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. 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 introduce a language with dependent types, and inductive families with custom representations. Representations are a version of Wadler's views [11], refined to inductive families like in Epigram [9]. 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 [12]. 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 particular, we show how we can reproduce the 'Nat-hack' from Agda, Idris and Lean, fully within our language. 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 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. A list as an inductively defined data structure is stored as a linked list on the heap, 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 desirable 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 [CITE?].

This is not the only issue with the compilation of inductive families. Despite advances in the erasure of irrelevant indices in inductive families [3] and the use of

theories with irrelevant fragments [2,10,1], there is still a need to convert between different indexed views of the same data. For example, the function to convert from List T to 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 harms dependent pattern matching due to the presence of non-structural witnesses), or a Church encoding is used in a Curry-style context [6] (which restricts the flexibility of data representation).

Wadler's views [11] 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 [9] 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 to a core language with dependent types and inductive families 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 defines an initial algebra for the endofunctor associated with the original inductive family. We define a type- and equality-preserving translation from the language with representations to a language without them, which provably erases all inductive types with defined representations from the program.

The in the final version of the paper, we plan to contribute the following:

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

2 A tour of data representations

A common optimisation done by programming languages with dependent types such as Idris 2 and Lean is to represent natural numbers as GMP-style big

integers. The core definition of natural numbers looks like

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

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

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

Moreover, dependent pattern matching with structural recursion on Nat can be elaborated into invocations of $\mathsf{elim}_{\mathsf{Nat}}$ [7,5,4], and lemmas such as constructor injectivity, disjointness and acyclicity [8] 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

$$|\mathsf{0}| = \mathsf{ubig\text{-}zero} \tag{2}$$

$$|\mathsf{1+}| = \mathsf{ubig\text{-}add\ ubig\text{-}one} \tag{3}$$

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

where $|\cdot|$ denotes a source translation into an untyped compilation target language with primitives ubig-$*$.

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 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.

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.

2.1 Main idea of representations

TODO

2.2 Natural numbers as big unsigned integers

To build up data out of something other than just inductive types, we need access to some primitive types which are 'provided by the runtime environment'. It would be reasonable to assume access to types corresponding to machine words of different widths, but for brevity we will go a step further and assume access to an unlimited-size unsigned integer type UBig, along with:

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

and propositional equalities

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

The equalities above postulate that UBig's eliminator satisfies the expected computation rules of the Nat eliminator, albeit propositionally rather than definitionally. In contrast to the earlier example (3), this interface is fully *typed*.

Having access to UBig, we can define a representation for the Nat data type which maps the constructors and eliminators of Nat to the primitives of UBig:

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

This will effectively erase the Nat type from the compiled program, replacing all occurrences with the UBig type and its primitives. This is an example of a *representation definition*, which is part of the global context of a program, which can access top-level symbols.

2.3 Vectors are just certain lists

We can use representations to share the underlying data when converting between Vec and List, so that the conversion is 'compiled away'. We can do this by representing the more indexed type as a refinement of the less indexed type by an appropriate relation. For the case of Vec, we know intuitively that

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

so we can start by choosing the type

$$\mathsf{List'}\ T\ n := \{l : \mathsf{List}\ T \mid \mathsf{length}\ l = n\}$$

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

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

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

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

Some heavy lifting can be done by dependent pattern matching, where we refine 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.

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

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 elim-List'. This completes the definition of the representation of Vec as List', which would be written as

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

Now the hard work is done; Every time we are working with a $v : \mathsf{Vec}\ T\ n$, its real 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 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 6 we show how this erasure is achieved in practice in Superfluid using quantitative type theory [2].

We can utilise this representation to convert between Vec and List at zero runtime cost, by using the **repr** and **unrepr** operators of the language. 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.

2.4 General reindexing

The idea from the previous example can be generalised to any data type. Given a simple data type $\mathsf{F} : \mathcal{U}$ and an algebra

$$\alpha : \tilde{\mathsf{F}}\ J \to J$$

which computes some index J from F, we can construct a new type $\mathsf{F}^\alpha : J \to \mathcal{U}$ which has the same constructors and arities, but with with new data arguments for the indices and a call to α in the recursive arguments and return type. For this new type F^α, we can define a representation as

$$\textbf{repr}\ \mathsf{F}^\alpha\ j\ \textbf{as}\ \{x : \mathsf{F} \mid \mathsf{fold}\ \alpha\ x = j\} \left\{ \begin{array}{r} \mathsf{c}_i\ \textbf{as}\ \mathsf{c}_i \\ \mathsf{elim}_{\mathsf{F}}\ \textbf{as}\ \mathsf{elim}_{\mathsf{G}} \\ \textbf{by}\ \mathsf{elim\text{-}iso} \end{array} \right\}$$

F,

If the interpretation of the algebraic ornament defined by α is the same as the type $\mathsf{G}\ (f\ i)$, then we can represent $\mathsf{F}\ i$ as $\{g : \mathsf{G}\ (f\ i) \mid \alpha\ g = \mathsf{im}\ f\ i\}$.

one can construct a data type $\mathsf{G} : J \to \mathcal{U}$ which is

2.5 Transitivity

Representations are transitive, so in the previous example, the 'terminal' representation of Vec also depends on the representation of List. It is possible to define a custom representation for List itself, for example a heap-backed array or a finger tree, and Vec would inherit this representation. However it will still be the case that **Repr** (Vec T n) $\equiv$ 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 terms which satisfy a certain 'eventual' representation. For example, given a Buf type of byte buffers, we can consider the set of all types which are eventually represented as a Buf:

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

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

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

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. 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.

2.6 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

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

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:

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

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:

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

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:

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

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

Then we can set

$$\begin{aligned}
\mathbf{repr}&\mathsf{index}\ \mathbf{as}\ \mathsf{array\text{-}index} \\
\mathbf{repr}&\mathsf{length}\ \mathbf{as}\ \mathsf{array\text{-}length}\,.
\end{aligned}$$

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

2.7 Reindexing and forgetful maps for free

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

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

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

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

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

Now consider that we have an indexed datatype

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

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 Φ, 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

$$\mathsf{forget}\text{-}\Phi : \mathsf{C}\ \delta\ \xi\ \phi \to \mathsf{D}\ \delta\ \xi \tag{5}$$

$$\mathsf{forget}\text{-}\Phi\ c\ d = \textbf{unrepr}\ (\mathsf{alg\text{-}orn\text{-}forget}\ (\textbf{repr}\ c)) \tag{6}$$

Example 1. Let $\Delta = (T : \mathcal{U})$, $\Xi = \cdot$ 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

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

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.

2.8 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 λ_{IND}. We then extend this language with data representations to form λ_{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, λ_{IND}

The core languge we start with is λ_{IND}. It contains Π-types and a single universe $\mathcal{U}$ with $\mathcal{U} : \mathcal{U}$. 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 [4] 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 Σ and context Γ, term t has type T". The rules for signatures, contexts and telescopes are given in fig. 1.

$$\textsc{Sig-Empty}\ \frac{}{\cdot\ \text{sig}} \qquad \textsc{Sig-Extend}\ \frac{\Sigma\ \text{sig} \qquad \Sigma \vdash Z}{\Sigma, Z\ \text{sig}}$$

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

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

Fig. 1. Rules for signatures, contexts and telescopes in λ_{IND}.

Telescopes [deBrujin] are very similar to contexts, but restricted to types from a single stage and well formed with respect to a context Γ, meaning that telescopes can contain open terms. We use the notation $\Delta \to t$ to denote a repeated function type with parameters from Δ 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: Γ, Δ or $\Gamma, \delta : \Delta$.

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

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

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

$$\textsc{Def-Type}\ \frac{\Sigma \mid \cdot \vdash M : \mathcal{U} \qquad \mathsf{f}\ \mathrm{label} \notin \Sigma}{\Sigma \vdash \mathbf{def}\ \mathsf{f} : M = m} \qquad \textsc{Def-Item}\ \frac{\Sigma, \mathbf{def}\ \mathsf{f} : M \mid \cdot \vdash m : M \qquad \mathsf{f}\ \mathrm{label} \notin \Sigma}{\Sigma \vdash \mathbf{def}\ \mathsf{f} : M = m}$$

Fig. 2. Rules for items in signatures in λ_{IND}.

Awkward spacing!

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

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

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

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

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

Fig. 3. Terms and types associated to items in signatures in λ_{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 $\Sigma \mid \Gamma \vdash \mathrm{Case}(\Phi, \{\Pi_k, \phi_k\}_k) : \mathcal{U}$ is defined as

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

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

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

3.2 Extending λ_{IND} with representations

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

$$\mathbf{Repr} : \mathrm{Ty}\ \Sigma\ \Gamma \to \mathrm{Ty}\ \Sigma\ \Gamma \tag{7}$$

along with an isomorphism

$$\mathbf{repr} : \mathrm{Tm}\ \Sigma\ \Gamma\ T \simeq \mathrm{Tm}\ \Sigma\ \Gamma\ (\mathbf{Repr}\ T) : \mathbf{unrepr}\,. \tag{8}$$

The type $\mathbf{Repr}\ T$ is the defined representation of the type T. The term **repr** takes a term of type T to its representation of type $\mathbf{Repr}\ T$, and **unrepr** undoes the effect of **repr**, treating a represented term as an inhabitant of its original type. These new constructs satisfy certain definitional equalities given in fig. 4 and fig. 5. While the former rules can be summarised as 'commute with Π and types represent themselves', the latter rules define how these constructs can be used to rewrite global definitions as their defined representations. From these rules, we can derive some additional admissible lemmas.

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

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

Proof. By the isomorphism in (8).

Importantly, this injectivity rule can only be applied if t and t' have the same type. Otherwise, if $\mathbf{Repr}\ \mathsf{Bool} = \mathbf{Repr}\ \mathsf{Bit}$, we could derive $\mathbf{repr}\ \mathsf{true} \equiv \mathbf{repr}\ \mathsf{zero}$ and then $\mathsf{true} \equiv \mathsf{zero}$. Of course, the injectivity lemmas can only be stated if the types are equal, but when it comes to implementation, a unification

$$\begin{aligned}
&\mathbf{Repr}\ (\Pi\ A\ B) \equiv \Pi\ A\ (\mathbf{Repr}\ B)\\
&\mathbf{Repr}\ \mathcal{U} \equiv \mathcal{U}\\
&\mathbf{Repr}\ (A[u]) \equiv (\mathbf{Repr}\ A)[u]
\end{aligned}$$

$$\begin{aligned}
&\mathbf{repr}\ (\lambda\ t) \equiv \lambda\ (\mathbf{repr}\ t)\\
&\mathbf{repr}\ (t\ @) \equiv (\mathbf{repr}\ t)\ @\\
&\mathbf{repr}\ (t[u]) \equiv (\mathbf{repr}\ t)[u]\\
&\mathbf{repr}\ (\mathbf{code}\ t) \equiv \mathbf{code}\ t
\end{aligned}
\qquad
\begin{aligned}
&\mathbf{unrepr}\ (\lambda\ t) \equiv \lambda\ (\mathbf{unrepr}\ t)\\
&\mathbf{unrepr}\ (t\ @) \equiv (\mathbf{unrepr}\ t)\ @\\
&\mathbf{unrepr}\ (t[u]) \equiv (\mathbf{unrepr}\ t)[u]\\
&\mathbf{unrepr}\ (\mathbf{code}\ t) \equiv \mathbf{code}\ t
\end{aligned}$$

Fig. 4. Basic definitional equalities for **Repr**, **repr** and **unrepr**, omitting the ones that hold by the afforementioned isomorphism. Notably **Repr** (**El** t), as well as **repr** x where x is a variable, do not reduce.

$$\textsc{Repr-Def-Id}\ \frac{\mathbf{repr}\ \mathsf{f}\ \Delta\ \mathbf{as}\ a \in \Sigma}{\Sigma \mid \Gamma \vdash \mathbf{repr}\ (\mathsf{f}\ \delta) \equiv a[\delta] : M[\delta]}
\qquad
\textsc{Repr-Ctor-Id}\ \frac{\mathbf{repr}\ \mathsf{c}\ \Pi\ \mathbf{as}\ \kappa \in \Sigma}{\Sigma \mid \Gamma \vdash \mathbf{repr}\ (\mathsf{c}\ \delta\ \pi) \equiv \kappa[\delta, \pi] : A[\delta, \xi[\pi]]}$$

$$\textsc{Repr-Data-Id}\ \frac{\mathbf{repr}\ \mathsf{D}\ \Delta\ \Xi\ \mathbf{as}\ A \in \Sigma}{\Sigma \mid \Gamma \vdash \mathbf{Repr}\ (\mathsf{D}\ \delta\ \psi) \equiv A[\delta, \psi]\ \mathrm{type}}$$

Fig. 5. Definitional equalities for **Repr** and **repr** relating to global items with defined representations.

algorithm must be type-aware to maintain soundness. While these term formers are injective, the type former **Repr** is not, since multiple distinct types can have the same representation. However, the weaker property holds that **Repr** is injective up to internal isomorphism.

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

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

Proof. The pair of maps $(\lambda(x : T).\ \mathbf{unrepr}\ (\mathbf{repr}\ x),\ \lambda(y : T').\ \mathbf{unrepr}\ (\mathbf{repr}\ y))$ forms an isomorphism, whose coherence proofs are both trivial.

Next, we define the rules for defining representations in λ_{REP}. Representations are a part of the signature, and can be defined for data types as well as global definitions. In the case of data types, the representation needs to be defined for the type itself, as well as for each of its constructors and the case eliminator.

For global definitions, the representation must be in the form of a term of the same type as the original definition, which is moreover. TODO: maybe we leave global definitions to rewrite rules?

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

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

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

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

Fig. 6. Rules for data representations in λ_{REP}.

3.3 Properties of λ_{REP}

- Decidability of equality
- Confluence of reduction
- Strong normalization

4 Translating from λ_{REP} to λ_{IND}

Now we define a translation step $\mathcal{R}$ from λ_{REP} to λ_{IND}, meant to be applied during the compilation process. We do this by translating contexts, substitutions, types, and terms in a mutual manner such that definitional equality is preserved. In other words, $\mathcal{R}$ is a setoid homomorphism if we consider definitional equality as a relation defining a setoid. The majority of the translation simply preserves the structure of λ_{REP}, so we focus on the translation of definitions into their representations. This is shown in fig. 7. By defining the translation in this way, we are awarded with some desirable properties by construction:

Lemma 3. *The translation $\mathcal{R}$ preserves typing, i.e.*

$$\frac{\Sigma \mid \Gamma \vdash t : T}{\mathcal{R}\Sigma \mid \mathcal{R}\Gamma \vdash \mathcal{R}t : \mathcal{R}T}$$

Lemma 4. *The translation $\mathcal{R}$ preserves definitional equality, i.e.*

$$\frac{\Sigma \mid \Gamma \vdash t \equiv t' : T}{\mathcal{R}\Sigma \mid \mathcal{R}\Gamma \vdash \mathcal{R}t \equiv \mathcal{R}t' : \mathcal{R}T}$$

$$\begin{aligned}
&\mathcal{R} : \mathrm{Ty}_{\text{REP}}\ \Sigma\ \Gamma \to \mathrm{Ty}_{\text{IND}}\ (\mathcal{R}\Sigma)\ (\mathcal{R}\Gamma)\\
&\mathcal{R}(\mathsf{D}\ \delta\ \psi) \mid \textbf{repr}\ \mathsf{D}\ \Delta\ \Xi\ \textbf{as}\ A \in \Sigma = \mathcal{R}(A[\delta, \psi])\\
&\mathcal{R}(\textbf{Repr}\ A) = \mathcal{R}A\\
&\mathcal{R}T = \text{recurse on } T
\end{aligned}$$

$$\begin{aligned}
&\mathcal{R} : \mathrm{Tm}_{\text{REP}}\ \Sigma\ \Gamma\ T \to \mathrm{Tm}_{\text{IND}}\ (\mathcal{R}\Sigma)\ (\mathcal{R}\Gamma)\ (\mathcal{R}T)\\
&\mathcal{R}(\mathsf{C}\ \delta\ \pi) \mid \textbf{repr}\ \mathsf{c}\ \Pi\ \textbf{as}\ \kappa \in \Sigma = \mathcal{R}(\kappa[\delta, \pi])\\
&\mathcal{R}(\mathsf{f}\ \delta) \mid \textbf{repr}\ \mathsf{f}\ \delta\ \textbf{as}\ a \in \Sigma = \mathcal{R}(a[\delta])\\
&\mathcal{R}(\textbf{repr}\ t) = \mathcal{R}t\\
&\mathcal{R}(\textbf{unrepr}\ t) = \mathcal{R}t\\
&\mathcal{R}t = \text{recurse on } t
\end{aligned}$$

Fig. 7. Translation from λ_{REP} to λ_{IND}.

TODO: show how the def.equalities are mapped.

The basic ones get mapped to reflexivity, the others are given as part of representations (validcase), and the rest are simply carried over.

– What other corollaries?

5 Primitives

6 Implementation

Superfluid is a programming language with dependent Π types, 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 types with defined representations. This is translated to JavaScript, where the core language has defined primitives that mirror JavaScript's core types and operations.

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, an inductively defined String

References

1. Abel, A., Danielsson, N.A., Eriksson, O.: A graded modal dependent type theory with a universe and erasure, formalized. Proc. ACM Program. Lang. **7**(ICFP), 920–954 (Aug 2023), https://doi.org/10.1145/3607862
2. Atkey, R.: Syntax and semantics of quantitative type theory. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 56–65. LICS '18, Association for Computing Machinery, New York, NY, USA (Jul 2018), https://doi.org/10.1145/3209108.3209189
3. 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
4. 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
5. Cockx, J., Devriese, D.: Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory. J. Funct. Prog. **28**(e12), e12 (Jan 2018), https://www.cambridge.org/core/services/aop-cambridge-core/content/view/E54D56DC3F5D5361CCDECA824030C38E/S095679681800014Xa.pdf/div-class-title-proof-relevant-unification-dependent-pattern-matching-with-only-the-axioms-of-you pdf
6. Diehl, L., Firsov, D., Stump, A.: Generic zero-cost reuse for dependent types. Proc. ACM Program. Lang. **2**(ICFP), 1–30 (Jul 2018), https://doi.org/10.1145/3236799
7. 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
8. 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
9. 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
10. 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
11. Wadler, P.: Views: a way for pattern matching to cohabit with data abstraction. In: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 307–313. POPL '87, Association for Computing

Machinery, New York, NY, USA (Oct 1987), https://doi.org/10.1145/41625.41653
12. Wadler, P.: Deforestation: transforming programs to eliminate trees. Theor. Comput. Sci. **73**(2), 231–248 (Jun 1990), https://www.sciencedirect.com/science/article/pii/030439759090147A