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, can lead to unsatisfactory runtime performance. In this paper, we introduce a language with dependent types, and inductive families with customisable representations. Representations are a version of Wadler's views [25], refined to inductive families like in Epigram [23], 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 during compilation. We show how we can express optimisation techniques such as representing Nat-like types as GMP-style [28] big integers, without special casing in the compiler. With dependent types, reasoning about data representations is also possible through a provided modality. This yields 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 programming languages with dependent types. An inductive definition is equipped with an eliminator that and enables structural recursion over the data, and captures the notion of mathematical induction. This is a powerful tool for programming as well as theorem proving. However, this abstraction has a cost when it comes to compilation: the standard runtime representation of inductive types is a linked tree structure. This representation is not always the most efficient and often forces users to rely on machine primitives to achieve desirable performance, at the cost of structural induction and dependent pattern matching.

Despite advances in the erasure of irrelevant indices in inductive families [12] and the use of theories with irrelevant fragments [8,24], there is still a need to convert between differently-indexed versions of the same data. For example, consider the function that converts from BinTreeOfHeight T n to BinTree T by

forgetting the height index n. This 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.

Wadler's views [25] 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 [23] that use the index refinement machinery of dependent pattern matching to avoid certain proof obligations with eliminator-like constructs. While Wadler's views exhibit a nice way to transport across a bijection between the original data and the viewed data, they do not *erase* the view from the final program.

In this paper, we propose an extension DATATT to Martin-Löf type theory [21] with which allows programmers to define inductive types with 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 custom *representations* (section 2).
- A dependent type system DATATT with data types formulated in terms of inductive algebras for signatures, along with a translation to MLTT that replaces all data types with their defined inductive algebras (section 3). We have formalised this in Agda (section 8).
- A prototype implementation of this system in SUPERFLUID, a programming language with inductive families (section 4).

2 A tour of data representations

A common optimisation done by programming languages with dependent types such as Idris 2 [29], Agda [27], Rocq [31] and Lean [30] is to represent natural numbers more efficiently. The definition of natural numbers is

$$\textbf{data}\ \mathsf{Nat}\ \left\{\begin{array}{l}\mathsf{zero} : \mathsf{Nat}\\ \mathsf{succ} : \mathsf{Nat} \to \mathsf{Nat}\end{array}\right\} \tag{1}$$

and generates a case analysis principle $\mathsf{case}_{\mathsf{Nat}}$ of type

$$(P : \mathsf{Nat} \to \mathcal{U}) \to P\ \mathsf{zero} \to ((n : \mathsf{Nat}) \to P\ (\mathsf{succ}\ n)) \to (s : \mathsf{Nat}) \to P\ s\,,$$

which powers pattern matching. This is a special case of the induction principle $\mathsf{elim}_{\mathsf{Nat}}$, where the inductive hypotheses are given in each method. Without further intervention, Nat is represented in unary, where each digit becomes a heap

cell at runtime. This is inefficient for many basic operations on natural numbers, especially since computers are 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 [28] integers. This is done with the replacements

$$
\begin{aligned}
&|\mathsf{zero}| = \mathtt{0},\\
&|\mathsf{succ}| = \mathtt{\backslash x => x + 1},\\
&|\mathsf{case}_{\mathsf{Nat}}\ P\ m_{\mathsf{zero}}\ m_{\mathsf{succ}}\ s| \\
&\qquad = \mathtt{let\ s} = |s|\ \mathtt{in\ if\ s == 0}\\
&\qquad\quad \mathtt{then}\ |m_{\mathsf{zero}}|\\
&\qquad\quad \mathtt{else}\ |m_{\mathsf{succ}}|\ \mathtt{(s - 1)},
\end{aligned}
$$

where $|\cdot|$ denotes a source translation into a compilation target language with appropriate big unsigned integer primitives. This is the 'Nat-hack'.[1]

In addition to the constructors and case analysis, the compiler might define translations for commonly used definitions which have a more efficient counterpart in the target, such as addition, multiplication, etc. The recursively-defined functions are well-suited to proofs and reasoning, while the primitives are more efficient for computation. This way, the surface program can take advantage of the structural properties of induction, while still benefiting from an efficient representation at runtime.

Unfortunately, this approach only works for the data types which the compiler recognises 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. It is also hard to know when such optimisations will fire and performance can become brittle upon refactoring. Hence, one of our goals is to extend this optimisation to work for any data type. To achieve 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

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

[1] Idris 2 will in fact look for any Nat-like types and apply this optimisation. 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.

The inductive type Nat is represented as the type UBig of big unsigned integers, with translations for the constructors zero and succ, and the eliminator $\mathsf{elim}_{\mathsf{Nat}}$ (now with the inductive hypotheses). Additionally, the eliminator must satisfy the computation rules of the Nat eliminator, which are postulated as propositional equalities. This representation is valid in a context containing the symbols

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

and propositional equalities

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

For the remainder of the paper we will work with eliminators rather than case analysis but the approach can be specialised to the latter if the language has general recursion. Assuming call-by-value semantics, inductive hypotheses are labelled $\overline{P}$ which denotes lazy values, that is, functions $\mathsf{Unit} \to P$.

The compiler knows how to perform pattern matching on Nat, and produces invocations of $\mathsf{elim}_{\mathsf{Nat}}$ as a result. On the other hand, it does not know how to pattern match on UBig. With this representation, we get the best of both worlds: Nat is used for pattern matching, but is then replaced with UBig during compilation, generating more efficient code. We expect that the underlying implementation of UBig indeed satisfies these postulated properties, which is a separate concern from the correctness of the representation itself. However, such 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 as a refinement of 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

$$\mathsf{Vec}\ T\ n \simeq \{l : \mathsf{List}\ T \mid \mathsf{length}\ l = n\}$$

which we shorthand as $\mathsf{List'}\ T\ n := \{l : \mathsf{List}\ T \mid \mathsf{length}\ l = n\}$. 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. We also assume that Vec's n index is computationally irrelevant. We can thus choose $\mathsf{List'}\ T\ n$ as the representation

of Vec T n. We are then tasked with providing terms that correspond to the constructors of Vec but for List'. These can be defined as

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

Next we need to define the eliminator for List', which should have the form

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

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 cong (succ) (succ-inj p) = p in the recursive case.

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

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

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

The first holds definitionally, and the second requires a small amount of equality reasoning. This completes the definition of the representation of Vec as List', which would be written as

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

Now the hard work is done. Every time we are working with a v : 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 [8]. In section 3.7 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. We can do this using the repr and unrepr operators of the language (defined in section 3). These allow us to convert between an inductive type and its representation. Specifically, we can define the functions

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

$$\begin{aligned}&\mathsf{remember\text{-}length} : (l : \mathsf{List}\ T) \to \mathsf{Vec}\ T\ (\mathsf{length}\ l)\\&\mathsf{remember\text{-}length}\ l = \mathsf{unrepr}\ (l, \mathsf{refl})\,.\end{aligned}$$

In section 3.7 we will show that such functions are inverses of one another and are computationally irrelevant. These operators are typed as

$$\mathsf{repr} : A \to \mathsf{Repr}\ A \qquad \mathsf{unrepr} : \mathsf{Repr}\ A \to A$$

where Repr A computes to the defined representation of A, if A is a data type. Repr is a kind of 'intensional' modality, with the property that Repr $A \simeq A$. It allows us to transport across equivalences introduced by representations in a computationally-irrelevant manner.

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

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

for some index family $X : P \to \mathcal{U}$. If we hope to represent G as some refinement of F then we must 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 [15] 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 as well as the recomputation map from F to G are erased, constructed with repr and unrepr. We formulate this pattern in the general setting in section 3.7.

2.4 Zero-copy deserialisation

The machinery of representations can be used to implement zero-copy deserialisation of data formats into inductive types. Here we sketch how this could work. Consider the following record for a player in a game:

```
data Player { player : (position : Position)
            → (direction : Direction)
            → (items : Fin MAX_INVENTORY)
            → (inventory : Inventory (fin-to-nat items)) → Player }
```

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:

```
data Inventory (n : Nat) { empty : Inventory zero
                         ; add : Item → Inventory n → Inventory (succ n) }
```

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 inventory : $(p$: Player$)$ → Inventory p.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

```
repr Player as {Buf | IsPlayer} { player as buf-is-player
                                ; elim_Player as elim-buf-is-player
                                  by elim-buf-is-player-id }
```

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 [5] 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 eventual representation of Vec at runtime is determined by 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 Repr (Vec T n) $\equiv$ List T, which means the Repr modality only considers the immediate defined representation of a term. 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:

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

Every such type comes with a projection function to the Buf type

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

which eventually computes to the identity function after applying repr the appropriate amount of times. Upon compilation, every type is converted to its eventual representation, and all repr calls are erased, so the as-buf function becomes the identity function at runtime, given that the r argument is known at compile-time and monomorphised.

3 A type system for data representations

In this section, we 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 inductive algebras for signatures. In section 3.5 we define a core language DATATT with these features. The base theory is intensional Martin-Löf type theory (MLTT) [21] with a single universe $\mathcal{U} : \mathcal{U}$. We omit considerations of consistency and universe hierarchy, though these can be added if needed. In section 3.5, we define the modality Repr that allows us to convert between inductive types and their representations. Finally, in section 3.6 we define a translation from DATATT to *extensional* MLTT, which 'elaborates away' all inductive families to their representations. All of the examples in the paper so far have been written in a surface language that elaborates to DATATT.

The languages we work with are defined in an intrinsically well-formed manner [7] as a setoid over definitional equality, with de-Bruijn indices for variables. Weakening of terms is generally left implicit to reduce syntactic noise, and often named notation is used when indices are implied. We use $(a : A) \to B$ for dependent functions, $(a : A) \times B$ for dependent pairs, $a \equiv_A a'$ 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 $\Gamma \vdash B[a]$. We also notationally

identify elements of the universe $A : \mathcal{U}$ with types A type. Besides the usual judgement forms of MLTT, we also have telescopic judgement forms

$\Gamma \vdash \Delta\ \mathsf{tel}$ $\quad$ Δ is a telescope in Γ

$\Gamma \vdash \delta :: \Delta$ $\quad$ δ is a spine (list of terms) matching telescope Δ

with accompanying rules shown in fig. 1.

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

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

Fig. 1. Rules for forming telescopes and spines.

Extending contexts by telescopes (such as Γ, Δ) is defined by induction on telescopes. 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 [3,20]. 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 the ones which can be used as a syntax for defining inductive families in a type theory. Thus, we define two new judgement forms

$\Gamma \vdash S\ \mathsf{sig}\ \Delta$ $\quad$ S is a signature with indices Δ in context Γ

$\Gamma \vdash O\ \mathsf{op}\ \Delta$ $\quad$ O is an operation with indices Δ in context Γ

with accompanying rules shown in fig. 1. Signatures are lists of operations, and operations build up constructor types.

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

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

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:

- $(x : A) \to_{\mathsf{ext}} O[x]$, a (dependent) abstraction over some external type A, 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 $\Gamma \vdash (\iota\ (\,) \lhd \iota\ (\,) \to_{\mathsf{int}} \iota\ (\,) \lhd \epsilon)\ \mathsf{sig}\ \bullet$. We can add labels to aid readability, omit index spines if they are empty, and omit the final ϵ from signatures:

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

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

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

Later (section 3.4) 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 [20]. 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 we are working with.

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

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

Induction The actual type of natural numbers $\Gamma \vdash \mathbb{N} \;\mathsf{type}$ is 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

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

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 with indices 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. 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.

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 coherence 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:

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

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

An empty signature ϵ produces an empty telescope, while an extended signature $O \lhd S'$ 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:

$$\boxed{\Gamma \vdash O^{\mathsf{in}}\ X\ \mathsf{tel}} \qquad \boxed{\Gamma \vdash \{O\}\ \nu^{\mathsf{out}} :: \Delta}$$

$$(A \to_{\mathsf{ext}} O')^{\mathsf{in}}\ X = (a : A,\ O'[a]^{\mathsf{in}}\ X) \qquad \{O = A \to_{\mathsf{ext}} O'\}\ (a, \nu')^{\mathsf{out}} = \nu'^{\mathsf{out}}$$

$$(\iota\ \delta \to_{\mathsf{int}} O')^{\mathsf{in}}\ X = (x : X[\delta],\ O'^{\mathsf{in}}\ X) \qquad \{O = \iota\ \delta \to_{\mathsf{int}} O'\}\ (x,\ \nu')^{\mathsf{out}} = \nu'^{\mathsf{out}}$$

$$(\iota\ \delta)^{\mathsf{in}}\ X = \bullet \qquad \{O = \iota\ \delta\}\ (\,)^{\mathsf{out}} = \delta\,.$$

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 $-^{\mathsf{dispAlg}}$:

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

$$\{S = \epsilon\}\ (\,)^{\mathsf{dispAlg}}\ Y = \bullet$$

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

We omit the definitions of $-^{\mathsf{dispIn}}$ and $-^{\mathsf{dispOut}}$ and refer to the Agda formalisation, 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

$$\boxed{\Gamma \vdash \{S\}\ \{\alpha\}\ \beta^{\mathsf{coh}}\ \sigma\ \mathsf{tel}}$$

$$\{S = \epsilon\}\ \{\alpha = (\,)\}\ (\,)^{\mathsf{coh}}\ \sigma = \bullet$$

$$\{S = O \lhd S'\}\ \{\alpha = (\alpha_O, \alpha')\}\ (\beta_O, \beta')^{\mathsf{coh}}\ \sigma$$
$$= ((\nu :: O^{\mathsf{in}}\ X) \to \sigma[\alpha_O\ \nu] \equiv \beta_O\ (\sigma\ \$\ \nu),\ \beta'^{\mathsf{coh}}\ \sigma)\,.$$

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.

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

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

where Y is the motive, β are the methods, and σ is the output section which must satisfy the propositional coherence conditions ρ. 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 [1], where all we need to provide is a signature.[2] Here, we must also implement the signature and prove the induction principle by providing γ, rather than it being 'built-in' to the type theory. For example, if the type theory has W-types, then we can construct γ using an appropriate W-type. In effect, this will later allow us to translate away inductive definitions to their defined representations. This leads to the formal definition of a representation:

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

In fig. 3 we define the data type, and its corresponding introduction, elimination, and computation rules. 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 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

[2] Notice that the data defining a W-type $(A : \mathcal{U}, B : A \to \mathcal{U})$ can be viewed as a kind of signature, where A describes the operations and their non-recursive parameters, while B describes the arities of the recursive parameters.

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

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

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

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.

principle suggests that the constructors corresponding to each method are disjoint. Since constructors ctor are primitive terms 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 [22] 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* (example 1), 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 $\mathbb{N} = \mathsf{data}\ \mathit{NatSig}\ (N, \mathit{zero}_N, \mathit{succ}_N, \mathit{elim}_N)$ because the syntax does not equate ctor_i and ctor_j unless $i = j$.

Importantly, 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 $\mathsf{elim}_{\mathbb{N}}$, which has the expected computation rules as shown in fig. 3. Converting dependent pattern matching to eliminators has been explored in depth by Goguen, McBride and McKinna [17], as well as by Cockx and Devriese [14] in absence of Axiom K.

3.5 Reasoning about 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 that is the carrier of two inductive algebras (X, α, κ) and (X, α', κ') for 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. It comes with two term formers repr and unrepr, which are definitional inverses of each other. We highlight the main rules of Repr in fig. 4.

$$\textsc{Repr-Form}\ \frac{\Gamma \vdash A\ \mathsf{type}}{\Gamma \vdash \mathsf{Repr}\ A\ \mathsf{type}} \qquad \textsc{Repr-Intro}\ \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{repr}\ a : \mathsf{Repr}\ A} \qquad \textsc{Repr-Elim}\ \frac{\Gamma \vdash a : \mathsf{Repr}\ A}{\Gamma \vdash \mathsf{unrepr}\ a : A}$$

$$\textsc{Repr-Id}_1\ \frac{\Gamma \vdash a : \mathsf{Repr}\ A}{\Gamma \vdash \mathsf{repr}\ (\mathsf{unrepr}\ a) = a : \mathsf{Repr}\ A} \qquad \textsc{Repr-Id}_2\ \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{unrepr}\ (\mathsf{repr}\ a) = a : A}$$

$$\textsc{Repr-Data}\ \frac{\Gamma \vdash S\ \mathsf{sig}\ \Delta \qquad \Gamma \vdash \gamma :: S^{\mathsf{indAlg}} \qquad \Gamma \vdash \delta :: \Delta}{\Gamma \vdash \mathsf{Repr}\ (\mathsf{data}_\Delta\ S\ \gamma\ \delta) = \gamma.X\ \delta}$$

Fig. 4. Introduction and elimination forms, as well as computation rules for the Repr modality.

These rules allow us to go between a data type $D = \mathsf{data}_\Delta\ S\ \gamma\ \delta$ and its representation $\gamma.X\ \delta$. In the translation to extensional MLTT that we are yet to define, this modality is also translated away. Indeed, its purpose is purely intensional: we do not want to equate $\mathsf{data}_\Delta\ S\ \gamma$ with $\gamma.X$ because that would render conversion checking undecidable, but we still want to make use of the fact that these types are 'the same'. In other words, $\mathsf{Repr}\ A \simeq A$ but not $\mathsf{Repr}\ A = A$. All the contextual machinery of general modal type systems [18] is not necessary here because this modality is fibred over contexts so it presents as a type former.

One might hope for additional computation rules. For example, the representation of a constructor should be equal to the underlying algebra element of

the constructor type's representation:[3]

$$\mathsf{repr}\ (\mathsf{ctor}_{S.O}\ \nu) = \gamma.\alpha_O\ (\mathsf{repr}\ \nu)\,.$$

Unfortunately, having this as a computation rule would render conversion checking undecidable, because if one applies unrepr to a term $\mathsf{repr}\ (\mathsf{ctor}_{S.O}\ \nu)$ which has already been reduced to its representation, $\mathsf{unrepr}\ (\gamma.\alpha_O\ (\mathsf{repr}\ \nu))$, there is no clear way to decide that this is convertible to $\mathsf{ctor}_{S.O}\ \nu$ even though the definitional equality rules would imply that it is (due to Repr-Id$_2$). Nevertheless, we can still postulate this equality propositionally, and it is justified by the translation step.

We can also ask for compatibility equations between Repr and the rest of the type formers of MLTT; for example $\mathsf{Repr}\ ((a : A) \times B[a]) = (a : \mathsf{Repr}\ A) \times \mathsf{Repr}\ B[\mathsf{unrepr}\ a]$, which can hold definitionally without breaking conversion. These are given for $\mathcal{U}$, Π, Σ, unit and identity types in the Agda formalisation.

Subuniverse of concrete types As described, the modality Repr is not idempotent: $\mathsf{Repr}\ (\mathsf{Repr}\ A) = \mathsf{Repr}\ A$ does not always hold. An inductive algebra used as a representation of a data type might itself be implemented in terms of another data type, which again reduces under the action of Repr. A more principled approach might be to view the image of Repr as a subuniverse of $\mathcal{U}$. The restriction

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

targets a universe of 'concrete' types $\mathcal{U}_C < \mathcal{U}$ closed under all standard type formers, but without any data types. We can then require that all inductive algebras γ used in the rule Data-Form must have a concrete carrier $X : \Delta \to \mathcal{U}_C$. This does not limit expressivity because we can always wrap any inductive algebra carrier with Repr to bring it down to $\mathcal{U}_C$. We do not assume this additional structure for simplicity, but it might be a useful feature in practice. For example, it would simplify the transitivity example in section 2.5.

3.6 Translating to extensional MLTT

We now define a type- and equality-preserving translation step $\mathcal{R}$ shown in fig. 5 from DATATT to extensional MLTT, to be applied during the compilation process. The extensional flavour of MLTT involves adding the equality reflection rule

$$\frac{\begin{array}{l}\textsc{Reflect}\ p\\ \Gamma \vdash p : a =_A a'\end{array}}{\Gamma \vdash a \equiv a' : A}\,.$$

General undecidability of conversion is not a problem because type checking is decidable for DATATT[4] and we apply this transformation after type checking, on

[3] When we write $\mathsf{repr}\ \nu$ we mean to apply repr to all recursive occurences (all places that ι appears in the domain of O).

[4] Not formalised in this paper.

fully-typed terms. The translation is defined over the syntax of DATATT [11] such that definitional equality is preserved, shown in fig. 5. $\mathcal{R}$ replaces data types with their underlying inductive algebras. We use the notation Ty and Tm for types and terms respectively, with a subscript indicating the language.

$$\boxed{\mathcal{R} : \mathsf{Ty}_{\text{DATATT}}\ \Gamma \to \mathsf{Ty}_{\text{MLTT}}\ \mathcal{R}\Gamma}$$

$$\mathcal{R}(\mathsf{Repr}\ A) = \mathcal{R}A$$
$$\mathcal{R}(\mathsf{data}_{\Delta}\ S\ \gamma\ \delta) = \mathcal{R}(\gamma.X)\ \mathcal{R}\delta$$
(otherwise recurse with $\mathcal{R}$)

$$\boxed{\mathcal{R} : \mathsf{Tm}_{\text{DATATT}}\ \Gamma\ A \to \mathsf{Tm}_{\text{MLTT}}\ \mathcal{R}\Gamma\ \mathcal{R}A}$$

$$\mathcal{R}(\mathsf{repr}\ t) = \mathcal{R}t$$
$$\mathcal{R}(\mathsf{unrepr}\ t) = \mathcal{R}t$$
$$\mathcal{R}(\mathsf{ctor}_{S.O}\ \{\gamma\}\ \nu) = \mathcal{R}(\gamma.\alpha_O)\ \mathcal{R}\nu$$
$$\mathcal{R}(\mathsf{elim}_S\ \{\gamma\}\ M\ \beta\ \delta\ x) = (\mathcal{R}(\gamma.\kappa)\ \mathcal{R}M\ \mathcal{R}\beta).\sigma\ \mathcal{R}\delta\ \mathcal{R}x$$
(otherwise recurse with $\mathcal{R}$)

Fig. 5. Translation of DATATT to MLTT, replaces data types with their underlying inductive algebras, and eliminators by the induction principle provided by representations.

All the mappings above are structurally recursive, demonstrated by the construction of a *model* of DATATT in the Agda formalisation. The translation is extended to contexts, substitutions, telescopes and spines pointwise, and the rest of the syntax is preserved: $\mathcal{R}((a : A) \to B[a]) = (a : \mathcal{R}A) \to (\mathcal{R}B)[a]$ etc. All data types are translated to the carriers of their inductive algebras. Invocations of Repr are removed. One can view Repr as locally applying $\mathcal{R}$ to a part of the program. The definitional equality preservation by $\mathcal{R}$ is shown in fig. 6. The isomorphism of $\mathsf{repr}/\mathsf{unrepr}$, as well as the rule REPR-DATA are preserved by metatheoretic reflexivity on the other side, since all representation operators are erased. Coherence rules for eliminators are preserved by reflecting the propositional coherence rules provided by their defined representations.

$$\boxed{\mathcal{R}^{\mathsf{Ty}\ \Gamma}_{\approx} : A \approx_{\text{DATATT}} A' \to \mathcal{R}A \approx_{\text{MLTT}} \mathcal{R}A'}$$

$$\mathcal{R}(\textsc{Repr-Data}\ \{S, \gamma, \Delta\}) = \textsc{Refl}\ (\mathcal{R}(\gamma.X)\ \mathcal{R}\delta)$$
(otherwise recurse with $\mathcal{R}_{\approx}$)

$$\boxed{\mathcal{R}^{\mathsf{Tm}\ \Gamma\ A}_{\approx} : a \approx_{\text{DATATT}} a' \to \mathcal{R}a \approx_{\text{MLTT}} \mathcal{R}a'}$$

$$\mathcal{R}(\textsc{Repr-Id}_1\ \{a\}) = \textsc{Refl}\ \mathcal{R}a$$
$$\mathcal{R}(\textsc{Repr-Id}_2\ \{a\}) = \textsc{Refl}\ \mathcal{R}a$$
$$\mathcal{R}(\textsc{Data-Comp}\ \{S, O, \gamma, M, \beta, \nu\}) = \textsc{Reflect}\ (\mathcal{R}(\gamma.\kappa)\ \mathcal{R}M\ \mathcal{R}\beta).\rho_O\ \mathcal{R}\nu$$
(otherwise recurse with $\mathcal{R}_{\approx}$)

Fig. 6. Equality translation of DATATT to extensional MLTT. This amounts to an inductive proof that $\mathcal{R}$ preserves equality.

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

Theorem 2 (AGDA). *$\mathcal{R}$ is a left-inverse of the inclusion i : MLTT $\hookrightarrow$ DATATT:*

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathcal{R}(i(a)) = a : A} \quad \textit{(in extensional MLTT).}$$

Inductive types in an empty context Basic MLTT with dependent functions, pairs and propositional equality as we have presented here is not sufficient to construct most inductive algebras in an empty context. Without W-types or fixpoints, we must postulate the induction principles we have access to, like the example with GMP integers in section 2. In practice, we often want to be able to construct inductive types 'from scratch'. For this, we can extend the base theory with W-types or something similar. One convenient choice is to extend the syntax of both the source and target languages of $\mathcal{R}$ with a class of data types like in fig. 3, but without requiring them to be implemented by inductive algebras. The advantage is that now we can opt-in when we want to represent inductive types specially, but otherwise fall back to some kind of default implementation. The downside is that now the target still contains inductive types, though this is okay for compilation purposes if we choose a 'sane default', usually tagged unions containing indirections. This is the approach we take in SUPERFLUID.

3.7 Computational irrelevance

In a compiler, there will be an additional program extraction step from the target of $\mathcal{R}$ into some simply-typed or untyped language, to be handled by the code-generation backend. We call this language PROG, and the transformation by vertical bars $|x|$. As opposed to $\mathcal{R}$, it might not preserve the definitional equality of the syntax—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 PROG.

Definition 3. *A function $\Gamma \vdash f : (a : A) \to B$, is* computationally irrelevant *if $|\mathcal{R}A| = |\mathcal{R}B|$ and $|\mathcal{R}f| =$ `\x => x`.*

Theorem 3 (AGDA). *The type former Repr is injective up to equivalence, i.e.*

$$\frac{\Gamma \vdash p : \mathit{Repr}\ T =_{\mathcal{U}} \mathit{Repr}\ T'}{\Gamma \ \vdash \mathit{conv}_p : T \simeq T'}, \tag{2}$$

and if $|-|$ erases internal equality reasoning, conv_p is computationally irrelevant.

Proof. *For the input proof p, for conv_p we have $\lambda x.\ \mathit{unrepr}_{T'}\ (\mathit{coe}\ (\mathit{repr}\ x)\ p)$ and for conv_p^{-1} we have $\lambda x.\ \mathit{unrepr}_T\ (\mathit{coe}\ (\mathit{repr}\ x)\ (\mathit{sym}\ p))$. Both directions map to $\lambda x.\ \mathit{coe}\ \_\ x$ by $\mathcal{R}$ which becomes `\x => x` by $|-|$.*

In addition, we can reason about the computational irrelevance of refinements. Consider extending both source and target languages of $\mathcal{R}$ with usage-aware 'subset' dependent pairs

$$\frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma, A \vdash B\ \mathsf{type}}{\Gamma \vdash \{A \mid B\}\ \mathsf{type}}$$

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 $|\pi_1 x| = |x|$. This can be implemented using quantitative type theory for example. Suppose we have an inductive family $G = \mathsf{data}_I\ S_G\ \gamma_G$ over some index type I, and an inductive type $F = \mathsf{data}\ S_F\ \gamma_F$ such that G is represented by a refinement $r : F \to I$, meaning

$$\gamma_G = (\lambda i.\ \{f : F \mid r\ f \equiv_I i\}, \alpha, \kappa)\,.$$

Then, we can construct computationally irrelevant functions

$$\begin{array}{ll} \mathsf{forget}_i : G\ i \to F & \mathsf{remember} : (f : F) \to G\ (r\ f) \\ \mathsf{forget}_i = \lambda g.\ \pi_1\ (\mathsf{repr}\ g) & \mathsf{remember} = \lambda x.\ \mathsf{unrepr}\ (x, \mathsf{refl})\,. \end{array}$$

By reasoning similar to theorem 3, $|\mathcal{R}\ \mathsf{forget}_i| = |\mathcal{R}\ \mathsf{remember}| = \texttt{\textbackslash x => x}$.

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

Currently we do not require proofs of eliminator coherence, but they are straightforward to add. We also treat the rule for representing constructors ($\mathsf{repr}\ (\mathsf{ctor}_{S.O}\ \nu) = \gamma.\alpha_O\ (\mathsf{repr}\ \nu)$) as definitional in the implementation, at the cost of breaking decidability of equality, but with the benefit of fewer manual transports. Superfluid also supports the definition of representations for global functions in addition to inductive families. This is a simple symbol-replacement mechanism (like Idris's `%transform` pragma) so that we can still take advantage of inductively defined functions—such as addition on natural numbers—for theorem proving, but use the optimised primitive addition when generating code.

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 [22]. 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.

5 Related work

Using inductive types as a form of abstraction was first explored by Wadler [25] through *views*. The extension to dependent types was developed by McBride and McKinna [23], as part of the Epigram project. Our system differs from views in the computational content of the abstraction; even with deforestation [26] views are not always zero-cost, but representations are. Atkey [9] 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 [16], but does not extend to custom representations.

Work by Allais [4,5] 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 partial evaluation to erase their views, which does not always fire. On the topic of memory layout optimisation, Baudon [10] 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, including Idris 2 [35], Rocq [34] and Agda [32], 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 [36] to erase the forgetful and recomputation functions for reindexing list/maybe/number-like types. There is also demand for this kind of optimisation in Agda [33].

6 Future work

There are elements of our formalisation which should be developed further. We did not formulate normalisation and decidability of equality for DATATT, which is needed for typechecking. We have implemented a normalisation-by-evaluation [2] algorithm used in SUPERFLUID, but have only sketched that it has the desired properties. On the practical side, we have not explored examples of representations in great detail. Once the implementation of SUPERFLUID is more complete, we would like to explore more sophisticated and complete examples, along with compelling benchmarks. We would also like to describe SUPERFLUID's feature of representing global function definitions more formally in the future.

As a next step we aim to expand the class of theories we consider, in particular to include quotient-induction. Representations for quotient-inductive types

in 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. Additionally, quotient-inductive types could be a good candidate for improving typechecking ergonomics by deciding their equational theories, similar to Frex [6]. Such a system could apply certain representations during typechecking rather than code generation, to solve equations involving free variables through a normalisation-by-evaluation procedure.

We would like to further explore the relationship of the Repr modality with general systems for defining modalities, such as *multi-modal type theory* by Gratzer [18]. Additionally, we expect that metaprogramming with representations is most naturally done in the context of *two-level type theory* (2LTT) [19]. We would like to explore the embedding of DATATT in a two-level type theory, where signatures become types in the meta-fragment. Then we could develop various methods of generating representations internally, for example through algebraic ornaments, without needing to laboriously prove induction principles by hand. The translation step in section 3.6 can already be viewed as a kind of staging procedure, and could be integrated with the one of 2LTT.

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, type-driven correctness can be leveraged further without great sacrifices in performance.

Acknowledgements We thank the anonymous reviewers for their feedback that improved the quality of this paper, Guillaume Allais for helpful comments on the extended abstract, as well as Nathan Corbyn, Ellis Kesterton, and Matthew Pickering for interesting discussions surrounding it.

References

1. 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
2. Abel, A.: Normalization by evaluation: Dependent types and impredicativity. Ph.D. thesis, https://www2.tcs.ifi.lmu.de/~abel/talkHabil2013.pdf

3. 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
4. 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
5. Allais, G.: Seamless, correct, and generic programming over serialised data. arXiv [cs.PL] (20 Oct 2023), http://arxiv.org/abs/2310.13441
6. Allais, G., Brady, E., Corbyn, N., Kammar, O., Yallop, J.: Frex: dependently-typed algebraic simplification. arXiv.org (2023), http://dx.doi.org/10.48550/ARXIV.2306.15375
7. 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
8. 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
9. 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
10. 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
11. 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
12. 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
13. 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
14. 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-your-type-theory-div.pdf
15. Dagand, P.E., McBride, C.: A categorical treatment of ornaments. In: 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science. IEEE (Jun 2013), http://dx.doi.org/10.5555/2591370.2591396

16. 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
17. 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
18. Gratzer, D., Kavvos, G.A., Nuyts, A., Birkedal, L.: Multimodal dependent type theory. In: Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science. ACM, New York, NY, USA (8 Jul 2020), http://dx.doi.org/10.1145/3373718.3394736
19. Kovács, A.: Staged compilation with two-level type theory. Proc. ACM Program. Lang. **6**(ICFP), 540–569 (29 Aug 2022), https://dl.acm.org/doi/10.1145/3547641
20. Kovács, A.: Type-theoretic signatures for algebraic theories and inductive types. Ph.D. thesis (2023), https://andraskovacs.github.io/pdfs/phdthesis_compact.pdf
21. Martin-Löf, P.: Intuitionistic type theory **1**, 1–91 (1984), https://intuitionistic.wordpress.com/wp-content/uploads/2010/07/martin-lof-tt.pdf
22. 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
23. 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
24. 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
25. 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
26. 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
27. The Agda Wiki. https://wiki.portal.chalmers.se/agda/pmwiki.php, accessed: 2024-5-3
28. The GNU MP Bignum Library. https://gmplib.org/, accessed: 2024-12-8
29. Idris: A Language for Type-Driven Development. https://www.idris-lang.org/, accessed: 2024-5-3
30. Lean: Programming Language and Theorem Prover. https://lean-lang.org/, accessed: 2024-5-3
31. Welcome to a World of Rocq. https://rocq-prover.org/, accessed: 2025-4-16
32. agda2hs Documentation — agda2hs documentation. https://agda.github.io/agda2hs/, accessed: 2025-2-19
33. Should Agda optimise away the erasure from Vec to List?. Issue #7701. https://github.com/idris-lang/Idris2/pull/3486, accessed: 2025-2-19

34. Program extraction — Coq 8.13.2 documentation. https://coq.inria.fr/doc/v8.13/refman/addendum/extraction.html, accessed: 2025-2-19
35. Pragmas — Idris2 0.0 documentation. https://idris2.readthedocs.io/en/latest/reference/pragmas.html#transform, accessed: 2025-2-19
36. Make 'CONS', 'NIL', 'JUST' and 'NOTHING' constructors have uniform names by Z-snails. Pull Request #3486. https://github.com/idris-lang/Idris2/pull/3486, accessed: 2025-2-19

8 Appendix

Implementation The implementation of SUPERFLUID can be found at `https://github.com/kontheocharis/superfluid`.

Formalisation The Agda formalisation of the developments of this paper at `https://github.com/kontheocharis/rep-agda`.

8.1 Definition of MLTT

For reference, we define Martin-Löf type theory with $\mathcal{U} : \mathcal{U}$, Π, Σ, propositional equality and unit. We omit the definition of the substitution calculus and equality coercions (see [13, 5.1.2]). We use de-Brujin indices, keeping weakening implicit, and abuse notation for substitutions of terms: $A[t]$ for $A[\mathsf{id}, t]$.

$$\textsc{Univ-Form}\ \frac{}{\Gamma \vdash \mathcal{U}\ \mathsf{type}} \qquad \textsc{El-Form}\ \frac{\Gamma \vdash a : \mathcal{U}}{\Gamma \vdash \mathsf{El}\ a\ \mathsf{type}} \qquad \textsc{Univ-Intro}\ \frac{\Gamma \vdash A\ \mathsf{type}}{\Gamma \vdash \mathsf{code}\ A : \mathcal{U}}$$

$$\textsc{Pi-Form}\ \frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma, x : A \vdash B\ \mathsf{type}}{\Gamma \vdash (x : A) \to B\ \mathsf{type}}$$

$$\textsc{Pi-Intro}\ \frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda\ x.\ b : (x : A) \to B} \qquad \textsc{Pi-Elim}\ \frac{\Gamma \vdash f : (x : A) \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B[a]}$$

$$\textsc{Eq-Form}\ \frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash a \equiv_A b\ \mathsf{type}} \qquad \textsc{Eq-Intro}\ \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}\ a : a \equiv_A a}$$

$$\textsc{Eq-Elim}\ \frac{\begin{array}{c}\Gamma \vdash A\ \mathsf{type} \qquad \Gamma, a : A, b : A, a \equiv_A b \vdash P\ \mathsf{type} \\ \Gamma, a : A \vdash r : P[a, a, \mathsf{refl}\ a] \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash p : a \equiv_A b\end{array}}{\Gamma \vdash \mathsf{J}\ P\ d\ p : P[a, b, p]}$$

$$\textsc{Unit-Form}\ \frac{}{\Gamma \vdash \top\ \mathsf{type}} \qquad \textsc{Unit-Intro}\ \frac{}{\Gamma \vdash \mathsf{tt} : \top}$$

$$\textsc{Sigma-Form}\ \frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma, x : A \vdash B\ \mathsf{type}}{\Gamma \vdash (x : A) \times B\ \mathsf{type}}$$

$$\textsc{Sigma-Intro}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B[a]}{\Gamma \vdash (a, b) : (x : A) \times B} \qquad \textsc{Sigma-Elim-Fst}\ \frac{\Gamma \vdash p : (x : A) \times B}{\Gamma \vdash \mathsf{fst}\ p : A} \qquad \textsc{Sigma-Elim-Snd}\ \frac{\Gamma \vdash p : (x : A) \times B}{\Gamma \vdash \mathsf{snd}\ p : B[\mathsf{fst}\ p]}$$

Fig. 7. Typing rules for MLTT.

Universes

$\mathsf{El}\ (\mathsf{code}\ A) = A$

$\mathsf{code}\ (\mathsf{El}\ t) = t$

Equality and unit

$\mathsf{J}\ P\ r\ (\mathsf{refl}\ a) = r\ a$

$x = \mathsf{tt}$

Π types

$(\lambda\ x.\ b)\ a = b[a]$

$\lambda\ x.\ (f\ x) = f$

Σ types

$\mathsf{fst}\ (a, b) = a$

$\mathsf{snd}\ (a, b) = b$

$(\mathsf{fst}\ p, \mathsf{snd}\ p) = p$

Fig. 8. Definitional equality rules for MLTT, omitting substitution rules such as $(\mathsf{El}\ a)[\sigma] = \mathsf{El}\ (a[\sigma])$.

8.2 Definition of DATATT

The language DATATT is the extension of MLTT. by the rules in figs. 3 and 4. Below we present some additional definitional equality rules of Repr that make it commute with most of the syntax, as well as some propositional equalities that are justified by the translation $\mathcal{R}$.

η rules

$\mathsf{unrepr}\ (\mathsf{repr}\ t) = t$

$\mathsf{repr}\ (\mathsf{unrepr}\ t) = t$

Stability under substitution

$\mathsf{repr}\ (t[\sigma]) = (\mathsf{repr}\ t)[\sigma]$

$\mathsf{unrepr}\ (t[\sigma]) = (\mathsf{unrepr}\ t)[\sigma]$

$\mathsf{Repr}\ (T[\sigma]) = (\mathsf{Repr}\ T)[\sigma]$

Compatibility with Π types

$\mathsf{Repr}\ ((x : A) \to B) = (x : A) \to \mathsf{Repr}\ B$

$\mathsf{repr}\ (\lambda\ x.\ b) = \lambda\ x.\ (\mathsf{repr}\ b)$

$\mathsf{unrepr}\ (\lambda\ x.\ b) = \lambda\ x.\ (\mathsf{unrepr}\ b)$

$\mathsf{repr}\ (f\ a) = (\mathsf{repr}\ f)\ a$

$\mathsf{unrepr}\ (f\ a) = (\mathsf{unrepr}\ f)\ a$

Compatibility with universes

$\mathsf{Repr}\ \mathcal{U} = \mathcal{U}$

$\mathsf{repr}\ (\mathsf{code}\ A) = \mathsf{code}\ A$

$\mathsf{unrepr}\ (\mathsf{code}\ A) = \mathsf{code}\ A$

Compatibility with equality

$\mathsf{Repr}\ (a \equiv_A b) = \mathsf{repr}\ a \equiv_{\mathsf{Repr}A}\ \mathsf{repr}\ b$

$\mathsf{repr}\ (\mathsf{refl}\ a) = \mathsf{refl}\ (\mathsf{repr}\ a)$

$\mathsf{unrepr}\ (\mathsf{refl}\ a) = \mathsf{refl}\ (\mathsf{unrepr}\ a)$

$\mathsf{repr}\ (\mathsf{J}\ P\ d\ p) = \mathsf{J}\ (\mathsf{Repr}\ P)\ (\mathsf{repr}\ d)\ p$

$\mathsf{unrepr}\ (\mathsf{J}\ (\mathsf{Repr}\ P)\ d\ p) = \mathsf{J}\ P\ (\mathsf{unrepr}\ d)\ p$

Compatibility with eliminators

$\mathsf{repr}\ (\mathsf{elim}_S\ M\ \beta\ \delta\ x)$
$\quad = \mathsf{elim}_S\ (\mathsf{Repr}\ M)\ (\mathsf{repr}\ \beta)\ \delta\ x$

$\mathsf{unrepr}\ (\mathsf{elim}_S\ (\mathsf{Repr}\ M)\ \beta\ \delta\ x)$
$\quad = \mathsf{elim}_S\ M\ (\mathsf{unrepr}\ \beta)\ \delta\ x$

Fig. 9. (AGDA) Definitional compatibility rules for Repr. Similar rules are given for Σ and $\top$ in the formalisation. In this version, Repr only applies to codomains of functions which aligns with the substitution rule. However, it is also possible to formulate it as $\mathsf{Repr}\ ((x : A) \to B) = (x : \mathsf{Repr}\ A) \to \mathsf{Repr}\ B[\mathsf{unrepr}\ x]$.

$$\mathsf{repr\text{-}ctor}_{S.O}\ \{\gamma\}\ \nu :$$
$$\mathsf{repr}\ (\mathsf{ctor}_{S.O}\ \{\gamma\}\nu) \equiv \gamma.\alpha_O\ (\mathsf{repr}\ \nu)$$

$$\mathsf{elim\text{-}equiv}_S\ \{\gamma\}\ M\ \beta\ \delta\ x :$$
$$\mathsf{elim}_S\ \{\gamma\}\ M\ \beta\ \delta\ x \equiv \gamma.\kappa\ (\delta.\ x.\ M[\delta, \mathsf{unrepr}\ x])\ (\mathsf{repr}^*\ \beta)\ \delta\ (\mathsf{repr}\ x)$$

Fig. 10. Additional propositional equalities for Repr on constructors and eliminators. Here, repr^* applies Repr on all the recursive arguments of a displayed algebra. The rule $\mathsf{elim\text{-}equiv}_S\ \{\gamma\}\ M\ \beta\ \delta\ x$ is also derivable internally by case analysis on x.