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 [30], refined to inductive families like in Epigram [28], 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 [15] and the use of theories with irrelevant fragments [11,29], 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 height 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 [21] (which restricts the flexibility of data representation). This is the second problem we aim to address in this paper.

Wadler's views [30] 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 [28] 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 λ_{REP} to a core language with dependent types and inductive families λ_{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 λ_{IND} and its extension by representations λ_{REP} formulated as inductive algebras for theories, along with a translation to λ_{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

$$\textbf{data}\ \mathsf{Nat}\ \left\{\begin{array}{r}\mathsf{0} : \mathsf{Nat}\\ \mathsf{1+} : \mathsf{Nat} \to \mathsf{Nat}\end{array}\right\} \tag{1}$$

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

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

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

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

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

$$|\mathsf{elim}_{\mathsf{Nat}}\ P\ m_{\mathsf{0}}\ m_{\mathsf{1+}}\ s| = \mathsf{ubig\text{-}elim}\ |s|\ |m_{\mathsf{0}}|\ |m_{\mathsf{1+}}| \tag{4}$$

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

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

[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.

Nat is represented as the type 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 Nat eliminator, which are postulated as propositional equalities. This representation is valid in a context containing the primitives

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

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 Nat, such as addition, multiplication, and other numeric operations, in terms of 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 λ_{IND}, like rewriting rules [18], given that we have the appropriate lemmas to justify them in the context.

This will effectively erase the Nat type from the compiled program, replacing all occurrences with the UBig type and its primitives. In a way, the hard work is done by the postulates above; we expect that the underlying implementation of 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 Vec in terms of 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 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 Vec but

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

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

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

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

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

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 $\textsf{cong}\ (\textsf{1+})\ (\textsf{1+-inj}\ p) = p$ in the recursive case.

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

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

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

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

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

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

Now the hard work is done; Every time we are working with a $v : \textsf{Vec}\ T\ n$, 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 Σ-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 [11]. In section 3.8 we show how to formally identify computationally irrelevant conversion functions.

We can utilise this representation to convert between Vec and 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 G as some refinement of F then we must be able to provide a way to compute G's extra indices X from F, like we computed Vec's extra Nat index from List with 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 G is 'equivalent' to the algebraic ornament of F by the algebra defining comp (given by an isomorphism between the underlying polynomial functors), then it is also equivalent to the Σ-type above. The 'recomputation lemma' of algebraic ornaments [20] then arises from its projections. Our system allows us to *set* the representation of G as $\mathsf{F}^{\mathsf{comp}}$, so that the forgetful map from G to 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:

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

We can use the Fin type to maintain the invariant that the inventory has a maximum size. Additionally, we can index the Inventory type by the number of items it contains, which might be defined similarly to 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 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 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 Fin in the same way as Nat, so that reading the items field is a constant-time operation (we do not need to build a unary numeral). We can thus define the representation of 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 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 [8] explores how data descriptions that index into a flat buffer can be defined.

2.5 Transitivity

Representations are transitive, so in the previous example, the 'terminal' representation of Vec also depends on the representation of List. It is possible to define a custom representation for List itself, for example a heap-backed array or a finger tree, and 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 **repr**/**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 Buf type of byte buffers, we can consider the set of all types which are eventually represented as a 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 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 λ_{IND} with data representations. The base type theory is intensional Martin-Löf type theory (MLTT) [26] with a single universe $\mathcal{U} : \mathcal{U}$. We omit considerations of consistency and universe hierarchy, though these can be added if needed. All of the examples in the paper so far are written in a surface language that elaborates to λ_{IND}. In section 3.5, we define a modality Repr that allows us to convert between inductive types and their representations. We also define a translation from λ_{IND} to *extensional* MLTT, which 'elaborates away' all inductive families to their representations.

Syntaxes for languages are defined in an intrinsically well-formed manner quotiented by definitional equality [10], and with de-Bruijn indices for variables. Weakening of terms is generally left implicit to reduce syntactic noise, and often named notation is used when de-Brujin indices are implied. We use $(a : A) \to B$ for function types, $a \equiv_A a'$ for propositional equality, and $a = a' : A$ for definitional equality. Substitution is denoted with square brackets: if $\Gamma, A \vdash B$ and $\Gamma \vdash a : A$ then $\Gamma \vdash B[a]$. Besides the usual judgement forms of MLTT, we also have telescopic judgement forms:

$\Gamma \vdash \Delta\ \mathsf{tel}$	Δ is a telescope in Γ.
$\Gamma \vdash \delta :: \Delta$	δ is a spine (list of terms) matching telescope Δ.
$\Gamma \vdash \Delta = \Delta'$	Δ and Δ' are equal telescopes in Γ.
$\Gamma \vdash \delta = \delta' :: \Delta$	δ and δ' are equal spines matching telescope Δ.

Below are the formation rules for telescopes and term lists. Since everything is intrinsically well-formed, all required pre-conditions are implied for every inference rule.

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

$$\frac{\textsc{Tel-Empty}}{\Gamma \vdash \bullet\ \mathsf{tel}} \qquad \textsc{Tel-Extend}\ \frac{\Gamma \vdash \Delta\ \mathsf{tel} \qquad \Gamma, \Delta \vdash A\ \mathsf{type}}{\Gamma \vdash (\Delta, A)\ \mathsf{tel}}$$

$$\frac{\textsc{Spine-Empty}}{\Gamma \vdash (\,) :: \bullet} \qquad \textsc{Spine-Extend}\ \frac{\Gamma \vdash \delta :: \Delta \qquad \Gamma \vdash a : A[\delta]}{\Gamma \vdash (\delta, a) :: (\Delta, A)}$$

Fig. 1. Rules for forming telescopes and spines. Extending contexts by telescopes (such as Γ, Δ) is defined mutually with these rules.

We write $\Delta \to X$ for an iterated function type with codomain $\Gamma, \Delta \vdash X$, and $(\delta :: \Delta) \to X[\delta]$ when names are highlighted. We will often use the notation $\delta.y$ to extract a certain index y from a spine δ. This is used when we define telescopes using named notation. For example, if $\delta :: (X : A \to \mathcal{U}, y : (a : A) \to X\ a)$, then $\delta.X : A \to \mathcal{U}$ and $\delta.y : (a : A) \to \delta.X\ a$.

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.

Algebraic signatures [6,25] 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 type theory. Thus, we define two new judgement forms:

$\Gamma \vdash S\ \mathsf{sig}\ \Delta$	S is a signature with indices Δ in context Γ
$\Gamma \vdash O\ \mathsf{op}\ \Delta$	O is an operation with indices Δ in context Γ

Signatures are lists of operations, and operations build up constructor types.

$$\textsc{Sig-Empty}\ \frac{\Gamma \vdash \Delta\ \mathsf{tel}}{\Gamma \vdash \epsilon\ \mathsf{sig}\ \Delta} \qquad \textsc{Sig-Extend}\ \frac{\Gamma \vdash \Delta\ \mathsf{tel} \qquad \Gamma \vdash O\ \mathsf{op}\ \Delta}{\Gamma \vdash (S \rhd O)\ \mathsf{sig}\ \Delta}$$

$$\textsc{Op-Ext}\ \frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma, A \vdash O\ \mathsf{op}\ \Delta}{\Gamma \vdash (A \ \to_{\mathsf{ext}} O)\ \mathsf{op}\ \Delta} \qquad \textsc{Op-Int}\ \frac{\Gamma \vdash \delta :: \Delta \qquad \Gamma \vdash O\ \mathsf{op}\ \Delta}{\Gamma \vdash (\iota\ \delta \to_{\mathsf{int}} O)\ \mathsf{op}\ \Delta} \qquad \textsc{Op-Ret}\ \frac{\Gamma \vdash \delta :: \Delta}{\Gamma \vdash (\iota\ \delta)\ \mathsf{op}\ \Delta}$$

Fig. 2. Rules for forming signatures and operations.

Each signature is described by an associated telescope of indices Δ, and a *finite list* of operations:

- $A \to_{\mathsf{ext}} O$, a (dependent) abstraction over some type A from the type theory, of another operation O.
- $\iota\ \delta \to_{\mathsf{int}}\ O$, an abstraction over a recursive occurence of the object being defined, with indices δ, of another operation O.
- $\iota\ \delta$, a constructor of the object being defined, with indices δ.

Example 1 (Natural numbers). The signature for natural numbers is indexed by the empty telescope $\bullet$. It is defined as $\Gamma \vdash (\epsilon \rhd \iota\ () \rhd \iota\ () \to_{\mathsf{int}} \iota\ ())\ \mathsf{sig}\ \bullet$. We can add labels to aid readability, and omit index spines if they are empty:

$$\Gamma \vdash (\epsilon \rhd \mathit{zero} : \iota \rhd \mathit{succ} : \iota \ \to_{\mathsf{int}} \iota)\ \mathsf{sig}\ \bullet\ .$$

Example 2 (Vectors). The signature for vectors of elements of type T and length n is indexed by the telescope $(T : \mathcal{U}, n : \mathbb{N})$, defined as

$$\begin{aligned}\Gamma \vdash (\epsilon \rhd \mathit{nil} &: (T : \mathcal{U}) \to_{\mathsf{ext}} \iota\ T\ \mathit{zero} \\ &\rhd \mathit{cons} : (T : \mathcal{U}) \to_{\mathsf{ext}} (n' : \mathbb{N}) \to_{\mathsf{ext}} (t : T) \to_{\mathsf{ext}} \iota\ T\ n' \to_{\mathsf{int}} \iota\ T\ (\mathit{succ}\ n')) \\ &\mathsf{sig}\ (T : \mathcal{U}, n : \mathbb{N})\,.\end{aligned}$$

Later we will see how we can use the signature in example 1 to define the type of natural numbers $\Gamma \vdash \mathbb{N}\ \mathsf{type}$.

Notice that this syntax only allows occurrences of ι in positive positions, which is a requirement for inductive types. Different classes of algebraic signatures, theories and quantification are explored in detail by Kovács [25]. We make no distinction between parameters and indices, though it is possible to add parameters by augmenting the syntax for signatures with an extra telescope that must be uniform across operations.

3.2 Interpreting signatures in the type theory

In order to make use of our definition for algebraic signatures, we would like to be able to interpret their structure as types in the type theory.

Algebras An *algebra* for a signature $\Gamma \vdash S \ \mathsf{sig}\ \Delta$ and carrier type $\Gamma, \Delta \vdash X \ \mathsf{type}$ interprets the structure of S in terms of the type X. Concretely, this produces a telescope which matches the structure of S but replaces each occurrence of $\iota\ \delta$ with $X[\delta]$. The function arrows $\to_{\mathsf{int}}$ and $\to_{\mathsf{ext}}$ in S are interpreted as the function arrow $\to$ of the type theory.

Example 3 (Natural numbers). An algebra for the signature of natural numbers (example 1) over a carrier $\Gamma \vdash N \ \mathsf{type}$ is a spine matching the telescope

$$\Gamma \vdash (\mathit{zero} : N,\ \ \mathit{succ} : N \to N)\ \mathsf{tel}.$$

Example 4 (Vectors). An algebra for the signature of vectors (example 4) over a carrier $\Gamma, T : \mathcal{U}, n : \mathbb{N} \vdash V \ \mathsf{type}$ is a spine matching the telescope

$$\begin{aligned}\Gamma \vdash (&\mathit{nil} : (T : \mathcal{U}) \to V[T, \mathit{zero}],\\ &\quad \mathit{cons} : (T : \mathcal{U}) \to (n' : \mathbb{N}) \to (t : T) \to (\mathit{ts} : V[T, n']) \to V[T, \mathit{succ}\ n'])\ \mathsf{tel}.\end{aligned}$$

Induction The actual type of natural numbers $\Gamma \vdash \mathbb{N}\ \mathsf{type}$ is just the carrier of an algebra over the signature of natural numbers. In particular, the 'best' such algebra: one whose operations do not forget any information. In the language of category theory, this is the initial algebra in the category of algebras over the signature of natural numbers. An equivalent formulation of initial algebras is algebras which support *induction*, which is more suitable for our (syntactic) purposes. An algebra $\alpha :: (\mathit{zero} : X,\ \mathit{succ} : X \to X)$ for natural numbers supports induction if:

> For any type family $X \vdash Y\ \mathsf{type}$, if we can construct a $\mathit{zero}_Y : Y[\alpha.\mathit{zero}]$ and a $\mathit{succ}_Y : (x : X) \to Y[x] \to Y[\alpha.\mathit{succ}\ x]$, then we can construct a $\sigma[x] : Y[x]$ for all $x : X$.

The type family Y is commonly called the *motive*, and $(\mathit{zero}_Y, \mathit{succ}_Y)$ are the *methods*. The produced term family $x : X \vdash \sigma : Y[x]$ is a *section* of the type family Y. Induction also requires that the section acquires its values from the provided methods. This means that

$$\sigma[\alpha.\mathit{zero}] = \mathit{zero}_Y \qquad \sigma[\alpha.\mathit{succ}\ x] = \mathit{succ}_Y\ x\ \sigma[x].$$

We call these *coherence conditions*. A section that satisfies these conditions is called a *coherent section*. These equations might or might not hold definitionally. In the former case, we have the definitional equalities

$$\begin{aligned}\Gamma \vdash \sigma[\alpha.\mathit{zero}] &= \mathit{zero}_Y : Y[\alpha.\mathit{zero}]\\ \Gamma, x : X \vdash \sigma[\alpha.\mathit{succ}\ x] &= \mathit{succ}_Y\ x\ \sigma[x] : Y[\alpha.\mathit{succ}\ x].\end{aligned}$$

In the latter case, we have a spine of propositional equality witnesses

$$\begin{aligned}\Gamma \vdash \sigma_{\mathrm{coh}} :: (&\mathit{zero}_{\mathrm{coh}} : \sigma[\alpha.\mathit{zero}] \equiv \mathit{zero}_Y,\\ &\quad \mathit{succ}_{\mathrm{coh}} : (x : X) \to \sigma[\alpha.\mathit{succ}\ x] \equiv \mathit{succ}_Y\ x\ \sigma[x]).\end{aligned}$$

Displayed algebras Notice that the methods $(zero_Y, succ_Y)$ look like an algebra for the signature of natural numbers too, but their carrier is now a type family over another algebra carrier X, and the types of the operations mention both X and Y, using α to go from Δ to X. These are *displayed algebras*. In general, a displayed algebra for a signature $\Gamma \vdash S$ sig Δ, algebra α for S over carrier $\Gamma, \Delta \vdash X$ type, and carrier family $\Gamma, \Delta, X \vdash Y$ type, interprets the structure of S in terms of both X and Y. This produces a telescope which matches the structure of S but replaces each recursive occurrence $\iota\ \delta$ with an argument $x : X$ as well as an argument $y : Y[x]$. Each operation returns a $Y[r]$ with r computed from α. Again, the function arrows in S are interpreted as function types in the type theory.

Example 5 (Vectors). A displayed algebra for an algebra $(nil, cons)$ for vectors (example 4) over a carrier family $\Gamma, T : \mathcal{U}, n : \mathbb{N}, v : V[T, n] \vdash W$ type is a spine matching the telescope

$$\begin{aligned}\Gamma \vdash (&nil_W : (T : \mathcal{U}) \to W[T, zero, nil\ T],\\ &cons_W : (T : \mathcal{U}) \to (n' : \mathbb{N}) \to (t : T) \to (ts : V[T, n'])\\ &\quad \to (ts_W : W[T, n', ts]) \to W[T, succ\ n', cons\ T\ n'\ t\ ts])\ \mathsf{tel}.\end{aligned}$$

In practice, in a call-by-value setting, it is desirable for the inductive hypotheses of a displayed algebra (ts_W above) to be *lazy* values. This improves performance when the inductive hypotheses are not needed. Lazy values of T can be implemented as functions out of the unit type: $1 \to T$. We leave this as an implementation detail.

Finally, we come to the central definition that classifies the algebras which support induction:

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

The elimination rule for inductive data types in programming languages is exactly this: given any motive and methods (a displayed algebra), we get a dependent function from the type of the scrutinee to the type of the motive (a section). Furthermore this function satisfies some appropriate computation rules: when we plug in a constructor, we get the result of the method corresponding to it (the coherence conditions). Usually in programming languages, these conditions hold definitionally, as they are the primary means of computation with data.

Uniqueness of inductive algebras As mentioned earlier, inductive algebras are equivalent to initial algebras. Since initial objects are unique up to isomorphism, this suggests the following result:

Theorem 1. *For any given signature S, if we can construct two inductive algebras for S, over carriers X and X', then there exists an isomorphism of types $X \simeq X'$. Moreover, this isomorphism extends to the category of S-algebras, meaning it respects the structure of S.*

Proof. *See section 8.2.*

3.3 Defining algebras and friends

In order to utilise these constructions for our type system, we now explicitly define the follwing objects:

$\Gamma \vdash S^{\mathsf{alg}}\ X\ \mathsf{tel}$	Algebras for a signature $\Gamma \vdash S\ \mathsf{sig}\ \Delta$ over a carrier $\Gamma, \Delta \vdash X\ \mathsf{type}$.
$\Gamma \vdash \alpha^{\mathsf{dispAlg}}\ Y\ \mathsf{tel}$	Displayed algebras for an algebra $\Gamma \vdash \alpha :: S^{\mathsf{alg}}\ X$ over a motive $\Gamma, \Delta, X \vdash Y\ \mathsf{type}$.
$\Gamma \vdash \beta^{\mathsf{coh}}\ \sigma\ \mathsf{tel}$	Propositional coherences for a section $\Gamma, \Delta, X \vdash \sigma : Y$ of a displayed algebra $\Gamma \vdash \beta :: \alpha^{\mathsf{dispAlg}}\ Y$.

All constructions labelled with superscripts are not part of the syntax of the type system, but rather functions in the metatheory which compute syntactic objects such as telescopes.

The algebras for a signature are defined by case analysis on S:

$$\boxed{\Gamma \vdash S^{\mathsf{alg}}\ X\ \mathsf{tel}}$$

$$\epsilon^{\mathsf{alg}}\ X = \bullet \qquad (S' \rhd O)^{\mathsf{alg}}\ X = (S'^{\mathsf{alg}}\ X,\ (\nu :: O^{\mathsf{in}}\ X) \to X[\nu^{\mathsf{out}}])$$

An empty signature ϵ produces an empty telescope, while an extended signature $S' \rhd O$ produces a telescope extended with a function corresponding to O. This function goes from the inputs of O interpreted in X, to X evaluated at the output indices. The inputs and outputs of each operation O in an algebra are defined by case analysis on O:

$$\boxed{\Gamma \vdash O^{\mathsf{in}}\ X\ \mathsf{tel}}$$

$$(A \to_{\mathsf{ext}} O')^{\mathsf{in}}\ X = (a : A,\ O'[a]^{\mathsf{in}}\ X)$$
$$(\iota\ \delta \to_{\mathsf{int}} O')^{\mathsf{in}}\ X = (x : X[\delta],\ O'^{\mathsf{in}}\ X)$$
$$(\iota\ \delta)^{\mathsf{in}}\ X = \bullet$$

$$\boxed{\Gamma \vdash \{O\}\ \nu^{\mathsf{out}} :: \Delta}$$

$$\{O = A \to_{\mathsf{ext}} O'\}\ (a, \nu')^{\mathsf{out}} = \nu'^{\mathsf{out}}$$
$$\{O = \iota\ \delta \to_{\mathsf{int}} O'\}\ (x,\ \nu')^{\mathsf{out}} = \nu'^{\mathsf{out}}$$
$$\{O = \iota\ \delta\}\ ()^{\mathsf{out}} = \delta$$

A similar construction can be performed for displayed algebras over algebras. Displayed algebras are defined by case analysis on S, which is an implicit parameter of $-^{\mathsf{dispAlg}}$.

$$\boxed{\Gamma \vdash \{S\}\ \alpha^{\mathsf{dispAlg}}\ Y\ \mathsf{tel}}$$

$$\{S = \epsilon\}\ ()^{\mathsf{dispAlg}}\ Y = \bullet$$
$$\{S = S' \rhd O\}\ (\alpha', \alpha_O)^{\mathsf{dispAlg}}\ Y = (\alpha'^{\mathsf{dispAlg}}\ Y,\ (\mu :: \alpha_O^{\mathsf{dispIn}}\ Y) \to Y[\mu^{\mathsf{dispOut}}])$$

We provide the definitions of $-^{\mathsf{dispIn}}$ and $-^{\mathsf{dispOut}}$ in the appendix (section 8.2), but they are similar to the definitions of $-^{\mathsf{in}}$ and $-^{\mathsf{out}}$.

Next we define the coherence conditions for a displayed algebra as a telescope

$$\boxed{\Gamma \vdash \{S\}\ \{\alpha\}\ \beta^{\mathsf{coh}}\ \sigma\ \mathsf{tel}}$$
$$\{S = \epsilon\}\ \{\alpha = (\,)\}\ (\,)^{\mathsf{coh}}\ \sigma = \bullet$$
$$\{S = S' \rhd O\}\ \{\alpha = (\alpha', \alpha_O)\}\ (\beta', \beta_O)^{\mathsf{coh}}\ \sigma$$
$$= (\beta'^{\mathsf{coh}}\ \sigma,\ (\nu :: O^{\mathsf{in}}\ X) \to \sigma[\alpha_O\ \nu] \equiv \beta_O\ (\sigma \,\$\, \nu))$$

The notation $\sigma \,\$\, \nu$ applies the section σ to the input ν, yielding a displayed input by sampling the section to get the inductive hypotheses (defined in section 8.2)

Now we can define induction for an algebra α as a type

$$\boxed{\Gamma \vdash \{S\}\ \alpha^{\mathsf{ind}}\ \mathsf{type}}$$
$$\{S\}\ \alpha^{\mathsf{ind}} = (Y : (\delta :: \Delta) \to X[\delta] \to \mathcal{U}) \to (\beta :: \alpha^{\mathsf{dispAlg}}\ (\delta.\ x.\ Y\ \delta\ x))$$
$$\to (\sigma : (\delta :: \Delta) \to (x : X[\delta]) \to Y\ \delta\ x)) \times \beta^{\mathsf{coh}}\ (\delta.\ x.\ \sigma\ \delta\ x).$$

where Y is the motive, β are the methods, and σ is the output section which must satisfy the propositional coherence conditions. Finally, we can package an inductive algebra over a signature as a telescope

$$\boxed{\Gamma \vdash S^{\mathsf{indAlg}}\ \mathsf{tel}}$$
$$S^{\mathsf{indAlg}} = (X : \Delta \to \mathcal{U},\ \alpha :: S^{\mathsf{alg}}\ (\delta.\ X\ \delta),\ \kappa : \alpha^{\mathsf{ind}}),$$

by collecting the carrier X, algebra α and induction κ all together.

3.4 Constructing inductive families

We now extend intensional MLTT with a type for inductive families, which we denote $\mathsf{data}_\Delta\ S\ \gamma$. This type defines an inductive family matching a signature S with indices Δ, together with an inductive algebra γ which 'implements' the signature S. Notice that this is different to the usual way that inductive families are defined in type theory, for example W-types [5], where all we need to provide is a signature. Here, we must also implement the signature and prove the induction principle by providing γ, rather than it being 'provided' by the type theory. For example, if our type theory W-types, then we can implement the signature using W-types. In effect, this will later allow us to 'elaborate away' inductive definitions to their defined representations. This leads us to the formal definition of a representation:

Definition 2. *A* representation *of a signature S is an inductive algebra for S.*

In section 3.4 we define the data type, and its corresponding introduction, elimination, and computation rules.

$$\textsc{Data-Form}\quad \frac{\Gamma \vdash S\ \mathsf{sig}\ \Delta \qquad \Gamma \vdash \gamma :: S^{\mathsf{indAlg}} \qquad \Gamma \vdash \delta :: \Delta}{\Gamma \vdash \mathsf{data}_\Delta\ S\ \gamma\ \delta\ \mathsf{type}}$$

$$\textsc{Data-Intro}\quad \frac{O \in S \qquad \Gamma \vdash \nu :: O^{\mathsf{in}}\ (\mathsf{data}_\Delta\ S\ \gamma)}{\Gamma \vdash \mathsf{ctor}_{S.O}\ \nu : \mathsf{data}_\Delta\ S\ \gamma\ \nu^{\mathsf{out}}}$$

$$\textsc{Data-Elim}\quad \frac{\begin{array}{c}\Gamma,\ \delta :: \Delta,\ \mathsf{data}_\Delta\ S\ \gamma\ \delta \vdash M\ \mathsf{type} \\ \Gamma \vdash \beta :: \mathsf{ctor}_S^{\mathsf{dispAlg}}\ M \qquad \Gamma \vdash \delta :: \Delta \qquad \Gamma \vdash x : \mathsf{data}_\Delta\ S\ \gamma\ \delta\end{array}}{\Gamma \vdash \mathsf{elim}_S\ M\ \beta\ \delta\ x : M[\delta, x]}$$

$$\textsc{Data-Comp}\quad \frac{\begin{array}{c}O \in S \qquad \Gamma \vdash \nu :: O^{\mathsf{in}}\ (\mathsf{data}_\Delta\ S\ \gamma) \\ \Gamma,\ \delta :: \Delta,\ \mathsf{data}_\Delta\ S\ \gamma\ \delta \vdash M\ \mathsf{type} \qquad \Gamma \vdash \beta :: \mathsf{ctor}_S^{\mathsf{dispAlg}}\ M\end{array}}{\Gamma \vdash \mathsf{elim}_S\ M\ \beta\ \nu^{\mathsf{out}}\ (\mathsf{ctor}_{S.O}\ \nu) = \beta_O\ (\mathsf{elim}_S\ M\ \beta\ \$\ \nu) : M[\nu^{\mathsf{out}}, \mathsf{ctor}_{S.O}\ \nu]}$$

Fig. 3. Rules for data types, constructors and eliminators. We write $O \in S$ to indicate that O is an operation in the signature S. We write α_O to extract the telescope element corresponding to operation O from the algebra α for S.

Constructors form an algebra for the signature S over $\mathsf{data}_\Delta\ S\ \gamma$, denoted by $\mathsf{ctor}_S = (\mathsf{ctor}_{S.O})_{O \in S}$. Similarly, the eliminator forms a coherent section over the constructor algebra, which holds definitionally.

One might think, what do we gain by adding data to the theory? If we can provide an inductive algebra γ for a signature S ourselves, then why not just use γ directly? The reason is that by having a primitive for inductive types, we can take advantage of their properties in an extensional way. For example, an induction principle suggests that the constructors corresponding to each method are disjoint. Since constructors ctor are a primitive term in the theory, we can make use of this when formulating a unification algorithm. Aside from disjointness, we can also rely on other properties such as injectivity, acyclicity, and 'no-confusion.' McBride [27] originally explored the properties which arise from the existence of induction principles, or equivalently, initiality.

For example, if we have an inductive algebra $(N, \mathit{zero}_N, \mathit{succ}_N, \mathit{elim}_N)$ for the natural numbers signature NatSig, we can prove propositionally that for all $x : N$, $\mathit{zero}_N \neq \mathit{succ}_N\ x$, by invoking elim_N. However, the typechecker does not know this fact; it is not derivable as a definitional equality contradiction. However, it *is* derivable definitionally that for all $x : \mathbb{N}$, $\mathsf{ctor}_{\mathit{zero}} \neq \mathsf{ctor}_{\mathit{succ}}\ x$, where $\mathbb{N} = \mathsf{data}\ \mathit{NatSig}\ (N, \mathit{zero}_N, \mathit{succ}_N, \mathsf{elim}_N)$ by the metatheoretic quotient-inductive definition of the syntax not equating ctor_i and ctor_j unless $i = j$.

Another way to state this is that the existence of data enables the use of *dependent pattern matching* on its inhabitants. Nested pattern matching on $\mathbb{N}$, for example, can be elaborated to invocations of $\mathsf{elim}_{\mathsf{NatSig}}$, which has the expected computation rules as shown in section 3.4. Converting dependent pattern matching to eliminators has been explored in-depth by Goguen, McBride and McKinna

[22], as well as by Cockx and Abel [17] in absence of Axiom K. These results can be used for pattern matching on data without significant modification.

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

So far we are able to construct data types using the $\mathsf{data}_{\Delta}\ S\ \gamma$ type constructor. These data types are themselves implemented in terms of inductive algebras. However, the rules for data types do not utilise them. We would like to be able to relate data types to their underlying inductive algebras. One reason is to avoid unnecessary computation. If we have a type X which has inductive algebras (X, α, κ) and (X, α', κ') for two signatures S and S' respectively, then we can form the data types $D = \mathsf{data}_{\Delta}\ S\ (X, \alpha, \kappa)$ and $D' = \mathsf{data}_{\Delta}\ S'\ (X, \alpha', \kappa')$ and make use of the structural properties of initiality. However, we would also like to be able to freely convert between X, D and D' without incurring any runtime cost. After all, D and D' are meant to be translated away to their underlying representation, X. This argument can also be made in the context of theorem proving. Sometimes it is easier to prove a property about D or D', due to their structure, but we should be able to 'transport' the property to X.

To make use of these conversions in a computationally-irrelevant manner, while still retaining the fact that D, D' and X are distinct types, we introduce a modality

$$\mathsf{Repr} : \mathcal{U} \to \mathcal{U}\ .$$

which takes types to their representations. This modality respects definitionally all the type formers of the theory, other than data.

Subuniverse of concrete types We can view the image of Repr as a subuniverse of $\mathcal{U}$. The restriction to its image

$$\mathsf{Repr} : \mathcal{U} \to \mathcal{U}_C\ .$$

targets a universe of types which do not contain any data types.

We extend the language λ_{IND} to form λ_{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

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

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 λ_{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 repr takes a term of type T to its representation of type $\mathsf{Repr}\ T$, and unrepr undoes the effect of 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. 4.

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

Fig. 4. Coherence of the representation operators with substitutions, Π, 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 repr to the recursive part of the arguments a. The full construction can be found in section 8.3 of the appendix

One might be tempted to make this equality definitional too. Unfortunately, this would render conversion checking undecidable, because if one applies 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 repr and unrepr). There is no equivalent of 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 repr to the motive and methods:

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

Additionally, eliminating something using E should be the same as eliminating the representation of that thing using the represented eliminator Q:

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

Above we use more auxilliary definitions which represent the carriers of algebras, as well as displayed algebras (appendix section 8.3):

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

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 λ_{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 λ_{IND} by adding the equality reflection rule. General undecidability of conversion is not a problem because type checking is decidable for λ_{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 λ_{REP} [14] such that definitional equality is preserved. Overall, $\mathcal{R}$ preserves the structure of λ_{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

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

[5] Not formalised in this paper.

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

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

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

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

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

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

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

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

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

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

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

The equality constructors of the syntax of λ_{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. 4) are preserved by metatheoretic reflexivity on the other side, since all representation operators are erased. Finally, coherence rules for definitions F and eliminators 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 2. *$\mathcal{R}$ preserves typing and definitional equality:* $(t_1, t_2 : \mathsf{Tm}_{\text{IND}}\ \Gamma\ A) \to t_1 = t_2 \to \mathcal{R}t_1 = \mathcal{R}t_2$.

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

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

Proof. The inclusion produces global contexts in λ_{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 λ_{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 λ_{IND} into some simply-typed calculus, denoted by vertical bars $|x|$. As opposed to $\mathcal{R}$, $\mathcal{E}$ operates on the unquotiented syntax of λ_{IND}. This can be justified by interpreting the quotient-inductive constructions from before into setoids [24]. 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 λ.

Definition 3. *A function $f : \mathsf{Tm}\ \Gamma\ (\Pi\ A\ B)$, is* computationally irrelevant *if $|\mathcal{R}A| = |\mathcal{R}B|$ and $|\mathcal{R}f| = \mathtt{id}$.*

Theorem 4. *The type former Repr is injective up to internal isomorphism, i.e.*

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

Moreover, this isomorphism is computationally irrelevant.

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

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

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

Consider extending our languages with usage-aware subset Σ-types

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

in such a way that 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 a result, with appropriate postulates in the prelude, we are able to represent Nat as JavaScript's `BigInt`, and $\mathsf{List}\ T/\mathsf{SnocList}\ T/\mathsf{Vec}\ T\ n$ 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 $\mathsf{Repr\text{-}C}_i$ ((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 [27]. We have written some of the examples in this paper in Superfluid, which can be found in the `examples` directory. Overall the implementation is a proof of concept, but we expect that our framework can be implemented in an existing language.

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

5 Related work

Using inductive types as a form of abstraction was first explored by Wadler [30] through *views*. The extension to dependent types was developed by McBride and McKinna [28], as part of the Epigram project. Our system differs from views in the computational content of the abstraction; even with deforestation [31] views are not always zero-cost, but representations are. Atkey [12] 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 [21], but does not extend to custom representations.

Work by Allais [7,8] 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 [13] 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 [19]. 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 λ_{REP}, which is needed for typechecking. We have developed a normalisation-by-evaluation [9] algorithm used in the implementation of SUPERFLUID, but have only sketched 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 λ_{IND} as pairs of a global context and a local context. In the language of categories with families (CWF) [16], our contexts are the Grothendieck construction $\int_{\Sigma:\mathsf{Glob}} \mathsf{Loc}\ \Sigma$. However, the base category Glob only has weakenings, which we have left implicit. Kaposi, Kovács and Altenkirch [23] 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 and usability. These representations are formalised as inductive algebras, and come with a framework for reasoning about them: provably zero-cost conversions between original and represented data.

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. Abbott, M., Altenkirch, T., Ghani, N.: Representing nested inductive types using W-types. In: Automata, Languages and Programming, pp. 59–71. Lecture notes in computer science, Springer Berlin Heidelberg, Berlin, Heidelberg (2004), `https://link.springer.com/chapter/10.1007/978-3-540-27836-8_8`
6. 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
7. 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
8. Allais, G.: Seamless, correct, and generic programming over serialised data. arXiv [cs.PL] (20 Oct 2023), http://arxiv.org/abs/2310.13441
9. 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
10. 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
11. 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
12. 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
13. 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
14. 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
15. 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/10.1007/3-540-24849-1_8
16. 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
17. Cockx, J., Devriese, D.: Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory. J. Funct. Prog. **28**(e12), e12 (Jan 2018), https://www.cambridge.org/core/services/aop-cambridge-core/content/view/E54D56DC3F5D5361CCDECA824030C38E/S095679681800014Xa.pdf/div-class-title-proof-relevant-unification-dependent-pattern-matching-with-only-the-axioms-of-you pdf
18. 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
19. 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

20. Dagand, P.E., McBride, C.: A categorical treatment of ornaments. arXiv [cs.PL] (16 Dec 2012), http://arxiv.org/abs/1212.3806
21. 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
22. 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
23. 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
24. Kovács, A.: Staged compilation with two-level type theory. arXiv [cs.PL] (20 Sep 2022), http://arxiv.org/abs/2209.09729
25. Kovács, A.: Type-theoretic signatures for algebraic theories and inductive types. Ph.D. thesis (2023), https://andraskovacs.github.io/pdfs/phdthesis_compact.pdf
26. Martin-Löf, P.: Intuitionistic type theory **1**, 1–91 (1984), http://people.csail.mit.edu/jgross/personal-website/papers/academic-papers-local/Martin-Lof80.pdf
27. 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
28. 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
29. 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
30. 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
31. 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 Implementation

The implementation of Superfluid can be found at https://github.com/kontheocharis/superfluid.

8.2 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

$\ulcorner\_\urcorner^{\mathsf{dispAlg}} : \mathsf{Tms}\ (\Gamma\ (\mathsf{dispAlg}\ (X, x)) \to \mathsf{DispAlg}\ (X, x)$

$\ulcorner(M, m)\urcorner^{\mathsf{dispAlg}} := (\mathsf{El}\ M@@, \ulcorner m\urcorner^{\mathsf{dispAlgebra}})\,.$

$\ulcorner\_\urcorner^{\mathsf{dispAlgebra}} : \mathsf{Tms}\ \Gamma\ (\mathsf{dispAlgebra}\ (X, x)\ M) \to \mathsf{DispAlgebra}\ (X, x)\ M$

$\ulcorner\bullet\urcorner^{\mathsf{dispAlgebra}}\ \{T = \bullet\}\ (v, \_)$ 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

$\mathsf{sec} : \mathsf{Ty}\ (\Gamma \rhd \mathsf{alg}\ T \rhd \mathsf{dispAlg})$

$\mathsf{sec}\ \{P\} := X\ x\ M\ m.\ \Pi\ P\ \Pi\ X\ M\,.$

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

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

$\mathsf{apply}\ \{O = \Pi\iota\ p\ B\}\ (y, t)\ f := (y, f[\langle y\rangle], \mathsf{apply}\ \{O = B\}\ t\ f)$

$\mathsf{apply}\ \{O = \iota\ p\}\ (\,)\ f := (\,)\,.$

$\mathsf{cohOpRet} : (v : \mathsf{Var}\ T) \to \mathsf{Tms}\ \Gamma\ (\mathsf{dispIn}\ (Tv)\ (X, x)\ M)$
$\quad \to \mathsf{DispAlgebra}\ (X, x)\ M \to \mathsf{Sec}\ M \to \mathsf{Ty}\ \Gamma$

$\mathsf{cohOpRet}\ v\ i\ m\ f := \mathrm{let}\ a = \mathsf{dispInIn}\ i\ \mathrm{in}$
$\quad \mathsf{Id}\ f[\langle \mathsf{dispOut}\ i\ (x\ (v, a))\rangle]\ (m\ (v, \mathsf{apply}\ a\ f))$

$\mathsf{cohOp} : \mathsf{Var}\ T \to ((X, x) : \mathsf{Alg}\ T) \to ((M, m) : \mathsf{DispAlg}\ (X, x)) \to \mathsf{Sec}\ M \to \mathsf{Ty}\ \Gamma$

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

$\mathsf{coh} : ((X, x) : \mathsf{Alg}\ T) \to ((M, m) : \mathsf{DispAlg}\ (X, x)) \to \mathsf{Tel}\ (\Gamma \rhd \Pi\ P\ \Pi\ X\ M)$

$\mathsf{coh}\ \{T = \bullet\}\ X\ M := \bullet$

$\mathsf{coh}\ \{T = T' \rhd O\}\ (X, x, y)\ (M, m, n) :=$
$\quad f.\ (\mathsf{coh}\ \{T'\}\ (X, x)\ (M, m))[\langle f\rangle] \rhd \mathsf{cohOp}\ \mathsf{here}\ (X, y)\ (M, n)\ f\,.$

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

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

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 $v : \mathsf{Var}\ \bullet$

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

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

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

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

$\ulcorner\_\urcorner^{\mathsf{indAlgebra}} : \mathsf{Tms}\ \Gamma\ (\Pi\ \mathsf{dispAlg}[\langle X\rangle]\ \mathsf{section}[\langle X\rangle])$
$\quad \to (m : \mathsf{DispAlg}\ X) \to (f : \mathsf{Sec}\ m) \times \mathsf{IntCoh}\ f$

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

8.3 Utilities when working with representations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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