# Custom Representations of Inductive Families
## (Extended Abstract, Research Paper)

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

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, both 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 aim to introduce a language with dependent types, and inductive families with custom representations. Representations are a version of Wadler's views [13], refined to inductive families like in Epigram [11]. 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 [14]. 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, we can indeed reason about data representations internally; in this spirit, we are awarded isomorphisms between the original and represented data.

## 1 Introduction

Inductive families are a broad generalisation of inductive data types found in most functional programming languages. Every inductive definition is equipped with a 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 [2,12], 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 [9] (which restricts the flexibility of data representation).

Wadler's views [13] 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 [11] 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 being able to handle 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 $\lambda_{\text{REP}}$ to a core language with dependent types and inductive families $\lambda_{\text{IND}}$, which allows programmers to define custom, correct-by-construction data representations. This is done through user-defined translations of the constructors and eliminators of an inductive type to a concrete implementation, which form a bijective view of the original data called a 'representation'. Representations are defined internally to the language, and require coherence properties that ensure a representation is faithful to its the original inductive family. The in the final version of the paper, we plan to contribute the following:

- A dependent type system with inductive families $\lambda_{\text{IND}}$, and its extension by representations $\lambda_{\text{REP}}$.
- 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 $\lambda_{\text{REP}}$ to $\lambda_{\text{IND}}$ 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 [1] big integers. The definition of natural numbers looks like

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

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

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

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

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

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

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

where $|\cdot|$ denotes a source translation into an untyped compilation target language with primitives $\mathsf{ubig\text{-}}*$.[1]

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 untyped target, such as recursively defined addition, multiplication, etc. The recursive versions are well-suited to proofs and reasoning, while the untyped versions based on 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] 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.

We choose to represent the Nat type as the type UBig of GMP-style unlimited-size unsigned integers, and provide translations for the constructors 0 and 1+, and the eliminator $\mathsf{elim}_{\mathsf{Nat}}$. Additionally, we have to show that the eliminator satisfies the expected computation rules of the Nat eliminator, which are postulated as propositional equalities. For this we assume access to a signature

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

which also contains propositional equalities

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

We can also define representations for functions on Nat, such as addition, multiplication, and other numeric operations, in terms of UBig primitives, in our system, akin to rewriting rules [7]

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

if we have the appropriate primitives 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 $\mathsf{Vec}\ T\ n$.[2] We are then tasked with providing terms that correspond to the constructors of Vec but for List'. These can be defined as

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

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

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

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

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.

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

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

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

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

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

Now the hard work is done; Every time we are working with a $v : \mathsf{Vec}\ T\ n$, its concrete form will be $(l, p) : \mathsf{List'}\ T\ n$ at runtime, where $l$ is the underlying list and $p$ is the proof that the length of $l$ is $n$. Under the assumption that the $\Sigma$-type's right component is irrelevant and erased at runtime, every vector is simply a list at runtime, where the length proof has been erased. In 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 $\mathsf{Vec}$ and $\mathsf{List}$ at zero runtime cost, by using the **repr** and **unrepr** operators of the language (defined in section 3). Specifically, we can define the functions

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

and it holds by reflexivity that forget-length is a left inverse of recall-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

$$\mathsf{F} : \Xi \to \mathcal{U} \qquad \mathsf{G} : \Xi \to X\ \xi \to \mathcal{U}$$

for some index family $X : \Xi \to \mathcal{U}$. 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

$$\mathsf{comp} :_{\forall \xi} \mathsf{F}\ \xi \to X\ \xi$$

which can then be used to form the family

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

If G is 'equivalent' to the algebraic ornament of F by the algebra defining comp (given by a cartesian isomorphism between the underlying polynomial functors), then it is also equivalent to the $\Sigma$-type above. The 'recomputation lemma' of algebraic ornaments [8] 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:

$$\textbf{data}\ \mathsf{Player} \left\{ \begin{array}{l} \mathsf{player} : (\mathsf{position} : \mathsf{Position}) \\ \qquad \to (\mathsf{direction} : \mathsf{Direction}) \\ \qquad \to (\mathsf{items} : \mathsf{Fin}\ \mathsf{MAX\_INVENTORY}) \\ \qquad \to (\mathsf{inventory} : \mathsf{Inventory}\ \mathsf{items}) \\ \qquad \to \mathsf{Player} \end{array} \right\}$$

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

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

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

We can define a representation for $\mathsf{Player}$ as a pair of a byte buffer and a proof that the byte buffer contents correspond to a player record. Similarly, we can define a representation for $\mathsf{Inventory}$ as a pair of a byte buffer and a proof that the byte buffer contents correspond to an inventory record of a certain size. The projection

$$\mathsf{inventory} : (p : \mathsf{Player}) \to \mathsf{Inventory}\ p.\mathsf{items}$$

is compiled into some code to slice into the inventory part of the player's byte buffer. We assume that the standard library already represents $\mathsf{Fin}$ in the same way as $\mathsf{Nat}$, so that reading the $\mathsf{items}$ field is a constant-time operation (we do not need to build a unary numeral). We can thus define the representation of $\mathsf{Player}$ as

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

with an appropriate definition of $\mathsf{IsPlayer}$ which refines a byte buffer. 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 $\mathsf{Vec}$ also depends on the representation of $\mathsf{List}$. It is possible to define a custom representation for $\mathsf{List}$ itself, for example a heap-backed array or a finger tree, and $\mathsf{Vec}$ would inherit this representation. However it will still be the case that $\textbf{Repr}\ (\mathsf{Vec}\ T\ n) \equiv \mathsf{List}\ T$, which means the **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 $\mathsf{Buf}$ type of byte buffers, we can consider the set of all types which are eventually represented as a $\mathsf{Buf}$:

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

Every such type comes with a projection function to the 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 as-buf function is effectively the identity function at runtime.[3]

## 3 A type system for data representations

In this section, we state the basics of the type system of $\lambda_{\text{REP}}$, a core language with dependent types and representations. We start with a core language with dependent types and inductive constructions $\lambda_{\text{IND}}$. We then extend this language with data representations to form $\lambda_{\text{REP}}$, which allow users to define custom representations for inductive types and other global symbols.

The core languge $\lambda_{\text{IND}}$, mirrors MLTT with $\Pi$-types and a Coquand-style universe hierarchy $\mathcal{U}_i$ [10, 2.1], extended with inductive families and global definitions. We follow a similar approach to [6] by packaging named inductive constructions and global function definitions into a signature $\Sigma$, and indexing contexts by signatures.

### 3.1 Extending $\lambda_{\text{IND}}$ with representations

We extend the language $\lambda_{\text{IND}}$ to form $\lambda_{\text{REP}}$, which allows users to define custom representations for inductive types and global functions. First, we add a modal type former

$$\mathbf{Repr} : \mathrm{Ty}\ (\Sigma \mid \Gamma) \to \mathrm{Ty}\ (\Sigma \mid \Gamma) \tag{5}$$

along with an isomorphism

$$\mathbf{repr} : \mathrm{Tm}\ (\Sigma \mid \Gamma)\ T \simeq \mathrm{Tm}\ (\Sigma \mid \Gamma)\ (\mathbf{Repr}\ T) : \mathbf{unrepr}\,. \tag{6}$$

which preserves $\Pi$-types and universes. The type **Repr** $T$ is the defined representation of the type $T$. The term **repr** takes a term of type $T$ to its representation of type **Repr** $T$, and **unrepr** undoes the effect of **repr**, treating a represented term as an inhabitant of its original type. These new constructs satisfy certain additional definitional equalities given in fig. 1, determining how these constructs can be used to rewrite type families and constructors as their defined representations.

[3] We do not guarrantee that an invocation of as-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.

$$\textsc{Repr-Ctor-Id}\ \frac{\textbf{repr}\ \mathsf{c}\ \Pi\ \textbf{as}\ \kappa \in \Sigma}{\Sigma \mid \Gamma \vdash \textbf{repr}\ (\mathsf{c}\ \delta\ \pi) \equiv \kappa[\delta, \pi] : A[\delta, \xi[\pi]]} \qquad \textsc{Repr-Data-Id}\ \frac{\textbf{repr}\ \mathsf{D}\ \Delta\ \Xi\ \textbf{as}\ A \in \Sigma}{\Sigma \mid \Gamma \vdash \textbf{Repr}\ (\mathsf{D}\ \delta\ \psi) \equiv A[\delta, \psi]\ \text{type}}$$

**Fig. 1.** Definitional equalities for **Repr** and **repr** relating to data types and constructors with defined representations. Similar equalities hold for representations of global function definitions and eliminators, albeit propositionally.

We state some basic lemmas below, whose proofs we reserve for the full version of the paper.

**Lemma 1.** *The term formers **repr** and **unrepr** are injective, i.e.*

$$\frac{\Sigma \mid \Gamma \vdash \textbf{\textit{repr}}\ t \equiv \textbf{\textit{repr}}\ t' : \textbf{\textit{Repr}}\ T}{\Sigma \mid \Gamma \vdash t \equiv t' : T} \qquad \frac{\Sigma \mid \Gamma \vdash \textbf{\textit{unrepr}}\ t \equiv \textbf{\textit{unrepr}}\ t' : T}{\Sigma \mid \Gamma \vdash t \equiv t' : \textbf{\textit{Repr}}\ T}$$

**Lemma 2.** *The type former **Repr** is injective up to internal isomorphism, i.e.*

$$\frac{\Sigma \mid \Gamma \vdash \textbf{\textit{Repr}}\ T \equiv \textbf{\textit{Repr}}\ T'\ \textit{type}}{\Sigma \mid \Gamma \vdash p : T \simeq T'}$$

*Moreover, this isomorphism is computationally irrelevant, i.e. it compiles to the identity function.*

## 4 Translating from $\boldsymbol{\lambda}_{\textsc{rep}}$ to $\boldsymbol{\lambda}_{\textsc{ind}}^{\textsc{Ext}}$

We can define a translation step $\mathcal{R}$ from $\lambda_{\textsc{rep}}$ to $\lambda_{\textsc{ind}}^{\textsc{Ext}}$, meant to be applied during the compilation process. More specifically, the translation target is the extensional flavour of $\lambda_{\textsc{ind}}$ by adding the equality reflection rule. We do this by translating well-formed contexts, substitutions, types, and terms in a mutual manner such that definitional equality is preserved. In other words, $\mathcal{R}$ is strict CWF [5] morphism comprising of setoid homomorphisms on all syntactical categories. Informally, $\mathcal{R}$ preserves the structure of $\lambda_{\textsc{ind}}$, but maps constructs to their terminal representations. Eliminator coherence rules are preserved by reflecting the propositional coherence rules provided by the defined representations. We prove some desired properties of $\mathcal{R}$ [3] such as typing and computational soundness, and preservation of consistency. The final program can then be converted into an untyped language which erases irrelevant data. We can recover a program in $\lambda_{\textsc{ind}}$ by translating extensional typing derivations to intensional proofs [15].

## 5 Implementation

Superfluid is a programming language with dependent $\Pi$ types, 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. The GNU MP bignum library. `https://gmplib.org/`, accessed: 2024-12-8
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)
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)
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)
5. Castellan, S., Clairambault, P., Dybjer, P.: Categories with families: Unityped, simply typed, and dependently typed (Apr 2019)
6. Cockx, J., Abel, A.: Elaborating dependent (co)pattern matching. Proc. ACM Program. Lang. **2**(ICFP), 1–30 (Jul 2018)
7. 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)
8. Dagand, P.E., McBride, C.: A categorical treatment of ornaments. arXiv [cs.PL] (Dec 2012)
9. Diehl, L., Firsov, D., Stump, A.: Generic zero-cost reuse for dependent types. Proc. ACM Program. Lang. **2**(ICFP), 1–30 (Jul 2018)
10. Gratzer, D., Kavvos, G.A., Nuyts, A., Birkedal, L.: Multimodal dependent type theory. arXiv [cs.LO] (Nov 2020)
11. Mcbride, C., Mckinna, J.: The view from the left. J. Funct. Programming **14**(1), 69–111 (Jan 2004)
12. Moon, B., Eades, III, H., Orchard, D.: Graded modal dependent type theory. In: Programming Languages and Systems. pp. 462–490. Springer International Publishing (2021)
13. 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)
14. Wadler, P.: Deforestation: transforming programs to eliminate trees. Theor. Comput. Sci. **73**(2), 231–248 (Jun 1990)
15. Winterhalter, T., Sozeau, M., Tabareau, N.: Eliminating reflection from type theory. In: Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs. ACM, New York, NY, USA (Jan 2019)