsf{repr}\ t) \equiv t$ | $\mathsf{Repr}\text{-}\mathcal{U} : \mathsf{Repr}\ \mathcal{U} \equiv \mathcal{U}$ |
| $\mathsf{reprl} : \mathsf{repr}\ (\mathsf{unrepr}\ t) \equiv t$ | $\mathsf{repr\text{-}code} : \mathsf{repr}\ (\mathsf{code}\ T) \equiv \mathsf{code}\ T$ |
| | $\mathsf{unrepr\text{-}code} : \mathsf{unrepr}\ (\mathsf{code}\ T) \equiv \mathsf{code}\ T$ |
| $\mathsf{Repr}\text{-}\Pi : \mathsf{Repr}\ (\Pi\ T\ U) \equiv \Pi\ T\ (\mathsf{Repr}\ U)$ | |
| $\mathsf{repr}\text{-}\lambda : \mathsf{repr}\ (\lambda\ u) \equiv \lambda\ (\mathsf{repr}\ u)$ | $\mathsf{Repr\text{-}Id} : \mathsf{Repr}\ (\mathsf{Id}\ a\ b) \equiv \mathsf{Id}\ (\mathsf{repr}\ a)\ (\mathsf{repr}\ b)$ |
| $\mathsf{unrepr}\text{-}\lambda : \mathsf{unrepr}\ (\lambda\ u) \equiv \lambda\ (\mathsf{unrepr}\ u)$ | $\mathsf{repr\text{-}refl} : \mathsf{repr}\ (\mathsf{refl}\ u) \equiv \mathsf{refl}\ (\mathsf{repr}\ u)$ |
| $\mathsf{repr}\text{-}@ : \mathsf{repr}\ (f\ @) \equiv (\mathsf{repr}\ f)\ @$ | $\mathsf{unrepr\text{-}refl} : \mathsf{unrepr}\ (\mathsf{refl}\ u) \equiv \mathsf{refl}\ (\mathsf{unrepr}\ u)$ |
| $\mathsf{unrepr}\text{-}@ : \mathsf{unrepr}\ (f\ @) \equiv (\mathsf{unrepr}\ f)\ @$ | $\mathsf{repr\text{-}J} : \mathsf{repr}\ (\mathsf{J}\ C\ w\ e) \equiv \mathsf{J}\ (\mathsf{Repr}\ C)\ (\mathsf{repr}\ w)\ e$ |
| $\mathsf{repr}[] : \mathsf{repr}\ (t[\sigma]) \equiv (\mathsf{repr}\ t)[\sigma]$ | $\mathsf{unrepr\text{-}J} : \mathsf{unrepr}\ (\mathsf{J}\ (\mathsf{Repr}\ C)\ w\ e) \equiv \mathsf{J}\ C\ (\mathsf{unrepr}\ w)\ e$ |
| $\mathsf{unrepr}[] : \mathsf{unrepr}\ (t[\sigma]) \equiv (\mathsf{unrepr}\ t)[\sigma]$ | |
| $\mathsf{Repr}[] : \mathsf{Repr}\ (T[\sigma]) \equiv (\mathsf{Repr}\ T)[\sigma]$ | |

**Fig. 1.** Coherence of the representation operators with substitutions, $\Pi$, $\mathsf{Id}$, and universes . The terms $\mathsf{Repr}\ (\mathsf{El}\ t)$, $\mathsf{repr}\ (\pi_2\sigma)$ and $\mathsf{unrepr}\ (\pi_2\sigma)$ do not reduce.

So far the representation operators do not really do much other than commute with almost everything in the syntax. In order to make them useful, we need to define how they compute when they encounter data types which have a defined representation in the global context. We use a decidable relation $R \in_i \Sigma'$ to mean that $R : \mathsf{Rep}\ \Sigma\ I$ is the representation of an item $I : \mathsf{Item}\ \Sigma$ where $i : I \in \Sigma'$. This relation is a proposition, so it is proof-irrelevant. Furthermore, it is stable under substitutions and global weakening, because each item can only be represented once in a global context. In the following rules, $r : \mathsf{datarep}\ (R, r, Q) \in_i \Sigma$.

Firstly, we define the reduction that occurs when a type $\mathsf{D}\ i$ is represented,

$$\mathsf{Repr\text{-}D}_i : \forall r.\ \mathsf{Repr}\ (\mathsf{D}\ i) = \mathsf{El}\ R@\,, \tag{6}$$

yielding the carrier $R$ of the inductive algebra that represents it (after converting it from a function into the universe to a type family).

Next, we add a rule for representing constructors, albeit in propositional form, where

$$\mathsf{repr\text{-}C}_i : \forall r.\ \mathsf{Tm}\ (\Sigma, \Delta)\ (\mathsf{Id}\ (\mathsf{repr}\ (\mathsf{C}\ a))\ (\ulcorner r \urcorner\ a^{\mathsf{Repr}})) \tag{7}$$

Here, the operation $\_^{\mathsf{Repr}}$ is used to apply the term former $\mathsf{repr}$ to the recursive part of the arguments $a$. The full construction can be found in section 8.2 of the appendix

One might be tempted to make this equality definitional too. Unfortunately, this would render conversion checking undecidable, because if one applies $\mathsf{unrepr}$ to a term $\mathsf{repr}\ (\mathsf{C}\ a)$ which has already been reduced to its representation, $\mathsf{unrepr}\ (\ulcorner r \urcorner\ a^{\mathsf{Repr}})$, there is no clear way to decide that this is convertible to $\mathsf{C}\ a$ even though the definitional equality rules would imply that it is (due to the annihilation of $\mathsf{repr}$ and $\mathsf{unrepr}$). There is no equivalent of $\mathsf{unrepr}$ for types, so (6) preserves the decidability of conversion checking.

We can also add a propositional equality rules for representing eliminators. First, representing an eliminator just applies $\mathsf{repr}$ to the motive and methods:

$$\mathsf{repr\text{-}E}_i : \forall r.\ \mathsf{Tm}\ (\Sigma, \Delta)\ (\mathsf{Id}\ (\mathsf{repr}\ (\mathsf{E}\ m))\ (\mathsf{E}\ m^{\mathsf{Repr}}))$$
$$\mathsf{unrepr\text{-}E}_i : \forall r.\ \mathsf{Tm}\ (\Sigma, \Delta)\ (\mathsf{Id}\ (\mathsf{unrepr}\ (\mathsf{E}\ m))\ (\mathsf{E}\ m^{\mathsf{Unrepr}}))$$

Additionally, eliminating something using $\mathsf{E}$ should be the same as eliminating the representation of that thing using the represented eliminator $Q$:

$$\mathsf{repr\text{-}equiv\text{-}E}_i : \forall r.\ \mathsf{Tm}\ (\Sigma, \Delta)\ (\mathsf{Id}\ (\mathsf{E}\ m)\ (s.\ (\ulcorner Q \urcorner_0\ m^{\mathsf{Repr*}})[\langle \mathsf{repr}\ s \rangle])))$$

Above we use more auxilliary definitions which represent the carriers of algebras, as well as displayed algebras (appendix section 8.2):

$$\_^{\mathsf{Repr}} : \mathsf{Algebra}\ T\ X \to \mathsf{Algebra}\ T\ (\mathsf{Repr}\ X)$$
$$\_^{\mathsf{Repr}} : \mathsf{DispAlgebra}\ a\ M \to \mathsf{DispAlgebra}\ a\ (\mathsf{Repr}\ M)$$
$$\_^{\mathsf{Repr}*} : \mathsf{DispAlgebra}\ a\ M \to \mathsf{DispAlgebra}\ a^{\mathsf{Repr}}\ (p\ x.\ M[\langle p; \mathsf{unrepr}\ x \rangle])$$

We do not need an additional equality rule for representing function definitions as this is given by the equality proof $p$ in the definition of a representation $\mathsf{defrepr}\ t\ p$, when accounting for the definitional equality between a definition and its implementation.

### 3.7 Translating representations away

We now define a translation step $\mathcal{R}$ from $\lambda_{\text{REP}}$ to $\lambda_{\text{IND}}^{\text{EXT}}$, meant to be applied during the compilation process. More specifically, the translation target is the extensional flavour of $\lambda_{\text{IND}}$ by adding the equality reflection rule. General undecidability of conversion is not a problem because type checking is decidable for $\lambda_{\text{REP}}$[5] and we only need to apply this transformation after type checking, on fully-typed terms. The translation is defined over the syntax of $\lambda_{\text{REP}}$ [13] such that definitional equality is preserved. Overall, $\mathcal{R}$ preserves the structure of $\lambda_{\text{REP}}$, but maps constructs to their 'terminal' representations. First, we define a translation of global contexts $\mathcal{R} : \mathsf{Glob}_{\text{REP}} \to \mathsf{Glob}_{\text{IND}}^{\text{EXT}}$ as

$$\mathcal{R}\ \bullet := \bullet \qquad \mathcal{R}\ (\Sigma \rhd I) := \mathcal{R}\Sigma \rhd \mathcal{R}I \qquad \mathcal{R}\ (\Sigma \trianglerighteq I\ R) := \mathcal{R}\Sigma$$

which erases all items with defined representations. This utilises a translation of items $\mathcal{R} : \mathsf{Item}_{\text{REP}}\ \Sigma \to \mathsf{Item}_{\text{IND}}^{\text{EXT}}\ \mathcal{R}\Sigma$ which simply recurses on all subterms with $\mathcal{R}$. Types are translated as

$$\mathcal{R} : \mathsf{Ty}_{\text{REP}}\ (\Sigma, \Delta)\ \to \mathsf{Ty}_{\text{IND}}^{\text{EXT}}\ (\mathcal{R}\Sigma, \mathcal{R}\Delta)$$

$$\mathcal{R}\ (\mathsf{D}\ i) := \begin{cases} \mathcal{R}\ (\mathsf{El}\ R@) & \text{if } \mathsf{datarep}\ (R, r, Q) \in_i \Sigma \\ \mathsf{D}\ \mathcal{R}i & \text{otherwise} \end{cases} \qquad \mathcal{R}\ (\mathsf{Repr}\ T) := \mathcal{R}T$$

(otherwise recurse on all subterms with $\mathcal{R}$) .

The definitional equality rules of $\mathsf{Repr}$ and $\mathsf{D}$ are mirrored, but $\mathcal{R}$ is now applied to all subterms. Similarly, terms are translated as

$$\mathcal{R} : \mathsf{Tm}_{\text{REP}}\ (\Sigma, \Delta)\ T \to \mathsf{Tm}_{\text{IND}}^{\text{EXT}}\ (\mathcal{R}\Sigma, \mathcal{R}\Delta)\ \mathcal{R}T$$

$$\mathcal{R}\ (\mathsf{C}_i\ a) = \begin{cases} \ulcorner \mathcal{R}r \urcorner\ \mathcal{R}a & \text{if } \mathsf{datarep}\ (R, r, Q) \in_i \Sigma \\ \mathsf{C}_{\mathcal{R}i}\ \mathcal{R}a & \text{otherwise} \end{cases}$$

$$\mathcal{R}\ (\mathsf{E}_i\ m) = \begin{cases} \ulcorner \mathcal{R}Q \urcorner_0\ \mathcal{R}m & \text{if } \mathsf{datarep}\ (R, r, Q) \in_i \Sigma \\ \mathsf{E}_{\mathcal{R}i}\ \mathcal{R}m & \text{otherwise} \end{cases}$$

$$\mathcal{R}\ (\mathsf{F}_i) = \begin{cases} \mathcal{R}t & \text{if } \mathsf{defrep}\ t\ p \in_i \Sigma \\ \mathsf{F}_{\mathcal{R}i} & \text{otherwise} \end{cases} \qquad \mathcal{R}\ (\mathsf{repr}\ t),\ \mathcal{R}\ (\mathsf{unrepr}\ t) := \mathcal{R}t$$

$$\mathcal{R}\ (\mathsf{repr\text{-}C}_i\ a),\ \mathcal{R}\ (\mathsf{repr\text{-}E}_i\ m),\ \mathcal{R}\ (\mathsf{unrepr\text{-}E}_i\ m),\ \mathcal{R}\ (\mathsf{repr\text{-}equiv\text{-}E}_i\ m) := \mathsf{refl}$$

(otherwise recurse on all subterms with $\mathcal{R}$)

Constructor, eliminator and definition translations mirror the equality rules in section 3.5, but apply $\mathcal{R}$ to all subterms rather than only the recursive occurrences of the data type being represented. As a result, all of the propositional

[5] Not formalised in this paper.

equality constructors are translated to reflexivity, since after applying $\mathcal{R}$ both sides are identical.

The equality constructors of the syntax of $\lambda_{\text{REP}}$ must also be translated. The base equalities of the theory are preserved by their counterparts in $\lambda^{\text{EXT}}_{\text{IND}}$. The coherence rules for representation operators (fig. 1) are preserved by metatheoretic reflexivity on the other side, since all representation operators are erased. Finally, coherence rules for definitions $\mathsf{F}$ and eliminators $\mathsf{E}$ are preserved by reflecting the propositional coherence rules provided by their defined representations:

$$\mathsf{ap}_{\mathcal{R}}\ (\mathsf{E\text{-}id}_i\ m) := \begin{cases} \mathsf{reflect}\ \ulcorner \mathcal{R}Q \urcorner_1\ \mathcal{R}m & \text{if } \mathsf{datarep}\ (R, r)\ Q \in_i \Sigma \\ \mathsf{E\text{-}id}_{\mathcal{R}i}\ \mathcal{R}m & \text{otherwise} \end{cases}$$

$$\mathsf{ap}_{\mathcal{R}}\ (\mathsf{F\text{-}id}_i) := \begin{cases} \mathsf{reflect}\ \mathcal{R}p & \text{if } \mathsf{defrep}\ t\ p \in_i \Sigma \\ \mathsf{F\text{-}id}_{\mathcal{R}i} & \text{otherwise} \end{cases}$$

(otherwise recurse on all equality constructors with $\mathsf{ap}_{\mathcal{R}}$)

**Theorem 1.** *$\mathcal{R}$ preserves typing and definitional equality:* $(t_1, t_2 : \mathit{Tm}_{\text{IND}}\ \Gamma\ A) \to t_1 = t_2 \to \mathcal{R}t_1 = \mathcal{R}t_2$.

*Proof. By* $\mathsf{ap}_{\mathcal{R}}$.

$\mathcal{R}$ is not injective in general, because two distinct (by their location in the global context) data types might be defined to have the same representation.

**Theorem 2.** *$\mathcal{R}$ is a left-inverse of the evident inclusion $i : \lambda_{\text{IND}} \hookrightarrow \lambda_{\text{REP}}$:*

$$(t : \mathit{Tm}_{\text{IND}}\ \Gamma\ A) \to \mathcal{R}(it) = t\,.$$

*Proof. The inclusion produces global contexts in $\lambda_{\text{REP}}$ without the $\trianglerighteq$ constructor. Thus no items have defined representations. Also, the action of $\mathcal{R}$ on the image of $i$ does not invoke the equality reflection rule. With that constraint, and by induction on the syntax, $\mathcal{R} \circ i$ is the identity function on $\lambda_{\text{IND}}$.*

### 3.8 Computational irrelevance

In order to reason about computational irrelevance, we assume that there is an additional program extraction step $\mathcal{E}$ from $\lambda_{\text{IND}}$ into some simply-typed calculus, denoted by vertical bars $|x|$. As opposed to $\mathcal{R}$, $\mathcal{E}$ operates on the unquotiented syntax of $\lambda_{\text{IND}}$. This can be justified by interpreting the quotient-inductive constructions from before into setoids [23]. This kind of transformation is used because we might want to compile two definitionally equal terms differently. For example, we might not always want to reduce function application redexes. We will use the `monospace` font for terms in $\lambda$.

**Definition 2.** *A function $f : \mathit{Tm}\ \Gamma\ (\Pi\ A\ B)$, is* computationally irrelevant *if* $|\mathcal{R}A| = |\mathcal{R}B|$ *and* $|\mathcal{R}f| = \texttt{id}$.

**Theorem 3.** *The type former $\mathsf{Repr}$ is injective up to internal isomorphism, i.e.*

$$\mathit{Tm}\ \Gamma\ (\mathit{Id}\ (\mathit{Repr}\ T)\ (\mathit{Repr}\ T')) \to \mathit{Tm}\ \Gamma\ (\mathit{Iso}\ T\ T') \tag{8}$$

*Moreover, this isomorphism is computationally irrelevant.*

*Proof.* *For the input proof $p$, the forward direction is $\lambda x.\ \mathit{unrepr}_{T'}\ (\mathit{J}\ \mathit{id}\ (\mathit{repr}\ x)\ p)$ and the backward direction is $\lambda x.\ \mathit{unrepr}_{T}\ (\mathit{J}\ \mathit{id}\ (\mathit{repr}\ x)\ (\mathit{sym}\ p))$. The coherence holds definitionally by*

$$\begin{aligned}
&\mathit{unrepr}_{T'}\ (\mathit{J}\ \mathit{id}\ (\mathit{repr}\ (\mathit{unrepr}_{T}\ (\mathit{J}\ \mathit{id}\ (\mathit{repr}\ x)\ (\mathit{sym}\ p))))\ p)\\
&= \mathit{unrepr}_{T'}\ (\mathit{J}\ \mathit{id}\ (\mathit{J}\ \mathit{id}\ (\mathit{repr}\ x)\ (\mathit{sym}\ p))\ p)\ \ by\ \mathit{reprl}\\
&= \mathit{J}\ \mathit{id}\ (\mathit{J}\ \mathit{id}\ x\ (\mathit{sym}\ p))\ p\ \ by\ \mathit{unrepr\text{-}J} \times 2 + \mathit{reprr}\\
&= x\ \ by\ (\mathit{uip} + \mathit{J\text{-}elim}) \times 2\,,
\end{aligned}$$

*and similarly for the other side. After applying $\mathcal{R}$, we get $\lambda x.\ \mathit{J}\ \mathit{id}\ x\ p =_{\mathit{uip}+\mathit{J\text{-}elim}} \lambda x.\ x$ on both sides.*

Consider extending our languages with usage-aware subset $\Sigma$-types

$$\{\_ \mid \_\} : (A : \mathsf{Ty}\ \Gamma) \to \mathsf{Ty}\ (\Gamma \triangleright A) \to \mathsf{Ty}\ \Gamma$$

in such a way that $\mathsf{Repr}$ and $\mathcal{R}$ preserve them, but the extraction step erases the right component, i.e. $|\{A \mid B\}| = |A|$, $|(x, y)| = |x|$ and $|\pi_1 x| = |x|$.[6] Suppose we have an inductive family $G : \mathsf{data}\ (\bullet \triangleright I)\ T_G \in \Sigma$ over some index type $I$, and an inductive type $F : \mathsf{data}\ \bullet\ T_F \in \Sigma$ such that $G$ is represented by a refinement $f : \mathsf{Tm}\ (\Sigma, \bullet)\ (\Pi\ (\mathsf{D}\ F)\ I)$ of $F$,

$$\mathsf{datarep}\ (i.\ \{x : \mathsf{D}\ F \mid \mathsf{Id}\ (f\ x)\ i\}, r, Q) \in_G \Sigma\,.$$

Then, we can construct computationally irrelevant functions

$$\begin{aligned}
&\mathsf{forget}_i : \mathsf{Tm}\ \Gamma\ (\Pi\ (\mathsf{D}\ G)[i]\ (\mathsf{D}\ F)) && \mathsf{remember} : \mathsf{Tm}\ \Gamma\ (\Pi\ (x : \mathsf{D}\ F)\ (\mathsf{D}\ G)[f\ x])\\
&\mathsf{forget}_i = \lambda g.\ \pi_1\ (\mathsf{repr}\ g) && \mathsf{remember} = \lambda x.\ \mathsf{unrepr}\ (x, \mathsf{refl})\,.
\end{aligned}$$

Clearly $|\mathcal{R}\ \mathsf{forget}_i| = |\mathcal{R}\ \mathsf{remember}| = \mathtt{id}$.

## 4 Implementation

Superfluid is a programming language with dependent types with quantities, inductive families and data representations. Its compiler is written in Haskell and the compilation target is JavaScript. After prior to code generation, the $\mathcal{R}$ transformation is applied to the elaborated core program, which erases all inductive constructs with defined representations. Then, a JavaScript program is extracted, erasing all irrelevant data by usage analysis similarly to Idris 2. As

[6] This can be implemented using quantitative type theory for example.

a result, with appropriate postulates in the prelude, we are able to represent Nat as JavaScript's `BigInt`, and List $T$/SnocList $T$/Vec $T$ $n$ as JavaScript's arrays with the appropriate index refinement, such that we can convert between them without any runtime overhead. The syntax of SUPERFLUID very closely mirrors the syntax given in the first half of this paper. It supports global definitions, inductive families, as well as postulates. Users are able to define custom representations for data types using `repr` blocks as defined earlier.

Currently we do not require proofs of eliminator coherence, but they are straightforward to add. We also treat the rule Repr-$C_i$ ((7)) rule as definitional in the implementation, at the cost of breaking confluence, but with the benefit of fewer manual transports. We are currently working on adding dependent pattern matching that is elaborated to internal eliminators, so that we can take advantage of the structural unification rules for data types [25]. We have written some of the examples in this paper in SUPERFLUID, which can be found in the `examples` directory.

## 5 Related work

Using inductive types as a form of abstraction was first explored by Wadler [28] through *views*. The extension to dependent types was developed by McBride and McKinna [26], as part of the Epigram project. Our system differs from views in the computational content of the abstraction; even with deforestation [29] views are not always zero-cost, but representations are. Atkey [11] shows how to generically derive inductive types which are refinements of other inductive types. This work could be integrated in our system to automatically generate representations for refined data types. Zero-cost data reuse when it comes to refinements of inductive types has been explored in the context of Church encoding in Cedille [20], but does not extend to custom representations.

Work by Allais [6,7] uses a combination of views, erasure by quantitative type theory, and universes of flattened data types to achieve performance improvements when working with serialised data in Idris 2. Our approach differs because we have access to 'native' data representations, so we do not need to rely on encodings. Additionally, they rely on heuristic compiler optimisations to erase their views. On the topic of memory layout optimisation, Baudon [12] develops Ribbit, a DSL for the specification of the memory representation of algebraic data types, which can specify techniques like struct packing and bit-stealing. To our knowledge however, this does not provide control over the indirection introduced by *inductive* types.

Dependently typed languages with extraction features, notably Coq [3] and Agda [1], have some overlapping capabilities with our approach, but they do not provide any of the correctness guarantees. Optimisation tricks such as the Nat-hack, and its generalisation to other types, can emulate a part of our system but are unverified and special casing in the compiler. Since the extended abstract version of this paper, an optimisation was merged into Idris 2 [2] to erase the forgetful and recomputation functions for reindexing list/maybe/number-like types. There also seems to be a demand for this kind of optimisation in Agda [4].

## 6 Future work

In the future we aim to expand the class of theories we consider, to include quotient-induction, induction-induction and induction-recursion. Representations for quotient-inductive types in particular could give rise to ergonomic ways of computing with more 'traditional' data structures such as hash maps or binary search trees. We could program inductively over these structures but extract programs without redundancy in data representation.

We could also look into automating the discovery of inductive algebras for 'known' classes of data types through metaprogramming [18]. This could reproduce optimisation techniques by modern proof assistants but as part of a standard library, with accompanying internal correctness proofs. It could also extend to identity function detection, by internalising the extraction step explored in section 3.8.

There are elements of our formalisation which should be developed further. We did not formulate decidability of equality and normalisation for $\lambda_{\text{REP}}$, which is needed for typechecking. We have developed a normalisation-by-evaluation [8] algorithm used in the implementation of Superfluid, but have not formally shown that it has the desired properties (although we expect it to). We are also working on a mechanisation of the developments of this paper in Agda.

On a more theoretical note, we define contexts in $\lambda_{\text{IND}}$ as pairs of a global context and a local context. In the language of categories with families (CWF) [15], our contexts are the Grothendieck construction $\int_{\Sigma:\mathsf{Glob}} \mathsf{Loc}\ \Sigma$. However, the base category $\mathsf{Glob}$ only has weakenings, which we have left implicit. Kaposi, Kovács and Altenkirch [22] showed that algebras for a theory form a CWF, whose initial objects are the inductive algebras. Thus we could have morphisms between global contexts when one's items can be represented in terms of another's. Then $\mathcal{R}$ could become a special case of a substitution calculus over representations.

## 7 Conclusion

This paper addresses some of the inefficiencies of inductive families in dependently typed languages by introducing custom runtime representations that preserve logical guarantees and simplicity of the surface language while optimising performance. These representations are formalised as inductive algebras, and come with a framework for reasoning about them: provably zero-cost conversions between indexed and unindexed views without runtime overhead.

The compilation process guarantees erasure of abstraction layers, translating high-level constructs to their defined implementations. Our hope is that by decoupling logical structure from runtime representation, ergonomic type-driven correctness can be leveraged without sacrificing performance.

## References


1. agda2hs documentation — agda2hs documentation. https://agda.github.io/agda2hs/, accessed: 2025-2-19
2. Make 'cons', 'nil', 'just' and 'nothing' constructors have uniform names by Z-snails · pull request #3486 · idris-lang/Idris2. https://github.com/idris-lang/Idris2/pull/3486, accessed: 2025-2-19
3. Program extraction — coq 8.13.2 documentation. https://coq.inria.fr/doc/v8.13/refman/addendum/extraction.html, accessed: 2025-2-19
4. Should agda optimise away the erasure from vec to list? · issue #7701 · agda/agda. https://github.com/idris-lang/Idris2/pull/3486, accessed: 2025-2-19
5. Adamek, J., Rosicky, J., Vitale, E.M.: Cambridge tracts in mathematics: Algebraic theories: A categorical introduction to general algebra series number 184. Cambridge University Press, Cambridge, England (18 Nov 2010), https://www.cambridge.org/academic/subjects/mathematics/logic-categories-and-sets/algebraic-theories-categorical-introduction-general-algebra?format=HB&isbn=9780521119221
6. Allais, G.: Builtin types viewed as inductive families. In: Programming Languages and Systems. pp. 113–139. Springer Nature Switzerland (2023), http://dx.doi.org/10.1007/978-3-031-30044-8_5
7. Allais, G.: Seamless, correct, and generic programming over serialised data. arXiv [cs.PL] (20 Oct 2023), http://arxiv.org/abs/2310.13441
8. Altenkirch, T., Geniet, C.: Big step normalisation for type theory (24 Sep 2020), https://drops.dagstuhl.de/storage/00lipics/lipics-vol175-types2019/LIPIcs.TYPES.2019.4/LIPIcs.TYPES.2019.4.pdf
9. Altenkirch, T., Kaposi, A.: Type theory in type theory using quotient inductive types. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 18–29. POPL '16, Association for Computing Machinery, New York, NY, USA (11 Jan 2016), https://doi.org/10.1145/2837614.2837638
10. Atkey, R.: Syntax and semantics of quantitative type theory. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 56–65. LICS '18, Association for Computing Machinery, New York, NY, USA (9 Jul 2018), https://doi.org/10.1145/3209108.3209189
11. Atkey, R., Johann, P., Ghani, N.: When is a type refinement an inductive type? In: Foundations of Software Science and Computational Structures, pp. 72–87. Lecture notes in computer science, Springer Berlin Heidelberg, Berlin, Heidelberg (2011), https://bentnib.org/inductive-refinement.pdf
12. Baudon, T., Radanne, G., Gonnord, L.: Bit-stealing made legal: Compilation for custom memory representations of algebraic data types. Proc. ACM Program. Lang. **7**(ICFP), 813–846 (31 Aug 2023), https://doi.org/10.1145/3607858
13. Boulier, S., Pédrot, P.M., Tabareau, N.: The next 700 syntactical models of type theory. In: Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs. pp. 182–194. CPP 2017, Association for Computing Machinery, New York, NY, USA (16 Jan 2017), https://doi.org/10.1145/3018610.3018620
14. Brady, E., McBride, C., McKinna, J.: Inductive families need not store their indices. In: Types for Proofs and Programs. pp. 115–129. Springer Berlin Heidelberg (2004), http://dx.doi.org/10.1007/978-3-540-24849-1_8

15. Castellan, S., Clairambault, P., Dybjer, P.: Categories with families: Unityped, simply typed, and dependently typed. arXiv [cs.LO] (1 Apr 2019), http://arxiv.org/abs/1904.00827
16. Cockx, J., Abel, A.: Elaborating dependent (co)pattern matching. Proc. ACM Program. Lang. **2**(ICFP), 1–30 (30 Jul 2018), https://doi.org/10.1145/3236770
17. Cockx, J., Tabareau, N., Winterhalter, T.: The taming of the rew: a type theory with computational assumptions. Proc. ACM Program. Lang. **5**(POPL), 1–29 (4 Jan 2021), https://doi.org/10.1145/3434341
18. Dagand, P.E.: The essence of ornaments. J. Funct. Programming **27**, e9 (Jan 2017), https://www.cambridge.org/core/services/aop-cambridge-core/content/view/4D2DF6F4FE23599C8C1FEA6C921A3748/S0956796816000356a.pdf/div-class-title-the-essence-of-ornaments-div.pdf
19. Dagand, P.E., McBride, C.: A categorical treatment of ornaments. arXiv [cs.PL] (16 Dec 2012), http://arxiv.org/abs/1212.3806
20. Diehl, L., Firsov, D., Stump, A.: Generic zero-cost reuse for dependent types. Proc. ACM Program. Lang. **2**(ICFP), 1–30 (30 Jul 2018), https://doi.org/10.1145/3236799
21. Goguen, H., McBride, C., McKinna, J.: Eliminating dependent pattern matching. In: Algebra, Meaning, and Computation, pp. 521–540. Lecture notes in computer science, Springer Berlin Heidelberg, Berlin, Heidelberg (2006), https://research.google.com/pubs/archive/99.pdf
22. Kaposi, A., Kovács, A., Altenkirch, T.: Constructing quotient inductive-inductive types. Proc. ACM Program. Lang. **3**(POPL), 1–24 (2 Jan 2019), http://dx.doi.org/10.1145/3290315
23. Kovács, A.: Staged compilation with two-level type theory. arXiv [cs.PL] (20 Sep 2022), http://arxiv.org/abs/2209.09729
24. Kovács, A.: Type-theoretic signatures for algebraic theories and inductive types. Ph.D. thesis (2023), https://andraskovacs.github.io/pdfs/phdthesis_compact.pdf
25. McBride, C., Goguen, H., McKinna, J.: A few constructions on constructors. In: Lecture Notes in Computer Science, pp. 186–200. Lecture notes in computer science, Springer Berlin Heidelberg, Berlin, Heidelberg (2006), http://www.e-pig.org/downloads/concon.pdf
26. Mcbride, C., Mckinna, J.: The view from the left. J. Funct. Programming **14**(1), 69–111 (Jan 2004), https://www.cambridge.org/core/services/aop-cambridge-core/content/view/F8A44CAC27CCA178AF69DD84BC585A2D/S0956796803004829a.pdf/div-class-title-the-view-from-the-left-div.pdf
27. Moon, B., Eades, III, H., Orchard, D.: Graded modal dependent type theory. In: Programming Languages and Systems. pp. 462–490. Springer International Publishing (2021), http://dx.doi.org/10.1007/978-3-030-72019-3_17
28. Wadler, P.: Views: a way for pattern matching to cohabit with data abstraction. In: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 307–313. POPL '87, Association for Computing Machinery, New York, NY, USA (1 Oct 1987), https://doi.org/10.1145/41625.41653
29. Wadler, P.: Deforestation: transforming programs to eliminate trees. Theor. Comput. Sci. **73**(2), 231–248 (1 Jun 1990), https://www.sciencedirect.com/science/article/pii/030439759090147A

# 8 Appendix

## 8.1 Utilities when working with algebras

We define in an Agda-like syntax, the construction of displayed algebras, sections and inductive algebras as telescopes in the object theory. We also define realisation functions which convert an internal telescopic construction into one which is partly lifted to the metatheory. Some implicit arguments are omitted for brevity.

**Realisation functions for algebras**

$\ulcorner \_ \urcorner^{\mathsf{alg}} : \mathsf{Tms}\ \Gamma\ (\mathsf{alg}\ T) \to \mathsf{Alg}\ T$

$\ulcorner (X, a) \urcorner^{\mathsf{alg}} := (\mathsf{El}\ X@, \ulcorner a \urcorner^{\mathsf{algebra}})\,.$

$\ulcorner \_ \urcorner^{\mathsf{algebra}} : \mathsf{Tms}\ \Gamma\ (\mathsf{algebra}\ T\ X) \to \mathsf{Algebra}\ T\ X$

$\ulcorner \bullet \urcorner^{\mathsf{algebra}}\ \{T = \bullet\}\ (v, \_)$ impossible by $v : \mathsf{Var}\ \bullet$

$\ulcorner (a', f) \urcorner^{\mathsf{algebra}}\ \{T = T' \rhd O\}\ (\mathsf{here}, d) := (f@)[\langle d \rangle]$

$\ulcorner (a', f) \urcorner^{\mathsf{algebra}}\ \{T = T' \rhd O\}\ (\mathsf{there}\ v, d) := \ulcorner a' \urcorner^{\mathsf{algebra}}\ (v, d)\,.$

**Displayed algebras**

$\mathsf{dispIn} : (O : \mathsf{Op}\ \Gamma\ P) \to ((X, x) : \mathsf{Alg}\ T) \to \mathsf{Ty}\ (\Gamma \rhd P \rhd X) \to \mathsf{Tel}\ \Gamma$

$\mathsf{dispIn}\ (\Pi\ A\ B)\ (X, x)\ M := \bullet \rhd (a : A) \rhd \mathsf{dispIn}\ B[\langle a \rangle]\ (X, x)\ M$

$\mathsf{dispIn}\ (\Pi\iota\ p\ B)\ (X, x)\ M := \bullet \rhd (y : X[\langle p \rangle]) \rhd (m : M[\langle p, y \rangle]) \rhd \mathsf{dispIn}\ B\ (X, x)\ M$

$\mathsf{dispIn}\ (\iota\ p)\ (X, x)\ M := \bullet\,.$

$\mathsf{dispInIn} : \mathsf{Tms}\ \Gamma\ (\mathsf{dispIn}\ O\ (X, x)\ M) \to \mathsf{Tms}\ \Gamma\ (\mathsf{in}\ O\ X)$

$\mathsf{dispInIn}\ \{\Pi\ A\ B\}\ (a, t)\ M := (a, \mathsf{dispInIn}\ \{B[\langle a \rangle]\}\ t\ M)$

$\mathsf{dispInIn}\ \{\Pi\iota\ p\ B\}\ (y, m, t)\ M := (y, \mathsf{dispInIn}\ \{B\}\ t\ M)$

$\mathsf{dispInIn}\ \{\iota\ p\}\ (\,)\ M := (\,)\,.$

$\mathsf{dispOut} : \{O : \mathsf{Op}\ \Gamma\ P\} \to (i : \mathsf{Tms}\ \Gamma\ (\mathsf{dispIn}\ O\ (X, x)\ M))$
$\quad \to \mathsf{Tms}\ \Gamma\ X[\langle \mathsf{out}\ (\mathsf{dispInIn}\ i)\ X \rangle]) \to \mathsf{Tms}\ \Gamma\ (P \rhd X)$

$\mathsf{dispOut}\ \{\Pi\ A\ B\}\ (a, t)\ u := \mathsf{dispOut}\ \{B[\langle a \rangle]\}\ t\ u$

$\mathsf{dispOut}\ \{\Pi\iota\ p\ B\}\ (y, m, t)\ u := \mathsf{dispOut}\ \{B\}\ t\ u$

$\mathsf{dispOut}\ \{\iota\ p\}\ x := (p, x)\,.$

$\mathsf{dispOp} : \mathsf{Var}\ T \to ((X, x) : \mathsf{Alg}\ T) \to \mathsf{Ty}\ (\Gamma \rhd P \rhd X) \to \mathsf{Ty}\ \Gamma$

$\mathsf{dispOp}\ v\ (X, x)\ M := \Pi\ (a : \mathsf{dispIn}\ (T\ v)\ (X, x)\ M)\ M[\langle \mathsf{dispOut}\ a\ (x\ (i, \mathsf{dispInIn}\ a)) \rangle]\,.$

$\mathsf{dispAlgebra} : ((X, x) : \mathsf{Alg}\ T) \to \mathsf{Ty}\ (\Gamma \rhd P \rhd X) \to \mathsf{Tel}\ \Gamma$

$\mathsf{dispAlgebra}\ \bullet\ X\ M := \bullet$
$\mathsf{dispAlgebra}\ (T \rhd O)\ (X, x, y)\ M := \mathsf{dispAlgebra}\ T\ (X, x)\ M \rhd \mathsf{dispOp}\ (X, y)\ \mathsf{here}\ M\,.$

$\mathsf{dispAlg} : \mathsf{Tel}\ (\Gamma \rhd \mathsf{alg}\ T)$
$\mathsf{dispAlg} := X\ x.\ \bullet \rhd (M : \Pi\ P\ \Pi\ X\ \mathcal{U}) \rhd \mathsf{dispAlgebra}\ \ulcorner (X, x) \urcorner^{\mathsf{algebra}}\ M\,.$

**Realisation functions for displayed algebras**
$\ulcorner \_ \urcorner^{\mathsf{dispAlg}} : \mathsf{Tms}\ (\Gamma\ (\mathsf{dispAlg}\ (X, x)) \to \mathsf{DispAlg}\ (X, x)$
$\ulcorner (M, m) \urcorner^{\mathsf{dispAlg}} := (\mathsf{El}\ M@@, \ulcorner m \urcorner^{\mathsf{dispAlgebra}})\,.$

$\ulcorner \_ \urcorner^{\mathsf{dispAlgebra}} : \mathsf{Tms}\ \Gamma\ (\mathsf{dispAlgebra}\ (X, x)\ M) \to \mathsf{DispAlgebra}\ (X, x)\ M$
$\ulcorner \bullet \urcorner^{\mathsf{dispAlgebra}}\ \{T = \bullet\}\ (v, \_)\ \text{impossible by}\ v : \mathsf{Var}\ \bullet$
$\ulcorner (m', u) \urcorner^{\mathsf{dispAlgebra}}\ \{T = T' \rhd O\}\ (\mathsf{here}, d) := (u@)[\langle d \rangle]$
$\ulcorner (m', u) \urcorner^{\mathsf{dispAlgebra}}\ \{T = T' \rhd O\}\ (\mathsf{there}\ v, d) := \ulcorner m' \urcorner^{\mathsf{dispAlgebra}}\ (v, d)\,.$

**Sections**
$\mathsf{sec} : \mathsf{Ty}\ (\Gamma \rhd \mathsf{alg}\ T \rhd \mathsf{dispAlg})$
$\mathsf{sec}\ \{P\} := X\ x\ M\ m.\ \Pi\ P\ \Pi\ X\ M\,.$

$\mathsf{apply} : \mathsf{Tms}\ \Gamma\ (\mathsf{in}\ O\ X) \to \mathsf{Sec}\ M \to \mathsf{Tms}\ \Gamma\ (\mathsf{in}\ O\ (X, x)\ M)$
$\mathsf{apply}\ \{O = \Pi\ A\ B\}\ (a, t)\ f := (a, \mathsf{apply}\ \{O = B[\langle a \rangle]\}\ t\ f)$
$\mathsf{apply}\ \{O = \Pi\iota\ p\ B\}\ (y, t)\ f := (y, f[\langle y \rangle], \mathsf{apply}\ \{O = B\}\ t\ f)$
$\mathsf{apply}\ \{O = \iota\ p\}\ (\,)\ f := (\,)\,.$

$\mathsf{cohOpRet} : (v : \mathsf{Var}\ T) \to \mathsf{Tms}\ \Gamma\ (\mathsf{dispIn}\ (Tv)\ (X, x)\ M)$
$\quad \to \mathsf{DispAlgebra}\ (X, x)\ M \to \mathsf{Sec}\ M \to \mathsf{Ty}\ \Gamma$
$\mathsf{cohOpRet}\ v\ i\ m\ f := \text{let}\ a = \mathsf{dispInIn}\ i\ \text{in}$
$\quad \mathsf{Id}\ f[\langle \mathsf{dispOut}\ i\ (x\ (v, a)) \rangle]\ (m\ (v, \mathsf{apply}\ a\ f))$

$\mathsf{cohOp} : \mathsf{Var}\ T \to ((X, x) : \mathsf{Alg}\ T) \to ((M, m) : \mathsf{DispAlg}\ (X, x)) \to \mathsf{Sec}\ M \to \mathsf{Ty}\ \Gamma$
$\mathsf{cohOp}\ v\ (X, x)\ (M, m)\ f :=$
$\quad \Pi\ (a : \mathsf{dispIn}\ (T\ v)\ (X, x)\ M)\ (\mathsf{cohOpRet}\ v\ a\ \ulcorner (M, m) \urcorner^{\mathsf{dispAlg}}\ f)\,.$

$\mathsf{coh} : ((X, x) : \mathsf{Alg}\ T) \to ((M, m) : \mathsf{DispAlg}\ (X, x)) \to \mathsf{Tel}\ (\Gamma \rhd \Pi\ P\ \Pi\ X\ M)$
$\mathsf{coh}\ \{T = \bullet\}\ X\ M := \bullet$
$\mathsf{coh}\ \{T = T' \rhd O\}\ (X, x, y)\ (M, m, n) :=$
$\quad f.\ (\mathsf{coh}\ \{T'\}\ (X, x)\ (M, m))[\langle f \rangle] \rhd \mathsf{cohOp}\ \mathsf{here}\ (X, y)\ (M, n)\ f\,.$

$\mathsf{section} : \mathsf{Tel}\ (\Gamma \rhd \mathsf{alg}\ T \rhd \mathsf{dispAlg})$

$\mathsf{section} := X\ x\ M\ m.\ \mathsf{sec}\ X\ x\ M\ m \rhd \mathsf{coh}\ \ulcorner(X,x)\urcorner^{\mathsf{alg}}\ \ulcorner(M,m)\urcorner^{\mathsf{dispAlg}}\,.$

**Realisation functions for sections and inductive algebras**

$\ulcorner\_\urcorner^{\mathsf{coh}} : \mathsf{Tms}\ \Gamma\ (\mathsf{coh}\ (X,x)\ (M,m))[\langle f\rangle] \to \mathsf{IntCoh}\ f@@$

$\ulcorner\bullet\urcorner^{\mathsf{coh}}\ \{T=\bullet\}\ (v,\_)$ impossible by $v : \mathsf{Var}\ \bullet$

$\ulcorner(c,u)\urcorner^{\mathsf{coh}}\ \{T=T'\rhd O\}\ (\mathsf{here},d) := (u@)[\langle d\rangle]$

$\ulcorner(c,u)\urcorner^{\mathsf{coh}}\ \{T=T'\rhd O\}\ (\mathsf{there}\ v,d) := \ulcorner c\urcorner^{\mathsf{coh}}\ (v,d)\,.$

$\ulcorner\_\urcorner^{\mathsf{section}} : \mathsf{Tms}\ \Gamma\ \mathsf{section}[\langle X,x,M,m\rangle] \to (f : \mathsf{Sec}\ \ulcorner(M,m)\urcorner^{\mathsf{dispAlg}}) \times \mathsf{IntCoh}\ f$

$\ulcorner(X,x,M,m,f,c)\urcorner^{\mathsf{dispAlg}} := (f@@, \ulcorner m\urcorner^{\mathsf{coh}})\,.$

$\ulcorner\_\urcorner^{\mathsf{indAlgebra}} : \mathsf{Tms}\ \Gamma\ (\Pi\ \mathsf{dispAlg}[\langle X\rangle]\ \mathsf{section}[\langle X\rangle])$

$\quad \to (m : \mathsf{DispAlg}\ X) \to (f : \mathsf{Sec}\ m) \times \mathsf{IntCoh}\ f$

$\ulcorner l\urcorner^{\mathsf{indAlgebra}}\ (M,m) := \ulcorner l@(\lambda\ \lambda\ M)@m\urcorner^{\mathsf{section}}$

## 8.2 Utilities when working with representations

We define the action of $\mathsf{repr}$ and $\mathsf{unrepr}$ on the algebra constructions above. We omit the definition of $\ulcorner\_\urcorner^{\mathsf{Unrepr}}$ as it mirrors $\ulcorner\_\urcorner^{\mathsf{Repr}}$.

$\_^{\mathsf{Repr}} : \mathsf{Tms}\ \Gamma\ (\mathsf{in}\ O\ X) \to \mathsf{Tms}\ \Gamma\ (\mathsf{in}\ O\ (\mathsf{Repr}\ X))$

$(a,t)^{\mathsf{Repr}}\ \{O=\Pi\ A\ B\} := (a, t^{\mathsf{Repr}}\ \{O=B[\langle a\rangle]\})$

$(x,t)^{\mathsf{Repr}}\ \{O=\Pi\iota\ p\ B\} := (\mathsf{repr}\ a, t^{\mathsf{Repr}}\ \{O=B\})$

$()^{\mathsf{Repr}}\ \{O=\iota\ p\} := ()\,.$

$\_^{\mathsf{Repr}} : \mathsf{Tms}\ \Gamma\ (\mathsf{dispIn}\ O\ (X,x)\ M) \to \mathsf{Tms}\ \Gamma\ (\mathsf{dispIn}\ O\ (X,x)\ (\mathsf{Repr}\ M))$

$(a,t)^{\mathsf{Repr}}\ \{O=\Pi\ A\ B\} := (a, t^{\mathsf{Repr}}\ \{O=B[\langle a\rangle]\})$

$(y,m,t)^{\mathsf{Repr}}\ \{O=\Pi\iota\ p\ B\} := (y, \mathsf{repr}\ m, t^{\mathsf{Repr}}\ \{O=B\})\,.$

$\_^{\mathsf{Repr}} : \mathsf{Algebra}\ T\ X \to \mathsf{Algebra}\ T\ (\mathsf{Repr}\ X)$

$f^{\mathsf{Repr}}\ (v,d) := \mathsf{repr}\ (f\ (v, d^{\mathsf{Unrepr}}))\,.$

$\_^{\mathsf{Repr}*} : \mathsf{Tms}\ \Gamma\ (\mathsf{dispIn}\ O\ (X,x)\ M)$

$\quad \to \mathsf{Tms}\ \Gamma\ (\mathsf{dispIn}\ O\ (X,x)^{\mathsf{Repr}}\ (p\ x.\ M[\langle p; \mathsf{unrepr}\ x\rangle]))$

$(a,t)^{\mathsf{Repr}*}\ \{O=\Pi\ A\ B\} := (a, t^{\mathsf{Repr}*}\ \{O=B[\langle a\rangle]\})$

$(y,m,t)^{\mathsf{Repr}*}\ \{O=\Pi\iota\ p\ B\} := (\mathsf{repr}\ y, m, t^{\mathsf{Repr}*}\ \{O=B\})$

$()^{\mathsf{Repr}*}\ \{O=\iota\ p\} := ()\,.$

$\_^{\mathsf{Repr}} : \mathsf{DispAlgebra}\ (X,x)\ M \to \mathsf{DispAlgebra}\ (X,x)\ (\mathsf{Repr}\ M)$

$f^{\mathsf{Repr}}\ (v, d) := \mathsf{repr}\ (f\ (v, d^{\mathsf{Unrepr}}))\,.$

$\_^{\mathsf{Repr}*} : \mathsf{DispAlgebra}\ (X, x)\ M \to \mathsf{DispAlgebra}\ (X, x)^{\mathsf{Repr}}\ (p\ x.\ M[\langle p; \mathsf{unrepr}\ x\rangle])$
$f^{\mathsf{Repr}}\ (v, d) := f\ (v, d^{\mathsf{Unrepr}})\,.$