# Custom Representations of Inductive Families

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, where dependent pattern matching automatically handles a lot of the otherwise manual work of reasoning about index refinement. However, when it comes to compilation, the default representation of inductive types can be inefficient. Often, we want multiple indexed views of the same data, which has runtime overhead with current methods. In this paper, introduce a language with dependent types, and inductive families with custom representations. A type family $F$ can be used to represent the inductive family D as long as $T$ can mimic $D$'s constructors and induction principle. This way, we can build up a standard library of convenient inductive families, whose re-indexing conversion functions are erased at compile-time. In particular, we show how we can reproduce and extend the 'Nat-hack' from Agda, Idris and Lean, fully within the language. Since we are working with dependent types, we can indeed reason about data representations internally; in this spirit, we are awarded with a correctness proof of the Nat-hack, for free.

## 1 Introduction

Inductive families are a broad generalisation of inductive data types found in most functional programming languages. Every inductive definition is equipped with an induction principle 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 often comes at a cost; the runtime representation of inductive types is a linked tree structure. By default, a list as an inductively defined data structure is stored as a linked list, and a natural number is stored as a unary number where each digit is an empty heap cell. This representation is not always the most efficient for all operations, and often forces users to rely on more efficient machine primitives to achieve good performance. The 'Nat-hack' in languages with dependent types is a prime example of this, where natural numbers are represented as GMP-style big integers for efficient arithmetic operations.

In this paper, we propose an extension to a core language with dependent types and inductive families which allows users to define custom representations for inductive types. This is done through user-defined translations of the constructors and eliminators of an inductive type to a concrete implementation,

which forms a 'representation'. Representations are defined in the language itself, and come with coherence properties that ensure that the representation forms an algebra isomorphic to the initial algebra of the inductive type.

## 2 A tour of data representations

### 2.1 Natural numbers

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

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

This definition is convenient because it generates a Peano-style induction principle

$$\begin{aligned} \mathsf{elim}_{\mathsf{Nat}} : (P : \mathsf{Nat} \to \mathsf{Type}) \\ \to (m_{\mathsf{0}} : P\ \mathsf{0}) \to (m_{\mathsf{1+}} : (n : \mathsf{Nat}) \to P\ n \to P\ (\mathsf{1+}\ n)) \\ \to (s : \mathsf{Nat}) \to P\ s\,. \end{aligned}$$

Moreover, dependent pattern matching with structural recursion on Nat can be elaborated into invocations of $\mathsf{elim}_{\mathsf{Nat}}$, and lemmas such as constructor injectivity, disjointness and acyclicity [Constructions on constructors, eliminating dep pattern matching] can be substituted automatically to simplify the context and goal. This way, if addition is defined using recursion and pattern matching, proofs like the commutativity of addition or the additive identity of 0 are easy to write.

When it comes to compilation, without further intervention, the Nat type is represented in unary form, where each digit is an empty heap cell. This is inefficient for a lot of the basic operations on natural numbers; addition is a linear-time operation and multiplication is quadratic. For this reason, most real-world implementations will treat Nat specially, swapping the default inductive type representation with one based on GMP integers. The basic idea is to perform the replacements

$$\begin{aligned} |\mathsf{0}| &= \mathsf{ubig\text{-}zero} \\ |\mathsf{1+}| &= \mathsf{ubig\text{-}add\ ubig\text{-}one} \\ |\mathsf{elim}_{\mathsf{Nat}}\ P\ m_{\mathsf{0}}\ m_{\mathsf{1+}}\ s| &= \mathsf{ubig\text{-}if\text{-}zero}\ |s|\ |m_{\mathsf{0}}|\ |m_{\mathsf{1+}}| \end{aligned}$$

where $|\cdot|$ denotes a source translation into an untyped compilation target language, and ubig-$*$ are primitives of the target language.

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 $|+| = \mathsf{ubig\text{-}add}$, $|\times| = \mathsf{ubig\text{-}mul}$, etc. Idris 2

will in fact look for any 'Nat-like' types and apply this optimisation. A Nat-like type is any type whose shape is $\mathbb{2}$ and positions are $\{0 \mapsto \mathbb{0}, 1 \mapsto \mathbb{1}\}$. A similar optimisation is also done with list-like and boolean-like types.

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, our first goal is to extend this optimisation to work for any data type. Additionally, the optimisation is defined as a compilation step, which means that if non-trivial representations are provided by the user, their correctness is not guaranteed. To address this, our framework requires that representations are fully typed, ensuring that the behaviour of the representation of a data type matches the behaviour of the data type itself.

### 2.2 Views on arrays

In functional languages, lists are automatically represented as linked lists by virtue of their inductive definition. This representation is particularly suitable for the recursion pattern where the head of the list is processed first, and the tail is processed recursively. However, it is not always the most efficient for all operations. For example, accessing the $n$-th element of a list requires $O(n)$ steps. In contrast, contiguous, homogenous arrays provide $O(1)$ access to elements, but cannot be defined inductively in any nice way. There is clearly a bijection between lists and arrays, which makes lists a good candidate for a custom representation.

We assume access to a primitive type Array $T$ representing contiguous heap arrays of elements of type $T$, along with primitives

$$
\begin{aligned}
&\mathsf{array} : (n : \mathsf{Nat}) \to (\mathsf{Fin}\ n \to T) \to \mathsf{Array}\ T \\
&\mathsf{array\text{-}length} : \mathsf{Array}\ T \to \mathsf{Nat} \\
&\mathsf{array\text{-}index} : (a : \mathsf{Array}\ T) \to \mathsf{Fin}\ (\mathsf{array\text{-}length}\ a) \to T \\
\\
&\mathsf{array\text{-}elim} : (P : \mathsf{Array}\ T \to \mathsf{Type}) \\
&\quad \to (m_1 : P\ \mathsf{array\text{-}nil}) \\
&\quad \to (m_2 : (x : T) \to (xs : \mathsf{Array}\ T) \to P\ xs \to P\ (\mathsf{array\text{-}cons}\ x\ xs)) \\
&\quad \to (s : \mathsf{Array}\ T) \to P\ s\,.
\end{aligned}
$$

using the auxilliary definitions

$$
\begin{aligned}
&\mathsf{array\text{-}nil} : \mathsf{Array}\ T \\
&\mathsf{array\text{-}nil} = \mathsf{array}\ 0\ (\mathsf{fin\text{-}zero\text{-}void}\ T)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{array\text{-}cons} : T \to \mathsf{Array}\ T \to \mathsf{Array}\ T \\
&\mathsf{array\text{-}cons}\ x\ xs = \mathsf{array}\ (1{+}\ \mathsf{array\text{-}length}\ n)\ \left\{ \begin{array}{c} 0\mathsf{f} \mapsto x \\ 1\mathsf{f}{+}\ m \mapsto \mathsf{array\text{-}index}\ xs\ m \end{array} \right\}
\end{aligned}
$$

In addition, we require that the following equalities hold definitionally:

$$\mathsf{array\text{-}elim}\ P\ a\ b\ \mathsf{array\text{-}nil} \equiv a : P\ \mathsf{array\text{-}nil}$$
$$\mathsf{array\text{-}elim}\ P\ a\ b\ (\mathsf{array\text{-}cons}\ x\ xs) \equiv b\ x\ xs : P\ (\mathsf{array\text{-}cons}\ x\ xs)$$

The symbols array-nil, array-cons, and array-elim along with the above definitional equalities form an inductive interface, which can be used as a representation for the List data type:

$$\textbf{repr}\ \mathsf{List}\ T\ \textbf{as}\ \mathsf{Array}\ T \left\{ \begin{array}{c} [] \ \textbf{as}\ \mathsf{array\text{-}nil} \\ (x{::}xs)\ \textbf{as}\ \mathsf{array\text{-}cons}\ x\ xs \\ \mathsf{elim}_{\mathsf{List}}\ \textbf{as}\ \mathsf{array\text{-}elim} \end{array} \right\}$$

Now the indexing function for lists can be represented as the indexing function for arrays, and the length function for lists can be represented as the length function for arrays. To do this, it is required that some further definitional equalities hold:

$$\mathsf{array\text{-}elim}\ (\kappa\ \mathsf{Nat})\ 0\ (\kappa\ \kappa\ 1{+}) \equiv \mathsf{array\text{-}length} : \mathsf{Array}\ T \to \mathsf{Nat}$$

$$\mathsf{array\text{-}elim}\ (\lambda a.\ \mathsf{Fin}\ (\mathsf{array\text{-}length}\ a) \to T)\ (\mathsf{fin\text{-}zero\text{-}void}\ T)\ (\kappa\ .\ \mathsf{elim}_{\mathsf{Fin}}\ T)$$
$$\equiv\ \mathsf{array\text{-}index} : \mathsf{Array}\ T \to \mathsf{Fin}\ (\mathsf{array\text{-}length}\ a) \to T$$

Then we can set

$$\textbf{repr}\ \mathsf{index}\ \textbf{as}\ \mathsf{array\text{-}index}$$
$$\textbf{repr}\ \mathsf{length}\ \textbf{as}\ \mathsf{array\text{-}length}\,.$$

This works because the definition of length elaborates to the algebra above which is equated to array-length, satisfying the definitional equality requirements.

### 2.3 Reindexing and forgetful maps for free

Let us assume that in our language, for some data type

$$\textbf{data}\ \mathsf{D}\ \Delta : \Xi \to \mathsf{Type}$$

we have access to an internal description $\mathsf{desc}_{\mathsf{D}} : \Delta \to \mathsf{Desc}\ \Xi$, along with an interpretation function

$$[\![\_]\!] : \mathsf{Desc}\ \Xi \to (\Xi \to \mathsf{Type}) \to (\Xi \to \mathsf{Type})$$

and a fixpoint operator

$$\textbf{data}\ \mu\ (D : \mathsf{Desc}\ \Xi)\ (\xi : \Xi)\ \{\ \mathsf{fix} : [\![D]\!]\ (\mu\ D)\ \xi \to \mu\ D\ \xi\ \}$$

such that $\mu\ (\mathsf{desc}_{\mathsf{D}}\ \delta) \simeq \mathsf{D}\ \delta$ is a strong bijection. Then, we might choose to define the representation of $D$ as $\mu\ (\mathsf{desc}_{\mathsf{D}}\ \delta)$; in this way, we represent the 'primitive' datatype D as the interpretation of its code $\mathsf{desc}_{\mathsf{D}}$.

Now consider that we have an indexed datatype

$$\textbf{data}\ \mathsf{C}\ \Delta : (\xi : \Xi) \to \Phi[\xi] \to \mathsf{Type}$$

which is a refined version of D. If we can construct an algebra

$$\mathsf{phi}\ \delta : [\![\mathsf{desc_D}]\!]\ \Phi\ \xi \to \Phi\ \xi$$

that computes the index $\Phi$, then we can form the description

$$\mathsf{alg\text{-}orn}\ (\mathsf{phi}\ \delta) : \mathsf{Desc}\ (\Sigma\ \Xi\ \Phi)$$

which is the ornament induced by the algebra phi. If a strong bijection can be established between $\mu\ (\mathsf{alg\text{-}orn}\ (\mathsf{phi}\ \delta))$ and $\mathsf{C}\ \delta$, then the former can be used as the representation of the latter. Finally, this allows us to define a zero-cost forgetful conversion function from C and D, which is erased at compile-time, as

$$\mathsf{forget}\text{-}\Phi : \mathsf{C}\ \delta\ \xi\ \phi \to \mathsf{D}\ \delta\ \xi \tag{1}$$

$$\mathsf{forget}\text{-}\Phi\ c\ d = \textbf{unrepr}\ (\mathsf{alg\text{-}orn\text{-}forget}\ (\textbf{repr}\ c)) \tag{2}$$

*Example 1.* Let $\Delta = (T : \mathsf{Type})$, $\Xi = \cdot$ and $\Phi = (n : \mathsf{Nat})$. Let $\mathsf{D}\ T = \mathsf{List}\ T$, and $\mathsf{C}\ T\ n = \mathsf{Vec}\ T\ n$. Then let $\mathsf{desc_{List}}$ be the description for lists

$$\sigma\ [\mathsf{nil}, \mathsf{cons}]\ \left\{ \begin{array}{l} \mathsf{nil} \mapsto \mathsf{end} \\ \mathsf{cons} \mapsto \sigma\ T\ (\lambda\_.\ \mathsf{node}\times\ \mathsf{end}) \end{array} \right\}$$

and construct the algebra $\mathsf{phi} = \mathsf{length\text{-}alg}$ mirroring the length function $\mathsf{length} : \mathsf{List}\ T \to \mathsf{Nat}$. It is easy to construct a strong bijection $\mu\ (\mathsf{alg\text{-}orn}\ (\mathsf{length\text{-}alg}\ T))\ n \simeq \mathsf{Vec}\ T\ n$, so we can represent the latter as the former, and define a zero-cost function to forget the length of a vector as forget-Nat.

### 2.4 Binary data

## 3 A type system for data representations

In this section, we describe a type system for data representations in a language with dependent types. We start by defining 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. We present these languages with intrinsically well-formed contexts, types, and terms, quotiented by their definitional equality rules [altenkirch].

### 3.1 A core language with inductive types, $\boldsymbol{\lambda_{\text{IND}}}$

The core languge we start with is $\lambda_{\text{IND}}$. It contains $\Pi$-types and a single universe Type with Type : Type. We are not concerned with universe polymorphism or a

sound logical interpretation as this is orthogonal to the main focus of this work. Nevertheless, all the results should be readily extensible to a sound language with a universe hierarchy. We follow a similar approach to [1] by packaging named inductive constructions and global function definitions into a signature, and indexing contexts by signatures. A typing judgement looks like

$$\Sigma \mid \Gamma \vdash t : T$$

and is read as "in signature $\Sigma$ and context $\Gamma$, term $t$ has type $T$". The rules for signatures, contexts and telescopes are given in fig. 1.

$$\textsc{Sig-Empty}\ \frac{}{\cdot\ \text{sig}} \qquad \textsc{Sig-Extend}\ \frac{\Sigma\ \text{sig} \qquad \Sigma \vdash Z}{\Sigma, Z\ \text{sig}} \qquad \textsc{Con-Empty}\ \frac{\Sigma\ \text{sig}}{\Sigma \vdash \cdot\ \text{con}} \qquad \textsc{Con-Extend}\ \frac{\Sigma \vdash \Gamma\ \text{con} \qquad \Sigma \mid \Gamma \vdash T\ \text{type}}{\Sigma \vdash \Gamma, T\ \text{con}}$$

$$\textsc{Tel-Empty}\ \frac{\Sigma \vdash \Gamma\ \text{con}}{\Sigma \mid \Gamma \vdash \cdot\ \text{tel}} \qquad \textsc{Tel-Extend}\ \frac{\Sigma \mid \Gamma \vdash \Delta\ \text{tel} \qquad \Sigma \mid \Gamma \vdash T\ \text{type}}{\Sigma \mid \Gamma \vdash \Gamma, T\ \text{tel}}$$

**Fig. 1.** Rules for signatures, contexts and telescopes in $\lambda_{\text{IND}}$.

Telescopes [deBrujin] are very similar to contexts, but restricted to types from a single stage and well formed with respect to a context $\Gamma$, meaning that telescopes can contain open terms. We use the notation $\Delta \to t$ to denote a repeated function type with parameters from $\Delta$ and codomain $T$ which may depend on the parameters. Additionally, we will sometimes explicitly bind the names of a telescope such as $(\delta : \Delta) \to T[\delta]$. Similar syntax is used to extend contexts with telescopes: $\Gamma, \Delta$ or $\Gamma, \delta : \Delta$.

Awkward spacing!

Next, the rules for well-formed items in signatures are given in fig. 2.

$$\textsc{Data-Item}\quad \frac{\Sigma \mid \cdot \vdash \Delta \text{ tel} \qquad \Sigma \mid \Delta \vdash \Xi \text{ tel} \qquad \mathsf{D} \text{ label} \notin \Sigma}{\Sigma \vdash \mathbf{data}\ \mathsf{D}\ \Delta : \Xi \to \mathsf{Type}}$$

$$\textsc{Ctor-Item}\quad \frac{\mathbf{data}\ \mathsf{D}\ \Delta : \Xi \to \mathsf{Type} \in \Sigma \qquad \mathbf{closed}\ \mathsf{D}, \_ \notin \Sigma \qquad \mathsf{C} \text{ label} \notin \Sigma \qquad \Sigma \mid \Delta \vdash \Pi \text{ tel} \qquad \Sigma \mid \Delta, \Pi \vdash \xi : \Xi}{\Sigma \vdash \mathbf{ctor}\ \mathsf{C}\ \Pi : \mathsf{D}\ \Delta\ \xi}$$

$$\textsc{Closed-Item}\quad \frac{\mathbf{data}\ \mathsf{D}\ \Delta : \Xi \to \mathsf{Type} \in \Sigma \qquad \forall i \in I.\ \mathbf{ctor}\ \mathsf{C}_i\ \Pi_i : \mathsf{D}\ \Delta\ \xi_i \in \Sigma \qquad \mathbf{closed}\ \mathsf{D}, \_ \notin \Sigma}{\Sigma \vdash \mathbf{closed}\ \mathsf{D}, \mathsf{C}}$$

$$\textsc{Def-Item}\quad \frac{\Sigma \mid \cdot \vdash m : M \qquad \mathsf{f} \text{ label} \notin \Sigma}{\Sigma \vdash \mathbf{def}\ \mathsf{f} : M = m}$$

**Fig. 2.** Rules for items in signatures in $\lambda_{\text{IND}}$.

$$\textsc{Data-Form}\quad \frac{\mathbf{data}\ \mathsf{D}\ \Delta : \Xi \to \mathsf{Type} \in \Sigma}{\Sigma \mid \Gamma \vdash \mathsf{D} : \Delta \to \Xi \to \mathsf{Type}}$$

$$\textsc{Data-Intro}\quad \frac{\mathbf{data}\ \mathsf{D}\ \Delta : \Xi \to \mathsf{Type} \in \Sigma \qquad \mathbf{ctor}\ \mathsf{C}\ \Pi : \mathsf{D}\ \Delta\ \xi \in \Sigma}{\Sigma \mid \Gamma \vdash \mathsf{C} : (\delta : \Delta) \to (\pi : \Pi[\delta]) \to \mathsf{D}\ \delta\ (\xi[\delta, \pi])}$$

$$\textsc{Data-Case}\quad \frac{\mathbf{data}\ \mathsf{D}\ \Delta : \Xi \to \mathsf{Type} \in \Sigma \qquad \forall i \in I.\ \mathbf{ctor}\ \mathsf{C}_i\ \Pi_i : \mathsf{D}\ \xi_i \in \Sigma \qquad \mathbf{closed}\ \mathsf{D}\ \mathsf{C} \in \Sigma}{\Sigma \mid \Gamma \vdash \mathsf{case}_\mathsf{D} : (\delta : \Delta) \to \mathrm{Case}((\xi : \Xi[\delta], x : \mathsf{D}\ \delta\ \xi), \{(\pi : \Pi_i[\delta]), (\xi_i[\delta], \mathsf{C}_i\ \delta\ \pi)\}_i)}$$

$$\textsc{Data-Case-ID}_j\quad \frac{\mathbf{data}\ \mathsf{D}\ \Delta : \Xi \to \mathsf{Type} \in \Sigma \qquad \forall i \in I.\ \mathbf{ctor}\ \mathsf{C}_i\ \Pi_i : \mathsf{D}\ \xi_i \in \Sigma \qquad \mathbf{closed}\ \mathsf{D}\ \mathsf{C} \in \Sigma \qquad \Sigma \mid \Gamma \vdash \delta : \Delta}{\Sigma \mid \Gamma \vdash \mathrm{ValidCase}(\mathsf{case}_\mathsf{D}\ \delta)}$$

$$\textsc{Def-Intro}\quad \frac{\mathbf{def}\ \mathsf{f} : M = m \in \Sigma}{\Sigma \mid \Gamma \vdash \mathsf{f} : M}$$

**Fig. 3.** Terms and types associated to items in signatures in $\lambda_{\text{IND}}$.

The rules for data constructors do not consider the recursive occurences of D explicitly, which means that strict positivity is not ensured. Rather, we assume

that a separate check is performed to ensure that the defined types adhere to strict positivity, if necessary. As a result, the eliminator for data types does not provide the inductive hypotheses directly, but we assume that the language allows general recursion. Similarly to the positivity requirements, we expect that if termination is a desirable property of the system, it is ensured separately to the provided typing rules.

In fig. 3, the type $\Sigma \mid \Gamma \vdash \mathrm{Case}(\Phi, \{\Pi_k, \phi_k\}_k) : \mathsf{Type}$ is defined as

$$\frac{\Sigma \mid \Gamma \vdash \Phi \text{ tel} \qquad \forall k \in K.\ \Sigma \mid \Gamma \vdash \Pi_k \text{ tel} \qquad \forall k \in K.\ \Sigma \mid \Gamma, \Pi_k \vdash \phi_k : \Phi}{\mathrm{Case}(\Phi, \{\Pi_k, \phi_k\}_k) := (P : \Phi \to \mathsf{Type}) \to \{(\Pi_k \to P\ \phi_k)\}_k \to (\phi : \Phi) \to P\phi}$$

The $\Sigma \mid \Gamma \vdash \mathrm{ValidCase}(c)$ condition is defined inductively as

$$\frac{\begin{array}{c}\Sigma \mid \Gamma \vdash c : \mathrm{Case}(\Phi, \{\Pi_k, \phi_k\}_k) \\ \forall j \in K.\ \Sigma \mid \Gamma, P : \Phi \to \mathsf{Type}, \{\kappa_k : \Pi_k \to P\ \phi_k\}_k, \pi : \Pi_j \\ \vdash c\ P\ \{\kappa_k\}_k\ \phi_j[\pi] \equiv \kappa_j\ \pi : P\ \phi_j[\pi]\end{array}}{\Sigma \mid \Gamma \vdash \mathrm{ValidCase}(c)}$$

### 3.2 Extending $\lambda_{\text{IND}}$ with data representations

We base language $\lambda_{\text{IND}}$ to form $\lambda_{\text{REP}}$, which allows users to define custom representations for inductive types and global functions.

$$\textsc{Repr-Data}\quad \frac{\mathbf{data}\ \mathsf{D}\ \Delta : \Xi \to \mathsf{Type} \in \Sigma \qquad \Sigma \mid \Delta, \Xi \vdash A : \mathsf{Type}}{\Sigma \vdash \mathbf{repr}\ \mathsf{D}\ \Delta\ \Xi\ \mathbf{as}\ A}$$

$$\textsc{Repr-Ctor}\quad \frac{\begin{array}{c}\mathbf{data}\ \mathsf{D}\ \Delta : \Xi \to \mathsf{Type} \in \Sigma \qquad \forall i \in I.\ \mathbf{ctor}\ \mathsf{C}_i\ \Pi_i : \mathsf{D}\ \Delta\ \xi_i \in \Sigma \\ \mathbf{repr}\ \mathsf{D}\ \Delta\ \Xi\ \mathbf{as}\ A \in \Sigma \qquad \forall i < k.\ \mathbf{repr}\ \mathsf{C}_i\ \mathbf{as}\ \Pi_i t_i \in \Sigma \qquad \Sigma \mid \Delta, \Pi_k \vdash t_k : A\ \Delta\ \xi_k\end{array}}{\Sigma \vdash \mathbf{repr}\ \mathsf{C}_k\ \Pi_k\ \mathbf{as}\ t_k}$$

$$\textsc{Repr-Case}\quad \frac{\begin{array}{c}\mathbf{data}\ \mathsf{D}\ \Delta : \Xi \to \mathsf{Type} \in \Sigma \\ \mathbf{repr}\ \mathsf{D}\ \Delta\ \Xi\ \mathbf{as}\ A \in \Sigma \qquad \forall i \in I.\ \mathbf{ctor}\ \mathsf{C}_i\ \Pi_i : \mathsf{D}\ \Delta\ \xi_i \in \Sigma \\ \forall i \in I.\ \mathbf{repr}\ \mathsf{C}_i\ \mathbf{as}\ \Pi_i t_i \in \Sigma \qquad \mathbf{closed}\ \mathsf{D}\ \mathsf{C} \in \Sigma \\ \Sigma \mid \delta : \Delta \vdash c : \mathrm{Case}((\xi : \Xi[\delta], x : A\ \delta\ \xi), \{(\pi : \Pi_i[\delta]), (\xi_i[\delta], A\ \delta\ \pi)\}_i) \\ \Sigma \mid \Delta \vdash \mathrm{ValidCase}(c)\end{array}}{\Sigma \vdash \mathbf{repr}\ \mathsf{case}_{\mathsf{D}}\ \Delta\ \mathbf{as}\ c}$$

$$\textsc{Repr-Def}\quad \frac{\mathbf{def}\ \mathsf{f} : M = m \in \Sigma \qquad \Sigma \mid \cdot \vdash a : M \qquad \Sigma \mid \cdot \vdash m \equiv a : M}{\Sigma \vdash \mathbf{repr}\ \mathsf{f}\ \mathbf{as}\ a}$$

**Fig. 4.** Rules for data representations in $\lambda_{\text{REP}}$.

### 3.3 Properties of $\lambda_{\text{REP}}$

- Decidability of equality
- Confluence of reduction
- Strong normalization

## 4 Translating from $\lambda_{\text{REP}}$ to $\lambda_{\text{IND}}$

- Setoid homomorphism
- Faithful translation (setoid injectivity)
- Corollaries

## References

1. Cockx, J., Abel, A.: Elaborating dependent (co)pattern matching. Proc. ACM Program. Lang. **2**(ICFP), 1–30 (Jul 2018), https://doi.org/10.1145/3236770