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 [12], refined to inductive families like in Epigram [10]. However, representations come with compilation guarantees: a represented inductive family will not leave any runtime traces behind, without having to rely on automated optimisations such as deforestation [13]. 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. In addition, 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; for example, 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 [4] and the use of theories with irrelevant fragments [2,11], there is still a need to convert between different indexed views of the same data. For example, the function to convert from $\mathsf{List}\ T$ to $\mathsf{Vec}\ T\ n$ by forgetting the length index n is *not* erased by any current language with dependent types, unless vectors are defined as a refinement of lists with an erased length 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 [8] (which restricts the flexibility of data representation). This is the second problem we aim to address in this paper.

Wadler's views [12] 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 [10] 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. Despite deforestation handling this erasure to some extent, it is not guaranteed to erase all traces of the view from the program, and the optimisation might be difficult to predict.

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. In the final version of the paper, we plan to contribute the following:

- A dependent type system with inductive families λ_{IND}, and its extension by representations λ_{REP}.
- A formulation of common optimisations such as the 'Nat-hack', and similarly for other inductive types, as representations.
- A demonstration of zero-cost data reuse when reindexing by using representations.
- A translation from λ_{REP} to λ_{IND} that erases all inductive types with representations from the program.
- An implementation of this system and accompanying examples in Superfluid, a programming language with inductive types and dependent pattern matching.

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

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

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 signature containing the primitives

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

and propositional equalities

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

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 [6], given that we have the appropriate lemmas to justify them in the signature.

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

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

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

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

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

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

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

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

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

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 Vec as $\mathsf{List'}$, which would be written as

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

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 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 the full paper we will show how this erasure is achieved in practice in SUPERFLUID using Quantitative Type Theory [2].

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

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

and it holds by reflexivity that $\mathsf{forget\text{-}length}$ is a left inverse of $\mathsf{recall\text{-}length}$.

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 [7] 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:

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

We can use the Fin type to maintain the invariant that the inventory has a maximum size. Additionally, we can index the $\mathsf{Inventory}$ 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 $\mathsf{Inventory}$ as a pair of a byte buffer and a proof that the byte buffer contents correspond to an inventory record of a certain size. 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 $\mathsf{IsPlayer}$ which refines a byte buffer. We will provide the full details of this construction in the final paper.

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

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

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 $\mathsf{as\text{-}buf}$ 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 3.1 by exploring the semantics of data representations in terms of algebras for signatures. In section 3.3 we define a core language with inductive families λ_{IND}. In section 3.4, we extend this language with data representations to form λ_{REP}. All of the examples in the paper are written in a surface language that elaborates to λ_{REP}.

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. 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 [1]. Weakening of terms is generally also left implicit to reduce syntactic noise, and sometimes higher-order abstract syntax notation is used for the languages defined.

3.1 Algebras, displayed algebras and inductive algebras

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 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

[4] We do not guarrantee that an invocation of $\mathsf{as\text{-}buf}$ will be entirely erased, but rather that any invocation will eventually produce the identity function without having to perform a case analysis on its T subject.

can be used as a syntax for defining inductive families in a type theory. Thus, we define

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

The Theory sort represents a simple class of algebraic signatures. We do not call it $\mathsf{Signature}$ to avoid a name clash with another kind of 'signatures' that we will define later. Indeed, algebraic theories are a generalization of algebraic signatures. Each theory has an associated telescope of parameters P picked from some external type theory (not the metatheory), and contains a list of operations:

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

For example, the theory of natural numbers lives in the empty context has an empty telescope of parameters. It is defined by

$$\begin{aligned}
&\mathsf{NatTh} : \mathsf{Theory}\ \bullet\ \bullet \\
&\mathsf{NatTh} := \bullet \rhd \iota\ (\,) \rhd \Pi\iota\ (\,)\ (\iota\ (\,))
\end{aligned}$$

We can add labels to aid readability, omitting parameters if they are empty, and use $\Rightarrow$ for simple arrows:

$$\mathsf{NatTh} := \bullet \rhd \mathit{zero} : \iota \rhd \mathit{succ} : \iota \Rightarrow \iota$$

Notice that this syntax only allows occurrences of ι in positive positions, which is a requirement for inductive types. We could also add other constructors for operations. For example, we allow external quantification in some other type theory, but we could also allow quantification on the level of the *metatheory* (ignoring size issues) by an operation

$$\Pi_{\mathsf{meta}} : (A : \mathbf{Set}) \to (A \to \mathsf{Op}\ \Gamma\ P) \to \mathsf{Op}\ \Gamma\ P$$

We do not require this kind of abstraction for this paper but different classes of theories and quantification are explored in detail by Kovács [9].

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 carrier X and theory T defines a way to interpret the structure of T in terms of a type in a type theory $X : \mathsf{Ty}\ \Gamma$. This produces a type which matches the structure of T but replaces each occurrence of ι with X. The function arrows in T are interpreted as function arrows in the target universe. Algebras for the theory of natural numbers might look like

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

We have a choice in terms of how much we want to interpret T in the external type 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 type 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 theory T with carrier M mirrors the shape of T like an algebra does, but each recursive occurrence ι is now replaced by M applied to the corresponding value of the algebra. The displayed algebras for natural numbers are

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

The type M is often called the *motive*, and m the *methods*.

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

A section is a dependent function from X to M which takes its values from the displayed algebra. For natural numebrs,

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

A section is the output of induction: a proof of M for all X.

3.2 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 they can be found in section 6.1 of the appendix.

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

First, we define an ‘external’ version of algebras

$$\mathsf{Alg}\ T := (X : \mathsf{Ty}\ \Gamma) \times \mathsf{Algebra}\ T\ X$$
$$\mathsf{Algebra}\ T\ X := (a : \mathsf{Args}\ T\ X) \to \mathsf{Tm}\ \Gamma\ X[\langle \mathsf{out}\ a\rangle]\,.$$

which take some arguments Args and produce the output X evaluated at the appropriate index $\mathsf{out}\ a$ based on the arguments. The type Args is defined as $(v : \mathsf{Var}\ T) \times \mathsf{Tms}\ \Gamma\ (\mathsf{argsFor}\ (T\ v)\ X)$ for the obvious type $\mathsf{Var}\ T$ which indexes operations in theories, where

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

and out is defined as $\mathsf{outFor}\ \{T\ v\}\ a\ X$ where

$$\mathsf{outFor} : \{O : \mathsf{Op}\ \Gamma\ P\} \to \mathsf{Tms}\ \Gamma\ (\mathsf{argsFor}\ O\ X) \to \mathsf{Ty}\ \Gamma \to \mathsf{Tms}\ \Gamma\ P$$
$$\mathsf{outFor}\ \{\Pi\ A\ B\}\ (a, t)\ X := \mathsf{outFor}\ \{B[\langle a\rangle]\}\ t\ X$$
$$\mathsf{outFor}\ \{\Pi\iota\ p\ B\}\ (r, t)\ X := \mathsf{outFor}\ \{B\}\ t\ X$$
$$\mathsf{outFor}\ \{\iota\ p\}\ (\,)\ X := p\,.$$

We can also define a fully-internal version of algebras $\mathsf{algebra}\ T\ X$ as $\mathsf{Var}\ T$-indexed telescopes of

$$\mathsf{algebraFor} : \mathsf{Op}\ \Gamma\ P \to \mathsf{Ty}\ \Gamma \to \mathsf{Ty}\ \Gamma$$
$$\mathsf{algebraFor}\ (\Pi\ A\ B)\ X := \Pi\ (a : A)\ (\mathsf{algebraFor}\ B[\langle a\rangle]\ X)$$
$$\mathsf{algebraFor}\ (\Pi\iota\ p\ B)\ X := \Pi\ X[\langle p\rangle]\ (\mathsf{algebraFor}\ B\ X)$$
$$\mathsf{algebraFor}\ (\iota\ p)\ X := X[\langle p\rangle]\,.$$

which have evident ‘realisation’ functions into the metatheory

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

A similar construction can be performed for displayed algebras over external algebras

$$\mathsf{dispAlgebra}\ (X, x)\ M : \mathsf{Tel}\ \Gamma$$
$$\mathsf{DispAlg}\ (X, x) := (M : \mathsf{Ty}\ (\Gamma \rhd P \rhd X)) \times \mathsf{Tms}\ \Gamma\ (\mathsf{dispAlgebra}\ (X, x)\ M)$$

and we can use the realisation function for algebras to get internal displayed algebras over internal algebras

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

with evident realisation functions

$$\ulcorner\_\urcorner : \mathsf{Tms}\ \Gamma\ (\mathsf{dispAlg}[\langle t\rangle]) \to \mathsf{DispAlg}\ \ulcorner t\urcorner$$
$$\ulcorner\_\urcorner : \mathsf{Tms}\ \Gamma\ (\mathsf{dispAlgebra}\ t\ M) \to \mathsf{DispAlgebra}\ t\ M\,.$$

Finally, we can get external sections over displayed algebras

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

which have coherence rules using either the equality of the metatheory (Coh) or the propositional equality of the type theory (IntCoh). The apply function takes a displayed algebra, a section, and some arguments Args, and evaluates the section at those arguments.

These also have internal analogues

$$\begin{aligned}
&\mathsf{sec} : \mathsf{Tel}\ (\Gamma \triangleright \mathsf{alg}\ T \triangleright \mathsf{dispAlg})\\
&\mathsf{sec} := X\ x\ M\ m.\ \Pi\ P\ (\Pi\ X\ M)\\[1ex]
&\mathsf{coh} : \mathsf{Tel}\ (\Gamma \triangleright (X, x) : \mathsf{alg}\ T \triangleright (M, m) : \mathsf{dispAlg} \triangleright \Pi\ P\ (\Pi\ X\ M))\\[1ex]
&\mathsf{section} : \mathsf{Tel}\ (\Gamma \triangleright \mathsf{alg}\ T \triangleright \mathsf{dispAlg})\\
&\mathsf{section} := X\ x\ M\ m.\ \mathsf{sec} \triangleright \mathsf{coh}
\end{aligned}$$

which only use propositional equality. Once again, we can define realisation functions

$$\ulcorner\_\urcorner_0 : \mathsf{Tms}\ \Gamma\ (\mathsf{section}[\langle t\rangle, \langle m\rangle]) \to \mathsf{Sec}\ \ulcorner m\urcorner$$
$$\ulcorner\_\urcorner_1 : (t : \mathsf{Tms}\ \Gamma\ (\mathsf{section}[\langle t\rangle, \langle m\rangle])) \to \mathsf{IntCoh}\ \ulcorner t\urcorner_0$$

which produce only internal coherence proofs.

Finally we define a synonym for internal inductive algebras as a telescope

$$\mathsf{indAlg}\ T := \bullet \triangleright \mathsf{alg}\ T \triangleright \Pi\ \mathsf{dispAlg}\ \mathsf{section}\,.$$

Next we will make use of the external versions of algebras, displayed algebras, and sections in order to 'freely' add inductive algebras to the inductive syntax of a type theory in a strictly-positive but 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.

3.3 The core language, $\boldsymbol{\lambda_{\text{IND}}}$

The languge λ_{IND}, is a dependent type theory with Π, Id, 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 [5] by packaging named inductive constructions and function definitions into a signature $\Sigma : \mathsf{Sig}$, and indexing contexts by signatures. The contexts Con in the resulting theory are pairs $(\Sigma : \mathsf{Sig}) \times \mathsf{Loc}\ \Sigma$ where $\mathsf{Loc}\ \Sigma$ are local contexts given by a closed telescope of types as usual. Substitutions only occur between contexts of the same signature. Items in a signature Σ 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 theory $T : \mathsf{Theory}\ (\Sigma, \bullet)\ P$.

We reuse the Theory type defined in the previous section. This allows us to define data types such as vectors

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

We do not make a distinction between parameters and indices, though this might be desirable in an implementation because it generates elimination rules which are structurally uniform in the parameters. Nevertheless, our system is extensible to one with uniform parameters and we leave its formulation as an implementation detail.

In order to actually construct inductive types in λ_{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 Σ, and terms for its indices p, constructs the data type $(\mathsf{D}\ i)[p]$. The relation $I \in \Sigma$ finds items in a signature, to be thought of in a similar way to how $\mathsf{Var}\ \Gamma\ A$ defines variables for type A in a local context Γ. It is evidently a decidable relation, and is a proposition for a fixed I and Σ.

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 Tm in $\mathsf{Algebra}$ are part of the inductive syntax of the type theory. We can now construct, for example, natural numbers as

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

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 Σ, 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

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

Once again, expanding all the definitions involved leads to positive occurrences for the syntactic categories of the type theory.

Lemma 1. *The constructor algebra C_i of a data type i is inductive.*

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

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

$$\begin{aligned} \mathsf{F} &: \mathsf{def}\ P\ A\ t \in \Sigma \to \mathsf{Tm}\ (\Sigma, \Delta \triangleright P)\ A \\ \mathsf{P} &: \mathsf{post}\ P\ A \in \Sigma \to \mathsf{Tm}\ (\Sigma, \Delta \triangleright P)\ A \end{aligned}$$

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 sufficient to express a lot of the programs which can be written in a modern proof assistant. Next, we will explore how to extend this language to support data representations as explored earlier in the paper.

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

We extend the language λ_{IND} to form λ_{REP}, which allows users to define custom representations for inductive types and global functions. The machinery of algebras that we have developed in section 3.1 allows for a very direct definition of representations.

Definition 2. *A representation of a data type $\mathsf{data}\ P\ T$ is an inductive algebra for T.*

We first modify the syntax for signatures Sig to introduce representations:

$$\begin{aligned} &\bullet : \mathsf{Sig} \\ &\triangleright : (\Sigma : \mathsf{Sig}) \to \mathsf{Item}\ \Sigma \to \mathsf{Sig} \\ &\trianglerighteq : (\Sigma : \mathsf{Sig}) \to (I : \mathsf{Item}\ \Sigma) \to \mathsf{Rep}\ \Sigma\ I \to \mathsf{Sig}\,. \end{aligned}$$

Representations in turn are defined as

$$\begin{aligned}
&\mathsf{Rep} : (\Sigma : \mathsf{Sig}) \to \mathsf{Item}\ \Sigma \to \mathbf{Set} \\
&\mathsf{data\text{-}rep} : \mathsf{Tms}\ (\Sigma, \epsilon)\ (\mathsf{indAlg}\ T) \to \mathsf{Rep}\ \Sigma\ (\mathsf{data}\ P\ T) \\
&\mathsf{def\text{-}rep} : (x : \mathsf{Tm}\ (\Sigma, P)\ A) \to \mathsf{Tm}\ (\Sigma, P)\ (\mathsf{Id}\ x\ t) \to \mathsf{Rep}\ \Sigma\ (\mathsf{def}\ P\ A\ t)
\end{aligned}$$

We will write $\mathsf{data\text{-}rep}\ (R, r, Q)$ to unpack the telescope of an inductive algebra for a data representation with carrier $R : \mathsf{Tm}\ (\Sigma, \epsilon)\ (\Pi\ P\ \mathcal{U})$, algebra $r : \mathsf{Tm}\ (\Sigma, \epsilon)\ (\mathsf{algebra}[\langle R\rangle])$, and induction $Q : \mathsf{Tm}\ (\Sigma, \epsilon)\ (\Pi\ \mathsf{dispAlg}\ \mathsf{section})[\langle R, r\rangle]$.

We also include representations for definitions, where a definition can be represented by a term propositionally equal to original definition, but perhaps with better computational properties. 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 weakening of contexts and signatures, because each item can only be represented once in a signature.

To allow reasoning about representations internally to λ_{REP} we add a type former

$$\mathsf{Repr} : \mathsf{Ty}\ \Gamma \to \mathsf{Ty}\ \Gamma \tag{5}$$

along with two new terms in the syntax, forming an isomorphism

$$\mathsf{repr} : \mathsf{Tm}\ \Gamma\ T \simeq \mathsf{Tm}\ \Gamma\ (\mathsf{Repr}\ T) : \mathsf{unrepr}\,. \tag{6}$$

which holds by equality constructors and preserves Π, Id and universes. 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. 1.

$$\begin{aligned}
&\mathsf{reprr} : \mathsf{unrepr}\ (\mathsf{repr}\ t) \equiv t \\
&\mathsf{reprl} : \mathsf{repr}\ (\mathsf{unrepr}\ t) \equiv t \\[1ex]
&\mathsf{Repr\text{-}}\Pi : \mathsf{Repr}\ (\Pi\ T\ U) \equiv \Pi\ T\ (\mathsf{Repr}\ U) \\
&\mathsf{repr\text{-}}\lambda : \mathsf{repr}\ (\lambda\ u) \equiv \lambda\ (\mathsf{repr}\ u) \\
&\mathsf{unrepr\text{-}}\lambda : \mathsf{unrepr}\ (\lambda\ u) \equiv \lambda\ (\mathsf{unrepr}\ u) \\
&\mathsf{repr\text{-}@} : \mathsf{repr}\ (f\ @) \equiv (\mathsf{repr}\ f)\ @ \\
&\mathsf{unrepr\text{-}@} : \mathsf{unrepr}\ (f\ @) \equiv (\mathsf{unrepr}\ f)\ @ \\[1ex]
&\mathsf{repr}[] : \mathsf{repr}\ (t[\sigma]) \equiv (\mathsf{repr}\ t)[\sigma] \\
&\mathsf{unrepr}[] : \mathsf{unrepr}\ (t[\sigma]) \equiv (\mathsf{unrepr}\ t)[\sigma] \\
&\mathsf{Repr}[] : \mathsf{Repr}\ (T[\sigma]) \equiv (\mathsf{Repr}\ T)[\sigma]
\end{aligned}$$

$$\begin{aligned}
&\mathsf{Repr\text{-}}\mathcal{U} : \mathsf{Repr}\ \mathcal{U} \equiv \mathcal{U} \\
&\mathsf{repr\text{-}code} : \mathsf{repr}\ (\mathsf{code}\ T) \equiv \mathsf{code}\ T \\
&\mathsf{unrepr\text{-}code} : \mathsf{unrepr}\ (\mathsf{code}\ T) \equiv \mathsf{code}\ T \\[1ex]
&\mathsf{Repr\text{-}Id} : \mathsf{Repr}\ (\mathsf{Id}\ a\ b) \equiv \mathsf{Id}\ (\mathsf{repr}\ a)\ (\mathsf{repr}\ b) \\
&\mathsf{repr\text{-}refl} : \mathsf{repr}\ (\mathsf{refl}\ u) \equiv \mathsf{refl}\ (\mathsf{repr}\ u) \\
&\mathsf{unrepr\text{-}refl} : \mathsf{unrepr}\ (\mathsf{refl}\ u) \equiv \mathsf{refl}\ (\mathsf{unrepr}\ u) \\
&\mathsf{repr\text{-}J} : \mathsf{repr}\ (\mathsf{J}\ C\ w\ e) \\
&\qquad\qquad \equiv \mathsf{J}\ (\mathsf{Repr}\ C)\ (\mathsf{repr}\ w)\ e \\
&\mathsf{unrepr\text{-}J} : \mathsf{unrepr}\ (\mathsf{J}\ (\mathsf{Repr}\ C)\ w\ e) \\
&\qquad\qquad \equiv \mathsf{J}\ C\ (\mathsf{unrepr}\ w)\ e
\end{aligned}$$

Fig. 1. Coherence of the representation operators with substitutions, Π, Id, universes and codes. 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 are represented in the signature. In the following rules, $r : \mathsf{data\text{-}rep}\ (R, r, Q) \in_i \Sigma$. Firstly, we define the reduction that occurs when a type $\mathsf{D}\ i$ is represented,

$$\mathsf{Repr\text{-}D} : \forall r.\ \mathsf{Repr}\ (\mathsf{D}\ i) = \mathsf{El}\ R@\,, \tag{7}$$

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

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

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

Here, the operation $\_^{\mathsf{Repr}}$ is used to apply the term former repr to the recursive part of the arguments a. This is definable because $\mathsf{Args}\ T\ X$ is natural in X as it is a sum of products. The full construction can be found in section 6.2 of the appendix

One might be tempted to make this equality definitional too. Unfortunately, this would render conversion checking undecidable, because if one applies 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 (7) 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:

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

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

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

Above we use more auxilliary definitions which 'represent' the carriers of algebras, as well as displayed algebras:

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

These are defined straightforwardly; we represent outputs in a positive position using repr and unrepresent inputs in a negative position using unrepr. The full definitions can be found in section 6.2 of the appendix.

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 $\mathsf{def\text{-}repr}\ t\ p$, when accounting for the definitional equality between a definition and its implementation.

3.5 Normalisation by evaluation for λ_{REP}

To demonstrate that definitional equality is decidable in λ_{REP} with the addition of the representation operators, we can define a syntax of values, and then show that every term uniquely corresponds to a value.

Lemma 2. *The term formers* repr *and* unrepr *are injective, i.e.* $\mathsf{repr}\, x = \mathsf{repr}\, y \to x = y$ *and* $\mathsf{unrepr}\ x = \mathsf{unrepr}\ y \to x = y$.

Proof. *By applying* $\mathsf{unrepr}/\mathsf{repr}$ *to both sides followed by the rule* $\mathsf{reprr}/\mathsf{reprl}$.

4 Translating from λ_{REP} to λ_{IND}

We can 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. Doing so results in undecidable type checking, but this is not a problem because type checking is decidable λ_{REP} and we only apply this transformation after typechecking (when we already know that everything is well-typed).

This translation is defined over the syntax of λ_{REP} [3] 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 signatures $\mathcal{R} : \mathsf{Sig}_{\text{REP}} \to \mathsf{Sig}_{\text{IND}}^{\text{EXT}}$ 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

$$\begin{aligned}
&\mathcal{R} : \mathsf{Ty}_{\text{REP}}\ (\Sigma, \Delta)\ \to \mathsf{Ty}_{\text{IND}}^{\text{EXT}}\ (\mathcal{R}\Sigma, \mathcal{R}\Delta) \\
&\mathcal{R}\ (\mathsf{D}\ i) := \begin{cases} \mathcal{R}\ (\mathsf{El}\ R@) & \text{if } \mathsf{data\text{-}rep}\ (R, r, Q) \in_i \Sigma \\ \mathsf{D}\ \mathcal{R}i & \text{otherwise} \end{cases} \\
&\mathcal{R}\ (\mathsf{Repr}\ T) := \mathcal{R}T \\
&\text{(otherwise recurse on all subterms with } \mathcal{R})
\end{aligned}$$

This mirrors the definitional equality rules of Repr, but $\mathcal{R}$ is applied to all subterms. Similarly, terms are translated as

$$
\begin{aligned}
&\mathcal{R} : \mathsf{Tm}_{\text{REP}}\ (\Sigma, \Delta)\ T \to \mathsf{Tm}^{\text{EXT}}_{\text{IND}}\ (\mathcal{R}\Sigma, \mathcal{R}\Delta)\ \mathcal{R}T \\
&\mathcal{R}\ (\mathsf{C}_i\ a) = \begin{cases} \ulcorner \mathcal{R}r \urcorner\ \mathcal{R}a & \text{if } \mathsf{data\text{-}rep}\ (R, r, Q) \in_i \Sigma \\ \mathsf{C}_{\mathcal{R}i}\ \mathcal{R}a & \text{otherwise} \end{cases} \\
&\mathcal{R}\ (\mathsf{E}_i\ m) = \begin{cases} \ulcorner \mathcal{R}Q \urcorner_0\ \mathcal{R}m & \text{if } \mathsf{data\text{-}rep}\ (R, r, Q) \in_i \Sigma \\ \mathsf{E}_{\mathcal{R}i}\ \mathcal{R}m & \text{otherwise} \end{cases} \\
&\mathcal{R}\ (\mathsf{F}_i) = \begin{cases} \mathcal{R}t & \text{if } \mathsf{def\text{-}rep}\ (t, p) \in_i \Sigma \\ \mathsf{F}_{\mathcal{R}i} & \text{otherwise} \end{cases} \\
&\mathcal{R}\ (\mathsf{repr}\ t) := \mathcal{R}t \\
&\mathcal{R}\ (\mathsf{unrepr}\ t) := \mathcal{R}t \\
&\mathcal{R}\ (\mathsf{repr\text{-}C}_i\ a) := \mathsf{refl} \\
&\mathcal{R}\ (\mathsf{repr\text{-}motive\text{-}E}_i\ m) := \mathsf{refl} \\
&\mathcal{R}\ (\mathsf{unrepr\text{-}motive\text{-}E}_i\ m) := \mathsf{refl} \\
&\mathcal{R}\ (\mathsf{repr\text{-}methods\text{-}E}_i\ m) := \mathsf{refl} \\
&(\text{otherwise recurse on all subterms with } \mathcal{R})
\end{aligned}
$$

Constructor, eliminator and definition translations mirror the equality rules in section 3.4, 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 type 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 F and eliminators E are preserved by reflecting the propositional coherence rules provided by their defined representations:

$$
\begin{aligned}
&\mathsf{ap}_{\mathcal{R}}\ (\mathsf{E\text{-}id}_i\ m) := \begin{cases} \mathsf{reflect}\ \ulcorner \mathcal{R}Q \urcorner_1\ \mathcal{R}m & \text{if } \mathsf{data\text{-}rep}\ (R, r)\ Q \in_i \Sigma \\ \mathsf{E\text{-}id}_{\mathcal{R}i}\ \mathcal{R}m & \text{otherwise} \end{cases} \\
&\mathsf{ap}_{\mathcal{R}}\ (\mathsf{F\text{-}id}_i) := \begin{cases} \mathsf{reflect}\ \mathcal{R}p & \text{if } \mathsf{def\text{-}rep}\ (t, p) \in_i \Sigma \\ \mathsf{F\text{-}id}_{\mathcal{R}i} & \text{otherwise} \end{cases} \\
&(\text{otherwise recurse on all equality constructors with } \mathsf{ap}_{\mathcal{R}})
\end{aligned}
$$

Thus, by construction, $\mathcal{R}$ is sound with respect to typing and definitional equality.

Theorem 1. *$\mathcal{R}$ preserves typing and definitional equality.*

Proof. By construction, since it is defined on well-typed syntax quotiented by equality.

Theorem 2. *$\mathcal{R}$ 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 signatures in λ_{REP} without the $\trianglerighteq$ constructor. Thus no items have defined representations. Furthermore, 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.

4.1 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 λ, 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. 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. *An function $f : \mathsf{Tm}_{\text{REP}}\ \Gamma\ (\Pi\ A\ B)$, is* computationally irrelevant *if $|\mathcal{R}A| = |\mathcal{R}B|$ and $|\mathcal{R}f| = \mathtt{id}$.*

Lemma 3. *The type former Repr is injective up to internal isomorphism, i.e.*

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

Moreover, this isomorphism is computationally irrelevant.

Proof. The forward direction is given by first applying repr to t, transporting over the given equality and then applying unrepr. The backward direction is given by applying repr to t', transporting over the equality and then applying unrepr. The coherence holds by the rules reprr and reprl. After applying $\mathcal{R}$, all representation operators are erased and the isomorphism is the identity on both sides (even before extraction).

Consider extending the languages λ_{REP} and λ_{REP} with subset Σ-types

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

in such a way that Repr and $\mathcal{R}$ preserve them, but such that the extraction step erases the right component, i.e. $|\{A \mid B\}| = |A|$, $|(x, y)| = |x|$ and $|\pi_1 x| = |x|$. Then if we have an inductive family $G : \mathsf{data}\ (\bullet \rhd I)\ T_G \in \Sigma$ 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,

$$\mathsf{data\text{-}rep}\ (i.\ \{x : \mathsf{D}\ F \mid \mathsf{Id}\ (\mathsf{app}\ f\ x)\ i\}, r, Q) \in_G \Sigma$$

then we can construct computationally irrelevant forgetful and recomputation functions

$$\mathsf{forget}_i : \mathsf{Tm}\ \Gamma\ (\Pi\ (\mathsf{D}\ G)[i]\ (\mathsf{D}\ F))$$
$$\mathsf{forget}_i = g.\ \pi_1\ (\mathsf{repr}\ g)$$

$$\mathsf{remember} : \mathsf{Tm}\ \Gamma\ (\Pi\ (x : \mathsf{D}\ F)\ (\mathsf{D}\ G[\mathsf{app}\ f\ x]))$$
$$\mathsf{remember} = x.\ (\mathsf{unrepr}\ x, \mathsf{refl})$$

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

5 Implementation

Superfluid is a programming language with dependent types, $\mathcal{U} : \mathcal{U}$, quantities, inductive families and dependent pattern matching. Its compiler is written in Haskell and the compilation target is JavaScript. Dependent pattern matching in Superfluid is elaborated to a core language with internal eliminators. The $\mathcal{R}$ transformation is then applied to the core program, which erases all inductive constructs with defined representations. This is finally translated to JavaScript, erasing all irrelevant data. As a result, 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.

References

1. 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 (Jan 2016), https://doi.org/10.1145/2837614.2837638
2. 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 (Jul 2018), https://doi.org/10.1145/3209108.3209189
3. 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 (Jan 2017), https://doi.org/10.1145/3018610.3018620
4. 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
5. Cockx, J., Abel, A.: Elaborating dependent (co)pattern matching. Proc. ACM Program. Lang. **2**(ICFP), 1–30 (Jul 2018), https://doi.org/10.1145/3236770
6. 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 (Jan 2021), https://doi.org/10.1145/3434341

7. Dagand, P.E., McBride, C.: A categorical treatment of ornaments. arXiv [cs.PL] (Dec 2012), http://arxiv.org/abs/1212.3806
8. Diehl, L., Firsov, D., Stump, A.: Generic zero-cost reuse for dependent types. Proc. ACM Program. Lang. **2**(ICFP), 1–30 (Jul 2018), https://doi.org/10.1145/3236799
9. Kovács, A.: Type-theoretic signatures for algebraic theories and inductive types. Ph.D. thesis (2023), https://andraskovacs.github.io/pdfs/phdthesis_compact.pdf
10. 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
11. 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
12. 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 (Oct 1987), https://doi.org/10.1145/41625.41653
13. Wadler, P.: Deforestation: transforming programs to eliminate trees. Theor. Comput. Sci. **73**(2), 231–248 (Jun 1990), https://www.sciencedirect.com/science/article/pii/030439759090147A

6 Appendix

6.1 Utilities when working with algebras

6.2 Utilities when working with representations