Practical Generic Programming over a Universe of Native Datatypes

LUCAS ESCOT, TU Delft, Netherlands
JESPER COCKX, TU Delft, Netherlands

Datatype-generic programming makes it possible to define a construction once and apply it to a large class of datatypes. It is often used to avoid code duplication in languages that encourage the definition of custom datatypes, in particular state-of-the-art dependently typed languages where one can have many variants of the same datatype with different type-level invariants. In addition to giving access to familiar programming constructions for free, datatype-generic programming in the dependently typed setting also allows for the construction of generic proofs. However, the current interfaces available for this purpose are needlessly hard to use or are limited in the range of datatypes they handle. In this paper, we describe the design of a library for safe and user-friendly datatype-generic programming in the Agda language. Generic constructions in our library are regular Agda functions over a broad universe of datatypes, yet they can be specialized to native Agda datatypes with a simple one-liner. Furthermore, we provide building blocks so that library designers can too define their own datatype-generic constructions.

CCS Concepts: • **Software and its engineering → Data types and structures**; • **Theory of computation → Type theory**.

Additional Key Words and Phrases: Generic programming, Dependent types

ACM Reference Format:
Lucas Escot and Jesper Cockx. 2022. Practical Generic Programming over a Universe of Native Datatypes. *Proc. ACM Program. Lang.* 6, ICFP, Article 113 (August 2022), 25 pages. https://doi.org/10.1145/3547644

1 INTRODUCTION

A generic program is a program or construction that can be applied for many different types, using a single implementation. In this paper, we focus on datatype-generic programming [Bird et al. 1996; Böhm and Berarducci 1985; Jay 1995], i.e. constructions that can be applied on a class of inductively defined datatypes. The motivation behind datatype-generic programming is to avoid code duplication: by implementing programs generically, a single implementation can be used for many different types. In addition, by using the same generic program to implement the same functionality for different types, we can expect the same consistent behaviour at each type, which would not be the case if it were implemented for each type by hand.

Datatype-generic programming is particularly important in dependently typed functional languages such as Agda [Agda Development Team 2021] or Idris [Brady 2021]. These languages encourage the definition of new inductive datatypes over reuse of existing types to encode invariants of the data and make impossible states unrepresentable. With this proliferation of new datatypes, some constructions and functions ought to be available for every inductive datatype, and it would

Authors' addresses: Lucas Escot, TU Delft, Delft, Netherlands, l.f.b.escot@tudelft.nl; Jesper Cockx, TU Delft, Delft, Netherlands, j.g.h.cockx@tudelft.nl.

2475-1421/2022/8-ART113
https://doi.org/10.1145/3547644

be unreasonable to expect programmers to have to define them again and again, for every new datatype introduced.

Language designers have proposed many solutions for automatically generating generic constructions on newly defined datatypes. Examples include both specialized 'deriving' mechanisms such as the **deriving** mechanism in Haskell [Jones 2003], **derive** traits in Rust [Klabnik and Nichols 2019], ppx_derive in OCaml [Jane Street Group 2018], and general frameworks for compile-time reflection such as template meta-programming in C++ [Abrahams and Gurtovoy 2004], Template Haskell [Sheard and Jones 2002], and elaborator reflection in Idris [Christiansen and Brady 2016] (which has also been adopted by Agda and Lean). However, all of them rely on the presence of certain primitives for compile-time code generation built into the compiler. Moreover, the generated programs are typically not guaranteed to be correct so they must pass through the typechecker before they can be used.

The approach that we follow in this paper instead is to work with a universe of *datatype encodings* [Chapman et al. 2010] that describe datatypes on which generic constructions can be implemented. In a dependently typed language, it is possible to define an interpretation function that maps these encodings to actual types and generic constructions to actual functions, without having to re-typecheck each individual instance. This provides an internalised representation of datatype declarations in the language itself, that can be freely inspected by the implementation of generic programs. Or, in the words of McBride [2013]:

> Once we have types that can depend computationally upon first class values, **metaprograms just become ordinary programs** manipulating and interpreting data which happen to stand for types and operations.

The main drawback of working with such a universe of encodings is that it forces the user of a generic function to work with the (interpretation of the) *encoded* datatype, rather than the datatype one would define by hand. As a consequence, the syntax is harder to understand and work with, and editor support might not work properly for encoded datatypes. Recent work on datatype-generic programming in Agda [effectfully 2020] combines the best of reflection and datatype encodings by defining a type that relates a 'native' Agda datatype to its encoding, and providing a shallow layer of reflection that automatically derives for a given Agda datatype both its encoding and the connection between the two. Once this is done, it is possible to define generic constructions that work on the encoding, and lift those constructions to work on the native Agda datatype using some reflection. This approach does come with some drawbacks: generic constructions do not have the expected computation rules, and the encoding is complex to work with.

In this paper, we present a new library[1] for datatype-generic programming in Agda that describes a specific class of parametrized and indexed datatypes and provides several generic constructions on them. From the point of view of the user, these generic constructions are safe, reflection-free Agda functions that can be specialized directly to a native Agda datatype with a simple one-liner. From the point of view of a library developer, implementing a new generic construction is a matter of writing a regular Agda function over our universe, which can use all of Agda's features and library functions directly without quoting.

Our main goal in the design of this library is to combine the best practices introduced by previous developments, and to fix the remaining issues standing in the way of their adoption in practice. Unlike effectfully [2020][2] we work with an explicit encoding of telescopes [Sijsling 2016], which means we rely even less on 'untrusted' reflection that might lead to unexpected failures while

[1]The latest version of the library is hosted at https://github.com/flupe/generics. Artefacts associated with this paper are publicly available [Escot and Cockx 2022]

[2]'effectfully' is a pseudonym, the real name of the author is not given.

generating the code. Our code can be used with the `--safe` flag, a desirable feature if we intend our library to be applicable in most situations. In addition, we greatly simplify the handling of telescopes and datatypes that are built from types at different universe levels by using Agda's built-in sort Setω (introduced in Agda 2.6.0).

Because we rely only on shallow conversion functions, we greatly simplified the implementation effort of effectfully [2020] all the while enabling generic constructions that reduce on open terms, making them practical inside proofs.

Contributions.

- We present an encoding for a class of parametrized and indexed inductive datatypes in Agda, including several Agda features such as datatypes at any given universe level, higher-order inductive arguments, and irrelevant and implicit function types (Sect. 3).
- We give a precise specification of when an actual Agda datatype corresponds to a type in our encoding (Sect. 3.3). We also provide a reflection macro that when given an Agda datatype automatically constructs its encoding together with the proof relating it to the actual datatype (Sect. 3.4).
- We define several generic constructions that work for all Agda datatypes that have an encoding, in particular the datatype eliminator, case analysis, fold, the 'no confusion' property [McBride et al. 2004], decidable equality, and a 'show' function. When applied to a specific datatype, the type of these generic constructions makes no reference to the underlying encoding, thus they can readily be used as-is in place of hand-written implementations (Sect. 2).
- We provide an interface for defining new generic constructions, which includes the possibility to generate constraints that restrict the class of datatypes a particular construction can be applied to (Sect. 4).

2 SHOWCASE

We start by introducing our library from the perspective of the user by showcasing the generic constructions it provides. In particular, we show how to instantiate the generic constructions in the library to three types: the type of natural numbers (Nat), a type of length-indexed vectors parametrized by the type of values (Vec A n), and the identity type (Id A x y) representing the propositional equality between x and y.

```
data Nat : Set where              data Vec (A : Set) : Nat → Set where
  zero : Nat                        []   : Vec A 0
  suc  : Nat → Nat                  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)

                 data Id (A : Set) (x : A) : A → Set where
                   refl : Id A x x
```

To derive a first-class encoding of the datatypes, we apply the deriveDesc macro:

```
natD : HasDesc Nat           vecD : HasDesc Vec           idD : HasDesc Id
natD = deriveDesc Nat        vecD = deriveDesc Vec        idD = deriveDesc Id
```

The types Nat, Vec, and Id now all have a *description*, i.e. an element of our generic universe of Agda datatypes, that is needed to use the datatype-generic constructions of the library. These descriptions are derived using Agda's reflection mechanism. This is the only place where reflection is used: once a description is derived, every datatype-generic construction provided can be applied safely. In other words, datatype-generic programs are safe, reflection-free Agda functions.

2.1 Generic Show

An easy but useful first example of a datatype-generic program is the show function for pretty-printing the data structure:

```
record Show {a} (A : Set a) : Set a where
  field show : A → String
open Show {{...}}
```

The syntax open Show {{...}} makes the show function available for use with instance search, similar to using a type class in Haskell.[3] Our library provides a function deriveShow that can be used to construct an instance of the Show class for a simple datatype such as Nat without parameters:

```
instance showNat : Show Nat
         showNat = deriveShow natD
```

On the other hand, because our length-indexed vectors Vec A n may contain elements of A, deriving an instance of Show (Vec A n) requires an instance of type Show A to be in scope:

```
instance showVec : ∀ {n} → {{ Show A }} → Show (Vec A n)
         showVec = deriveShow vecD
```

The generated show function works as expected, printing a structural representation of the datatype element. For example, show 2 returns the string `"suc (suc (zero))"`. For higher-order constructor arguments it simply prints the string `"?f"`.

2.2 Decidable Equality

Similarly, we provide a function deriveDecEq to derive decidable equality for a described datatype. A datatype is said to have decidable equality if for any two values, we can decide whether they are equal.

```
data Dec {a} (A : Set a) : Set a where
  yes :   A → Dec A
  no  : ¬ A → Dec A

record DecEq {a} (A : Set a) : Set a where
  field _≟_ : (x y : A) → Dec (x ≡ y)
```

We derive decidable equality for natural numbers and vectors as follows:

```
instance decEqNat : DecEq Nat
         decEqNat = deriveDecEq natD

instance decEqVec : ∀ {n} → {{ DecEq A }} → DecEq (Vec A n)
         decEqVec = deriveDecEq vecD
```

Writing out the proof of decidable equality by hand the naive way instead would take a number of cases quadratic in the number of constructors of the datatype.

One fundamental limitation of generic decidable equality comes from the fact that there is no general way to derive it for datatypes with higher-order inductive arguments. To reflect this fact, when applied to such a type the deriveDecEq function requires an instance constraint of the empty

[3]https://agda.readthedocs.io/en/v2.6.2/language/instance-arguments.html

dummy type HigherOrderArgumentsNotSupported.[4] For example, consider the definition of the W type:

```
data W (A : Set) (B : A → Set) : Set where
  sup : (x : A) (f : B x → W A B) → W A B

WD : HasDesc W
WD = deriveDesc W
```

When trying to use deriveDecEq on WD, we get the following error:

```
No instance of type HigherOrderArgumentsNotSupported WD was found in
scope
```

2.3 Generic Induction Principle

A basic construction to reason about properties of inductive structures is the induction principle, also known as the eliminator. While Coq users are used to getting an induction principle for free when they introduce a datatype, Agda users have to implement one by hand, with the help of built-in dependent pattern-matching. For example, the eliminator for Nat is defined by hand as follows:

```
elimNat : ∀ {c} (P : Nat → Set c)
          → P zero → (∀ n → P n → P (suc n)) → (n : Nat) → P n
elimNat P Pz Ps zero    = Pz
elimNat P Pz Ps (suc n) = Ps n (elimNat P Pz Ps n)
```

Using our library, you can get the definition of the eliminator for free:

```
elimNat′ : ∀ {c} (P : Nat → Set c)
           → P zero → (∀ {n} → P n → P (suc n)) → (n : Nat) → P n
elimNat′ = deriveElim natD
```

The shape of the derived eliminator is identical to the hand-written version, and it can be derived for all described datatypes. Deriving it for the identity type gives us the expected J eliminator.

```
elimId : ∀ {A c x} (P : ∀ {y} → Id A x y → Set c)
         → P refl
         → ∀ {y} (e : Id A x y) → P e
elimId = deriveElim idD
```

Datatypes with higher-order inductive arguments are supported as well:

```
elimW : ∀ {A B c} (P : W A B → Set c)
        → (∀ x {f} → (∀ y → P (f y)) → P (sup x f))
        → ∀ w → P w
elimW = deriveElim WD
```

Note that type signatures for our generic constructions are only written above and in the following examples to convince the reader that they have the expected type, but they are not mandatory. The library computes it along with the definition itself. However, giving the type anyway is good practice as it ensures that the generated type corresponds to the type we expected.

[4]The reason for using a custom empty type instead of the standard one is to inform the user of the reason why the construction could not be derived.

An added benefit of our datatype-generic induction principle and all the other constructions is that they reduce on open terms, meaning their computational behaviour holds definitionally. In contrast to previous implementations [effectfully 2020] of datatype-generic constructions in the dependently-typed setting where encodings are not first-class, this means anything defined using some of our constructions can be reasoned about easily, and will behave as expected even when applied to abstract terms.

```
elimNat′-suc : ∀ {c} (P : Nat → Set c) (H0 : P 0)
                     (HS : ∀ {n} → P n → P (suc n))
  → ∀ n → elimNat′ P H0 HS (suc n) ≡ HS (elimNat′ P H0 HS n)
elimNat′-suc P H x f = refl

elimId-refl : ∀ {A x c} (P : ∀ {y} → Id A x y → Set c) (p : P refl)
            → elimId P p refl ≡ p
elimId-refl P p = refl
```

We also provide special cases of the induction principle, such as *case analysis.*

```
open import Generics.Constructions.Case

caseVec : ∀ {A c} (P : ∀ {n} → Vec A n → Set c)
          → P [] → (∀ {n} x (xs : Vec A n) → P (x :: xs))
          → ∀ {n} (xs : Vec A n) → P xs
caseVec = deriveCase vecD
```

Likewise, we implement a datatype-generic *fold.*

```
open import Generics.Constructions.Fold

foldNat : ∀ {c} {X : Set c} → X → (X → X) → Nat → X
foldNat = deriveFold natD

add : Nat → Nat → Nat
add x = foldNat x suc
```

Again, we demonstrate that our datatype-generic fold reduces properly on values that are not fully evaluated.

```
add0 : ∀ x → add x 0 ≡ x
add0 x = refl

addS : ∀ x y → add x (suc y) ≡ suc (add x y)
addS x y = refl
```

2.4 No Confusion

McBride et al. [2004] present a recipe for constructing proofs that constructors of datatypes are injective and disjoint, called the *No Confusion* principle. We provide a generic construction noConfusion of this property in our library. We also implement the converse noConfusion′ which is exactly the congruence of equality through datatype constructors. At the moment we do not yet provide automatic construction of injectivity that follows from this principle, so some manual work is still required to extract its proof. But this principle is sufficient to prove disjointness between any two distinct constructors. For example, here we show how to extract the proof of injectivity of the suc constructor from the generic noConfusion construction, and some other proofs:

```
open import Generics.Constructions.NoConfusion

- noConfusion  natD : {x y : Nat} → x ≡ y → NoConfusion natD x y
- noConfusion′ natD : {x y : Nat} → NoConfusion natD x y → x ≡ y

_ : ∀ x → NoConfusion natD (suc x) zero ≡ ⊥
_ = λ x → refl

_ : ∀ x y → NoConfusion natD (suc x) (suc y) ≡ (x ≡ y × ⊤)
_ = λ x y → refl

suc≢zero : ∀ {x} → suc x ≢ zero
suc≢zero = noConfusion natD

zero≢suc : ∀ {x} → zero ≢ suc x
zero≢suc = noConfusion natD

cong-suc : ∀ {x y} → x ≡ y → suc x ≡ suc y
cong-suc = noConfusion′ natD ∘ (_, tt)

inj-suc : ∀ {x y} → suc x ≡ suc y → x ≡ y
inj-suc = proj₁ ∘ noConfusion natD
```

This datatype-generic construction really shines for quickly proving that constructors are disjoint. Without it, a quadratic number of definitions would be needed for complete coverage. Here, a single definition can be used for all the proofs of disjointness.

The shape of NoConfusion for more involved datatypes becomes quickly impractical when trying to derive either injectivity or congruence, as using the homogeneous propositional equality forces us to do more equality substitution than necessary. For this reason, we provide a datatype-generic congruence with a nicer interface that does not rely on NoConfusion, with the only downside that one has to use the *heterogeneous* equality from the standard library.

```
open import Relation.Binary.HeterogeneousEquality
open import Generics.Constructions.Cong

node-cons : {A : Set} {n m : Nat}            → n  ≅ m
          → {x y : A}                        → x  ≅ y
          → {xs : Vec A n} {ys : Vec A m}    → xs ≅ ys
          → x :: xs ≅ y :: ys
node-cons = deriveCong vecD (suc zero)
```

3 ENCODING AGDA DATATYPES IN AGDA

Now that we have seen what the library can do, we show how datatypes are encoded as first class values so that datatype-generic programs can be implemented. Instead of relying on Agda's reflection API, our goal is to find a proper encoding for inspecting datatype definitions from inside Agda itself, thus providing a safe environment for developing datatype-generic constructions. In Agda, the general shape of the definition of a dataype D is the following:[5]

$$\begin{array}{l} \mathsf{data}\ \mathsf{D}\ (x_1 : P_1)\cdots(x_k : P_k) : (y_1 : Q_1) \to \cdots \to (y_l : Q_l) \to \mathsf{Set}\ \ell\ \mathsf{where} \\ \quad \mathsf{c}_1 \quad : \quad A_1 \\ \quad \cdots \\ \quad \mathsf{c}_n \quad : \quad A_n \end{array}$$

This definition introduces a new type family D, parametrized by the telescope of *parameters* $(x_1 : P_1)\dots(x_k : P_k)$, and indexed by the telescope of *indices* $(y_1 : Q_1)\dots(y_l : Q_l)$. The type family D itself lives in the universe Set ℓ. The n constructors $(\mathsf{c}_1, \dots, \mathsf{c}_n)$ are the *introduction rules* of type family D. Their types $(A_1, \dots, A_n)$ must be function types of the form

$$(z_1 : B_1) \to \cdots \to (z_m : B_m) \to \mathsf{D}\ x_1\ \cdots\ x_k\ t_1\ \cdots\ t_l$$

where $z_1, \cdots, z_m$ are the *arguments* of the constructor. Furthermore, each argument $z_i : B_i$ is either *non-inductive*, in which case B_i does not mention D, or *inductive*, and B_i must have the following shape:

$$(w_1 : C_1) \to \cdots \to \left(w_p : C_p\right) \to \mathsf{D}\ x'_1\ \cdots\ x'_k\ t'_1\ \cdots\ t'_l$$

This restriction is known as *strict positivity*, and (together with the termination check) rules out non-terminating definitions, thus ensuring consistency of the theory. For inductive arguments, Agda allows the parameters $\left(x'_1, \cdots, x'_k\right)$ to be different than the parameters of the constructor $(x_1, \cdots, x_k)$. However, in this work we consider them to be uniform, which simplifies the definition of the datatype-generic constructions. Usually non-uniform parameters are instead considered as indices,[6] so this is not a real restriction to the class of datatypes we support. Agda makes it possible to write mutually-defined datatypes, but this is beside the scope of this encoding. Likewise, we ignore definitions by induction-recursion.

Our goal is thus to find a proper first-class encoding for the above parametrized and indexed datatypes. We build upon existing encodings [effectfully 2020; Sijsling 2016], making three important improvements:

- We use an explicit encoding of telescopes for parameters and indices [Sijsling 2016]. This enables us to provide clean user-facing interfaces without having to rely on reflection and unsafe manipulation of terms. With explicit telescopes, our descriptions for datatypes are fully first-order, as can be seen for the π constructor of the ConDesc type (see Sect. 3.2). One argument describes the type of the argument in the constructor, depending on the value currently in scope. The second argument is the remaining description, with a new value available in scope, whose type is the one given by the first argument.
- Since Agda 2.6, it is possible to define datatypes living in Setω, and still be able to pattern match on them. We make use of this new feature to avoid having to determine the precise universe levels for telescopes and datatypes in advance, greatly simplifying the treatment of the different universe levels that appear in the definition of a datatype.

[5]https://agda.readthedocs.io/en/v2.6.2/language/data-types.html
[6]This might require increasing the universe level of the datatype overall, unless a forcing analysis can determine the index to be forced. This is out of the scope of our current work.

- Furthermore, by exclusively working with shallow conversion functions and accessibility predicates, we are able to implement datatype-generic constructions that reduce even on open terms, making them applicable to situations where reasoning about abstract values is an absolute necessity, such as inside proof terms.

3.1 Encoding Telescopes

A *telescope* is a list of variable bindings together with their types, each of which can depend on all the previously bound variables. Telescopes appear in the definition of an inductive type for the parameters and indices as well as the arguments to each constructor and each inductive constructor argument. The first crucial step in order to reason about inductive families is thus to have a first-class model for their parameters and indices. Taking inspiration from the work of Sijsling [2016], we define the following datatype of telescope descriptions by induction-recursion.

```
data Telescope {ℓ} (A : Set ℓ) : Setω
levelOfTel : Telescope A → Level
⟦_⟧tel : (T : Telescope A) → A → Set (levelOfTel T)
```

A telescope is either empty, or it is an extension of an existing telescope with an additional variable, by providing a type family S that produces a type for any given instantiation of the left telescope T:

```
data Telescope A where
  ϵ   : Telescope A
  _⊢_ : (T : Telescope A) {ℓ′ : Level} (S : Σ A ⟦ T ⟧tel → Set ℓ′) → Telescope A
```

Contrary to Sijsling [2016], we define telescope descriptions inside Setω, and compute the upper level using levelOfTel, after the fact. This makes it possible to describe telescopes containing sets living at many different levels.

```
levelOfTel ϵ = lzero
levelOfTel (_⊢_ T {ℓ} _) = ℓ ⊔ levelOfTel T
```

Mutually with the definition of telescopes we define their interpretation as a nested sequence of dependent products.

```
⟦ ϵ     ⟧tel x = ⊤
⟦ T ⊢ S ⟧tel x = Σ (⟦ T ⟧tel x) (S ∘ (x ,_))
```

We define an *instantiation* of a telescope T at x to be an inhabitant of ⟦ T ⟧tel x.

The novelty is that we parametrize telescope descriptions by some type A so that each type in the telescope can depend on a foreign value $x : A$. This enables us to describe parameters and indices of an inductive family using two separate telescope descriptions, with the types of the indices possibly depending on the parameters.

Concretely, parameters of a datatype are described by inhabitants of the type Telescope ⊤, and given a parameter telescope P, indices are described by the inhabitants of a *telescope extension* ExTele P, where ExTele is defined as follows:

```
ExTele : Telescope ⊤ → Setω
ExTele P = Telescope (⟦ P ⟧tel tt)
```

This encoding enables us to describe inductive families where parameters appear in the types of the indices, such as the identity type. The alternative definition ExTele P := ⟦ P ⟧tel tt → Telescope ⊤ would also allow this, but it is in fact too general, as it allows not just the types of the indices to depend on the parameters but also the structure of the telescope itself (e.g. the number of parameters).

As an example, here is how we would describe the parameters (A : Set) and indices (n : Nat) of the Vec inductive family of length-indexed lists:

```
P : Telescope ⊤                    I : ExTele P
P = ϵ ⊢ const Set                  I = ϵ ⊢ const Nat
```

We define further helper functions to interpret both parameter and index telescopes as a single dependent product with ⟦_,_⟧xtel, and even interpret a third telescope depending on the same parameters using ⟦_,_&_⟧xtel. This last function will be useful in the encoding of datatypes, where we have two telescopes relying on parameters: indices and values in scope of constructors.

```
⟦_,_⟧xtel : ∀ P (I : ExTele P) → Set _
⟦ P , I ⟧xtel = Σ (⟦ P ⟧tel tt) ⟦ I ⟧tel

⟦_,_&_⟧xtel : ∀ P (V I : ExTele P) → Set _
⟦ P , V & I ⟧xtel = Σ (⟦ P ⟧tel tt) λ p → ⟦ V ⟧tel p × ⟦ I ⟧tel p
```

We thus provide an instantiation for parameters and indices of Vec in a single step:

```
pi : ⟦ P , I ⟧xtel
pi = (tt , Nat) , (tt , 3)
```

Instead of interpreting a telescope as an iterated sigma type, we can also interpret it as an iterated (curried) Π-type:

```
Curried′ : ∀ T → (⟦ T ⟧tel x → Set l) → Set (l ⊔ levelOfTel T)
Curried′ ϵ       Pr = Pr tt
Curried′ (T ⊢ S) Pr = Curried′ T λ t → (s : S (_ , t)) → Pr (t , s)

Curried : ∀ P I {ℓ} → (⟦ P , I ⟧xtel → Set ℓ) → Set (ℓ ⊔ levelOfTel I ⊔ levelOfTel P)
Curried P I {ℓ} Pr = Curried′ P λ p → Curried′ I λ i → Pr (p , i)
```

The function uncurry converts from curried functions on the interpretation of a telescope to their uncurried version:

```
uncurry′ : ∀ T (P : ⟦ T ⟧tel x → Set l) → Curried′ T P → ∀ y → P y
uncurry′ ϵ       P B tt       = B
uncurry′ (T ⊢ S) P B (tx , gx) = uncurry′ T (λ p → (s : S (_ , p)) → P (p , s)) B tx _

uncurry : ∀ P I {ℓ Pr} → Curried P I {ℓ} Pr → ∀ pi → Pr pi
uncurry P I C (p , i) = uncurry′ I _ (uncurry′ P _ C p) i
```

We also define a type Pred that is identical to Curried except it uses implicit instead of explicit Π-types. We use it to compute the type of predicates over some datatype, whose quantification over parameters and indices of the latter is left implicit. Given telescope descriptions for both parameters and indices, we can compute the type of a family parametrized and indexed by those telescopes.

```
Indexed : ∀ P (I : ExTele P) ℓ → Set _
Indexed P I ℓ = Curried P I (const (Set ℓ))
```

Going back to our example for the Vec inductive family, we can check that indeed Vec has the type Indexed P I lzero, and it is possible to provide all parameters and indices at once through uncurrying:

```
S = uncurry P I Vec ((tt , Nat) , (tt , 3))
-- S normalises to 'Vec Nat 3'
```

What we presented here is a simplified version of the telescope encoding used in our library. Most notably, our actual encoding takes into account whether parameters should be *visible*, *hidden* or *instance arguments*, and whether they are *relevant* or *irrelevant*. We stick to the simplified presentation for the rest of this paper, but the full version can be found in the code accompanying the paper.

3.2 Datatype Descriptions

Now that we have an encoding of telescopes for parameters and indices, we turn our attention to the encoding of datatypes, following the general overall strategy of Chapman et al. [2010]. We first introduce a universe of descriptions for datatype constructors. Next, we come to descriptions of datatypes. Finally, we give an interpretation of datatype descriptions as a functor on indexed families. We diverge from previous work in that we do not construct the least fixed-point of the interpretation μ and define generic constructions over it, but rather provide an interface for proving how an existing type implements the description.

Describing constructors and inductive constructor arguments. We define a universe of constructor descriptions ConDesc V, parametrised by a telescope V of values in scope:

```
data ConDesc (V : ExTele P) : Setω where
  var  : (((p , v) : ⟦ P , V ⟧xtel) → ⟦ I ⟧tel p)              → ConDesc V
  π    : ∀ {ℓ} (S : ⟦ P , V ⟧xtel → Set ℓ) (C : ConDesc (V ⊢ S)) → ConDesc V
  _⊗_ : (A B : ConDesc V)                                       → ConDesc V
```

This description characterizes the *shape* of a given datatype constructor: how many arguments it contains, the order of these arguments, and which of those are inductive arguments. In addition, this encoding conveys how the final indices of the constructor are computed from previous arguments, where the types of arguments can depend on values previously bound in the constructor.

- var denotes the empty constructor description, and holds a function to compute indices of type ⟦ I ⟧tel p, given an instantiation of parameters p : ⟦ P ⟧tel tt and values in scope v : ⟦ V ⟧tel p.
- π describes a constructor that expects an argument of type S (p , v), in scope for the types of the subsequent constructor arguments in C. Note that the type S (p , v) depends on values v already bound. Because we define ConDesc in Setω, S (p , v) can live at any universe level ℓ.
- _⊗_ describes a constructor with an inductive argument A and the remainder of the constructor B. The novel idea from effectfully [2016b] is to use ConDesc both to describe the shape of constructors and the shape of inductive arguments.

As noted in Sect. 3, Agda allows higher-order inductive arguments in datatype constructors, therefore an encoding of datatypes should account for the kind of inductive arguments appearing inside each constructor. Under the interpretation of ConDesc as the representation of an inductive constructor argument, var represents a first-order inductive argument with specified indices, while π characterizes higher-order inductive arguments. In this case, _⊗_ is used to describe a pair of inductive arguments.

We combine this encoding with explicit handling of values in scope through the telescope V [Sijsling 2016], so that descriptions themselves are fully first-order. On the one hand, this allows Agda to accept the definition of ConDesc without any unsafe pragmas, on the other hand it enables us to precisely describe Agda datatypes, and nothing more. As was the case for telescopes, our actual encoding stores more information about arguments, their *relevance* and *visibility*, which we omit here.

As an example, we describe some constructors of datatypes from Section 2.

Constructor	Description
zero : Nat	var (const tt)
suc : Nat → Nat	var (const tt) ⊗ var (const tt)
[] : Vec A 0	var (const (tt , 0))
refl : Id A x x	var (λ ((_ , x) , _) → tt , x)

Describing datatypes. Once the shape of every constructor of a datatype has been described, we can give a description for the full datatype as a list of descriptions for each of its constructors:

```
data DataDesc P (I : ExTele P) : Nat → Setω where
  []   : DataDesc P I 0
  _::_ : ∀ {n} (C : ConDesc P I ϵ) (D : DataDesc P I n) → DataDesc P I (suc n)

lookupCon : ∀ {P I n} → DataDesc P I n → Fin n → ConDesc P I ϵ
lookupCon (C :: D) zero = C
lookupCon (C :: D) (suc k) = lookupCon D k
```

This is essentially the type Vec of length-indexed lists from standard library, with the notable difference that the universe of elements lives in Setω. It is in fact the main drawback of using Setω : most if not all of the tools of the standard library are no longer applicable and need to be duplicated.

Our encoding, much like effectfully [2020], can describe parametrized and indexed datatypes defined at any given level. It supports higher-order inductive arguments, but nested datatypes are not allowed. We decided to forbid inductive arguments with *different* parameters as we found it made generic constructions such as the induction principle less practical to use. Mutually-defined datatypes and datatypes defined by induction-recursion are not supported.

Interpreting datatype descriptions. We define the interpretation of a datatype description D as a functor on ⟦ P , I ⟧xtel-indexed sets. That is, given a type family X : ⟦ P , I ⟧xtel → Set, the interpretation ⟦ D ⟧Data X results in a new type family indexed by ⟦ P , I ⟧xtel that is an iterated dependent pair containing the arguments of constructors, where recursive arguments are interpreted as values of type X.

```
⟦_⟧Data : ∀ {P I ℓ n} (D : DataDesc P I n) → (⟦ P , I ⟧xtel → Set ℓ)
                                          → (⟦ P , I ⟧xtel → Setω )
```

To give this interpretation, we first need the action of a constructor description C on an indexed family X, noted ⟦ C ⟧Con X, whose shape mimics the one of the constructor, but where inductive arguments are interpreted as values of X. The interpretation of inductive arguments is in turn taken care of with the interpretation of an inductive argument description C on X, noted ⟦ C ⟧IndArg X. We omit the definition of levelIndArg and levelCon that compute the level at which the interpretation of descriptions should live.

```
levelIndArg levelCon : ∀ {V} → ConDesc P I V → Level → Level

⟦_⟧Con : ∀ {V ℓ} (C : ConDesc P I V) → (⟦ P , I       ⟧xtel → Set ℓ)
                                     → (⟦ P , V & I ⟧xtel → Set (levelCon C ℓ))

⟦_⟧IndArg : ∀ {V ℓ} (C : ConDesc P I V) → (⟦ P , I ⟧xtel → Set ℓ)
                                        → (⟦ P , V ⟧xtel → Set (levelIndArg C ℓ))
```

As noted before, we use the same type to describe both constructors and inductive constructor arguments, so the only difference lies in which interpretation function we use. These two interpretations of descriptions are what effectfully [2016b] refers to as *computational* and *propositional* interpretations. Notice how, in contrast to the implementation of effectfully [2020], we do not try to keep the resulting family at the same level as the input family. Because we don't parametrize descriptions by the upper bound level, and compute the level of interpretations from the description, our implementation of the interpretation is much closer to the one of Chapman et al. [2010].

Descriptions for inductive arguments are interpreted as such:

```
⟦ var f   ⟧IndArg X (p , v)        = X (p , f (p , v))
⟦ π S C   ⟧IndArg X pv@(p , v) = (s : S pv) → ⟦ C ⟧IndArg X (p , v , s)
⟦ A ⊗ B ⟧IndArg X pv             = ⟦ A ⟧IndArg X pv × ⟦ B ⟧IndArg X pv
```

We see here that indeed, ⟦ var f ⟧Con X pv reduces to $X\ (p , f\ (p , v))$, so the description of first-order inductive arguments is interpreted as type X with the proper indices. And ⟦ π S C ⟧Con X pv reduces to $(s : S\ pv) \rightarrow \cdots \rightarrow X \cdots$, a function type ending with X with the proper indices. These are precisely the types of inductive arguments allowed by Agda at the beginning of Sect. 3.

Constructor descriptions, in turn, are interpreted as such:

```
⟦ var x   ⟧Con X (p , v , i)          = i ≡ x (p , v)
⟦ π S C   ⟧Con X pvi@(p , v , i)  = Σ[ s ∈ S (p , v) ] ⟦ C ⟧Con X (p , (v , s) , i)
⟦ A ⊗ B ⟧Con X pvi@(p , v , _) = ⟦ A ⟧IndArg X (p , v) × ⟦ B ⟧Con X pvi
```

The resulting indexed family is an iterated dependent pair, ending with a witness that the indices computed by the constructor are consistent with the indices of the family. Coming back to natural numbers, we show in this table the interpretation of each constructor:

Constructor (C)	⟦ C ⟧Con X (tt , tt , tt)
zero : Nat	tt ≡ tt
suc : Nat → Nat	X (tt , tt) × (tt ≡ tt)

To give an element in the interpretation of zero, no argument is required apart from a proof that tt ≡ tt, trivially refl. To give an element in the interpretation of suc, an inductive argument is required, of type X (tt , tt), and again, a trivial proof.

The interpretation of a datatype description is simply a dependent pair composed of the choice of constructor, along with an element of the interpretation of said constructor.

```
record ⟦_⟧Data {n ℓ} (D : DataDesc P I n) (X : ⟦ P , I ⟧xtel → Set ℓ)
                     (pi : ⟦ P , I ⟧xtel) : Setω where
  constructor _,_
  field k    : Fin n
        val : ⟦ lookupCon D k ⟧Con X (proj₁ pi , tt , proj₂ pi)
```

We have to place this interpretation in Setω since it contains the interpretation of a constructor, which lives in different universes depending on the choice of constructor. Because effectfully [2020] follows the approach of Chapman et al. [2010] by taking the least fixed point μ of this interpretation function, they had to make sure that the interpretation preserves the level of the input family, which is the main reason for their complex and verbose encoding. Thankfully, as we no longer need to rely μ, we are free to simplify the interpretation by embracing Setω.

3.3 Relating to Native Agda Datatypes

So far, we have seen how to *describe* inductive families through the means of datatype descriptions and telescope encodings, and have shown how to interpret those as a type family that behaves as would a datatype introduced via a standard data declaration.

However, rather than define the least-fixed fixpoint $\mu\ D$ of the interpretation ⟦ D ⟧Data as per [Chapman et al. 2010], we choose to only work with the interpretation of D on the concrete underlying Agda inductive family for which D is a description. For this purpose, we define the following record type HasDesc A, which helps bridge the gap between a concrete Agda type family A and its encoding:

```
record HasDesc {P I} {ℓ} (A : Indexed P I ℓ) : Setω where

  A′ : ⟦ P , I ⟧xtel → Set ℓ
  A′ = uncurry P I A

  field
```

The first step required to attach a description to A is, unsurprisingly, to provide an actual description D, using the same telescopes P and I.

```
    {n} : Nat
    D   : DataDesc P I n
```

We additionally require all constructors to be named, so that this information can be used in our datatype-generic Show implementation.

```
    names : Vec String n
```

Then, we ask for shallow conversion functions going from A′ to ⟦ D ⟧Data A′ and back, along with proofs that they are inverse of one another.

```
    constr : ∀ {pi} → ⟦ D ⟧Data A′ pi → A′ pi
    split  : ∀ {pi} → A′ pi → ⟦ D ⟧Data A′ pi

    constr∘split : ∀ {pi} (x : A′ pi)             → constr (split x) ≡  x
    split∘constr : ∀ {pi} (x : ⟦ D ⟧Data A′ pi) → split (constr x) ≡ω x
```

This differs from [effectfully 2020], [Sijsling 2016] and other attempts in that they try to derive from : A′ pi → μ D pi and to : μ D pi → A′ pi that *fully* convert any value to and back from an internal representation. Here, we take the approach of using more atomic operations, constr only unfolding a value *once* to retrieve the outermost constructor, with the underlying inductive occurences still referring to the original type family. Crucially, shallow conversion functions allow us to make progress on open terms.

But an ingredient is missing. Most datatype-generic constructions rely on recursing on an inductive element, stressing the need for recursively applying these conversion functions. Because the inductive arguments in the return value of split are elements of the underlying type family, we need to call split on them to inspect which constructor they are made of, and so on. However, since the inductive arguments in split x are not syntactically smaller than x, Agda's termination checker will refuse any recursive definition implemented as such.

Our way out is to introduce an *accessibility* predicate to make use of the fact that A ought to be well-founded. We begin by introducing AllData, AllIndArg and AllCon such that AllData $Pr\ D\ d$ states that $Pr : X\ \ (p\ ,\ i) \to$ Set ℓ holds for every $x : X\ \ (p\ ,\ i)$ in d.

```
AllIndArg : ∀ {V p} {X : ⟦ P , I ⟧xtel → Set ℓ}
            (Pr : ∀ {i} → X (p , i) → Setω) (C : ConDesc P I V)
          → ∀ {v} → ⟦ C ⟧IndArg X (p , v) → Setω
AllIndArg Pr (var _) x = Pr x
AllIndArg Pr (π S C) x = (s : S _) → AllIndArg Pr C (x s)
AllIndArg Pr (A ⊗ B) (xa , xb) = AllIndArg Pr A xa ω×ω AllIndArg Pr B xb

AllCon : ∀ {V p} {X : ⟦ P , I ⟧xtel → Set ℓ}
         (Pr : ∀ {i} → X (p , i) → Setω) (C : ConDesc P I V)
       → ∀ {v i} → ⟦ C ⟧Con X (p , v , i) → Setω
AllCon Pr (var _) x = ⊤ω
AllCon Pr (π _ C) (_ , x) = AllCon Pr C x
AllCon Pr (A ⊗ B) (xa , xb) = AllIndArg Pr A xa ω×ω AllCon Pr B xb

AllData : ∀ {p n} {X : ⟦ P , I ⟧xtel → Set ℓ}
          (Pr : ∀ {i} → X (p , i) → Setω)
          (D : DataDesc P I n)
        → ∀ {i} ((k , x) : ⟦ D ⟧Data X (p , i)) → Setω
AllData Pr D (k , x) = AllCon Pr (lookupCon D k) x
```

Constructor (C)	x : ⟦ C ⟧Con X (tt , tt , tt)	AllCon Pr C x
zero : Nat	refl : tt ≡ tt	⊤ω
suc : Nat → Nat	(n , refl) : X (tt , tt) × (tt ≡ tt)	(Pr n) ω×ω ⊤ω

Then, we introduce the accessibility predicate Acc x stating that x is *accessible* if all elements of split x are accessible.

```
data Acc {pi} (x : A′ pi) : Setω where
  acc : AllData Acc D (split x) → Acc x
```

Finally, we simply add as a requirement in the HasDesc record a proof that the inductive family being described is well-founded, that is, any x is accessible.

```
field wf : ∀ {pi} (x : A′ pi) → Acc x
```

Thus, any datatype-generic construction needing to do recursion on some x can first retrieve a proof that x is accessible with wf x, and then recurse on this witness rather than x.

This is all that is required for us to implement practical datatype-generic constructions, as we demonstrate in Sect. 4.

3.4 Deriving the Encoding

We showcase here the final encoding of natural numbers:

```
natD : HasDesc {P = ϵ} {I = ϵ} Nat
natD .n = 2
natD .D = var (const tt) :: (var (const tt) ⊗ var (const tt)) :: []
natD .names = "zero" :: "suc" :: []
natD .constr (zero           , refl) = 0
natD .constr (suc zero , n  , refl) = suc n
natD .split 0 = (Fin.zero , refl)
natD .split (suc n) = (Fin.suc Fin.zero , n , refl)
natD .constr∘split 0          = refl
natD .constr∘split (suc x) = refl
natD .split∘constr (zero           , refl) = refl
natD .split∘constr (suc zero , n , refl) = refl
natD .wf Nat.zero = Accessibility.acc ttω
natD .wf (suc x) = Accessibility.acc (natD .wf x , ttω)
```

It should become clear that although every field of the record is very easy to fill in, we cannot reasonably expect users to provide their own instances of HasDesc A for every family A they want to derive generic constructions for. Therefore, we need a way to automate this process. This is the purpose of the deriveDesc macro from our library, that we showcased in Sect. 2. As the details of the implementation of this macro using the reflection API are rather technical but ultimately unsurprising, we omit its definition here. The interested reader can refer to the supplementary material for the full implementation.

4 DEFINING DATATYPE-GENERIC CONSTRUCTIONS

Now that we have settled on a suitable encoding, it is time to build datatype-generic constructions. In this section, we show in detail how two of our generic constructions are defined: the generic show function and the generic induction principle. To ease the implementation of these generic constructions, we explain how our library introduces a generally usable type Helpers to specify the constraints that are required for the construction. To define a generic construction, we then follow the following recipe:

- The first step is to refine the Helpers datatype to specify the type of additional information and instances Agda should look for.
- Then, from a module parametrized by an instance of appropriate helpers, we define the desired construction on any value for which we have an accessibility witness, recursing on the latter.
- Finally, we define the user-facing function by calling the previous construction while using the witness of well-foundedness from the encoding.

4.1 Generic Helpers

In many situations, it is not sufficient to simply know the shape of datatypes, and more information about types appearing inside datatype constructors is required. For example, recall how an instance of Show A was needed to derive an instance of Show (Vec A n). More generally, to derive an instance of Show D for some inductive datatype D, we need an instance of Show A for each type A appearing in the constructors of D. Since the same is required for deriving DecEq D, we provide tools for requesting instances of *Arg A* for any *A* appearing in constructors and any given type family *Arg*.

That is, given the family *Arg* : ∀ {ℓ} → Set ℓ → Set (*levelArg l*), of which we want instances, and the parameter *Ind* : ∀ {*V*} (*C* : ConDesc *P V I ℓ*) → Setω (whose role we explain below), we define the inductive datatype ConHelper *p C*, indexed by the constructor description it is providing further information for:

```
data ConHelper p {V} : ConDesc P I V → Setω where
  instance
```

For an empty constructor, no extra information is required.

```
    var : ∀ {f} → ConHelper p (var f)
```

When a constructor expects an argument of type *S* (*p* , *v*), its associated helper is made out of an instance of *Arg* (*S* (*p* , *v*)) and a helper for the remainder of the constructor.

```
    pi  : {S : ⟦ P , V ⟧xtel → Set ℓ} {C : ConDesc P I (V ⊢ S)}
        → ⦃ ∀ {v} → Arg (S (p , v)) ⦄ → ⦃ ConHelper p C ⦄
        → ConHelper p (π S C)
```

Finally, when a constructor has an inductive argument described by *A*, its helper is composed of an instance of *Ind A* and a helper for the remaining part.

```
    prod : {A B : ConDesc P I V}
         → ⦃ Ind A ⦄ → ⦃ ConHelper p B ⦄ → ConHelper p (A ⊗ B)
```

The purpose of *Ind* is to control how the presence of inductive arguments must affect the automated helper resolution. By choosing *Ind* ≡ λ _ → ⊤ω as we do for our datatype-generic Show implementation, all inductive arguments are permitted. Were we to define *Ind* ≡ λ _ → ⊥ω, the instance search would fail for any datatype containing inductive arguments. In our datatype-generic implementation of DecEq, we make it so that *Ind* only allows first-order inductive arguments by choosing *Ind* ≡ OnlyFO defined as such: [7]

```
OnlyFO : ∀ {V} (C : ConDesc P I V) → Setω
OnlyFO (var x)  = ⊤ω
OnlyFO (π S C)  = HigherOrderArgumentsNotSupported
OnlyFO (A ⊗ B) = OnlyFO A ω×ω OnlyFO B
```

Then it is only a matter of stating that a helper for a datatype description *D* is made out of instances of ConHelper *p C*, one for every constructor description *C* in *D*.

```
data Helpers p : DataDesc P I n → Setω where
  instance nil   : Helpers p []
           cons : ∀ {n} {C : ConDesc P I ϵ} {D : DataDesc P I n}
                → ⦃ ConHelper p C ⦄ → ⦃ Helpers p D ⦄ → Helpers p (C ∷ D)

lookupHelper : Helpers p D → (k : Fin n) → ConHelper p (lookupCon D k)
```

Because these two definitions have *constructor instances*, it suffices to prefix datatype-generic functions with an instance argument of type Helpers *p D* to make Agda search for the needed instances available in scope. Defining all the arguments to the constructors of Helpers *p D* and ConHelpers *p c* as instance arguments is what makes the recursive instance search happen. Since

[7] ⊤ω, ⊥ω and _ω×ω_ are redefinitions of ⊤, ⊥ and _×_ in Setω. HigherOrderArgumentsNotSupported is a custom datatype with no constructors, for error-reporting purposes.

helpers are indexed by constructor descriptions, the path to follow while seeking instances is fully syntax-directed and determined by the description of the datatype at hand.

4.2 Generic Show

We follow the recipe given at the start of this section to define a datatype-generic Show instance. Given (A : Indexed $P\ I\ \ell$) and (H : HasDesc A), we start by refining the type of helpers we are interested in. In this case, we want an instance of Show for every type of value appearing in our datatype constructors (Arg ≡ Show). Furthermore, we allow high-order inductive arguments, even though we only display those that are first-order (Ind ≡ λ _ → ⊤ω).

```
open HasDesc H
open Helpers P I Show (λ _ → ⊤ω)

ShowHelpers : ∀ p → Setω
ShowHelpers p = Helpers p D
```

Given helpers (SH : ShowHelpers p) for our current datatype and parameters p, we define a generic show function using 3 mutually-recursive functions: To display a value, we retrieve its outermost constructor along with its arguments, then return the constructor's name concatenated with the displayed arguments.

```
show-wf : (x : A′ (p , i)) → Acc x → String
show-wf x (acc xacc) with split x
... | (k , x’) = Vec.lookup names k ++ "("
                ++ showCon (lookupCon D k) (lookupHelper SH k) x’ xacc
                ++ ")"
```

If one argument is a value of some other type, we retrieve the appropriate Show instance provided by the helper and use it. We only display inductive arguments if they are *first-order*, recursing on the accessibility witness.

```
showCon : (C : ConDesc P I V) (H : ConHelper p C) (x : ⟦ C ⟧Con A′ (p , v , i))
          → AllCon Acc C x → String
showCon _ var x _ = ""
showCon _ (pi {C = C} ⦃ SS ⦄ ⦃ HC ⦄) (x , y) yacc
  = show ⦃ SS ⦄ x ++ "," ++ showCon C HC y yacc
showCon _ (prod {A} {var _} ⦃ HA ⦄) (x , _) (xacc , _)
  = showIndArg A x xacc
showCon _ (prod {A} {B} ⦃ HA ⦄ ⦃ HB ⦄) (x , y) (xacc , yacc)
  = "(" ++ showIndArg A x xacc ++ "),  " ++ showCon B HB y yacc

showIndArg : (C : ConDesc P I V) (x : ⟦ C ⟧IndArg A′ (p , v))
             → AllIndArg Acc C x → String
showIndArg (var i) x xacc = show-wf x xacc
showIndArg (π S C) x _ = "?f"
showIndArg (A ⊗ B) (x , y) (xacc , yacc) =
  "(" ++ showIndArg A x xacc ++ "),  (" ++ showIndArg B y yacc ++ ")"
```

The public interface simply computes the accessibility witness before recursing.

```
show’ : A′ (p , i) → String
show’ x = show-wf x (wf x)
```

Using this construction to display a natural number, we get:

```
_ : show’ natD 5 ≡ "suc(suc(suc(suc(suc(zero())))))"
_ = refl
```

The deriveShow from our library is obtained by properly currying show’ over the telescopes of parameters and indices, so that each of them are given sequencially to deriveShow rather than together inside a deeply-nested sigma type.

4.3 Generic Induction Principle

As demonstrated by Chapman et al. [2010], formulating the induction principle on an universe of datatypes is straightforward. The challenge we face here is the one of defining an induction principle that behaves just as if it were handwritten. Given a predicate or *motive* on a concrete Agda datatype:

$$Pr : \mathsf{Pred}'\ I\ \lambda\ i \to \mathsf{A}'\ (p\ ,\ i) \to \mathsf{Set}\ c$$

The first step is to compute the *type* of the elimination rule for each constructor [McBride 2002]. Given a constructor, its elimination rule must convey that if Pr holds for some recursive occurence, then it holds for any value built using said constructor applied to them.

```
Pr′ : A′ (p , i) → Set c
Pr′ {i} = unpred′ I _ Pr i

levelElimIndArg levelElimCon : ConDesc P I V → Level

MethodIndArg : (C : ConDesc P I V) → ⟦ C ⟧IndArg A′ (p , v) → Set (levelElimIndArg C)
MethodIndArg (var _) x = Pr′ x
MethodIndArg (π S C) x = (s : S _) → MethodIndArg C (x s)
MethodIndArg (A ⊗ B) (mA , mB) = MethodIndArg A mA × MethodIndArg B mB

MethodCon : (C : ConDesc P I V)
          → (∀ {i} → ⟦ C ⟧Con A′ (p , v , i) → Set c)
          → Set (levelElimCon C)
MethodCon (var x) f   = f refl
MethodCon (π S C) f   = (s : S _) → MethodCon C (f ∘ (s ,_))
MethodCon (A ⊗ B) f = {g : ⟦ A ⟧IndArg A′ (p , _)}
                        (Pg : MethodIndArg A g)
                      → MethodCon B (f ∘ (g ,_))

Methods : ∀ k → Set (levelElimCon (lookupCon D k))
Methods k = MethodCon (lookupCon D k) λ x → Pr′ (constr (k , x))
```

In the code above, note how for the MethodCon $(A \otimes B)\ f$ case i.e. when there is an inductive argument in a constructor, the induction method must quantify over any such value (g) and requires a proof that Pr holds for g (Pg). We omit the definition of levelElimIndArg and levelElimCon. If we do a sanity check and compute the type of the elimination rules for natural numbers, we get:

```
Methods zero       ≡ Pr 0
Methods (suc zero) ≡ ∀ {n} → Pr n → Pr (suc n)
```

For vectors, we have:

```
Methods zero       ≡ Pr []
Methods (suc zero) ≡ ((n : Nat) (x : A) {xs : Vec A n} → Pr xs → Pr (x :: xs))
```

Assuming we have been given all the elimination methods (*methods* : ∀ *k* → Methods *k*), we show that we can prove the motive *Pr* holds for any value, as such:

- We find the outermost constructor of the input value, and select the corresponding elimination rule.
- We iteratively give all the arguments to the elimination method, calling the elimination principle recursively for every inductive argument.
- When all arguments have been supplied, what remains of the method is simply the proof that the property holds for the input value, which we return.

```
elimData-wf : (x : ⟦ D ⟧Data A′ (p , i)) → AllData Acc D x → Pr′ (constr x)

elim-wf : (x : A′ (p , i)) → Acc x → Pr′ x
elim-wf x (acc a) = subst Pr′ (constr∘split x) (elimData-wf (split x) a)

elimData-wf (k , x) a = elimCon (lookupCon D k) (methods k) x a
  where
    elimIndArg : (C : ConDesc P I V) (x : ⟦ C ⟧IndArg A′ (p , v))
               → AllIndArg Acc C x → MethodIndArg C x
    elimIndArg (var _  ) x a = elim-wf x a
    elimIndArg (π S C  ) x a = λ s → elimIndArg C (x s) (a s)
    elimIndArg (A ⊗ B ) (xa , xb) (aa , ab)
      = elimIndArg A xa aa , elimIndArg B xb ab

    elimCon : (C   : ConDesc P I V)
              {mk : ∀ {i} → ⟦ C ⟧Con A′ (p , v , i) → ⟦ D ⟧Data A′ (p , i)}
              (mth : MethodCon C (λ x → Pr′ (constr (mk x))))
              (x   : ⟦ C ⟧Con A′ (p , v , i))
             → AllCon Acc C x → Pr′ (constr (mk x))
    elimCon (var _  ) mth refl a    = mth
    elimCon (π _ C  ) mth (s , x) a = elimCon C (mth s) x a
    elimCon (A ⊗ B ) mth (xa , xb) (aa , ab)
      = elimCon B (mth (elimIndArg A xa aa)) xb ab

elim : (x : A′ (p , i)) → Pr′ x
elim x = elim-wf x (wf x)
```

Notice how since elimData-wf proves Pr′ (constr *x*) for any accessible *x*, and we apply elimData-wf on split *x* in elim-wf, we have to use subst in order to go from a proof of Pr′ (constr (split *x*)) to a proof of Pr′ *x*. Because constr and split are *shallow* conversion functions in the sense that they only unfold and reconstruct the outermost constuctor of a value, it is always possible to define constr∘split such that it computes to refl as soon as the outermost constructor of x is known.

If such is the case, subst disappears entirely and computation goes through on open terms. We made sure that our deriveDesc macro always generates proofs that satisfy this requirement.

This implementation thus highlights the following properties of our encoding:

- Because parameters and indices are explicitely accounted for in the encoding, we are able to define constructions whose types are identical to the handwritten version.
- Because we rely solely on shallow conversion functions rather than convert full terms to an internal representation, we are able to define constructions that never get stuck on equality proofs, leading to a much more user-friendly computational behaviour.
- Thanks to Setω, the implementation of datatype-generic constructions in Agda is made much simpler, as we no longer have to find tricks to emulate cumulativity [effectfully 2016c].

5 RELATED WORK

Two main ideas were crucial in the development of datatype-generic programming in dependent type theory. First, the notion of a *universe* with an interpretation function was introduced by Martin-Löf [1984], and was equipped with an *elimination principle* by Nordström et al. [1990], enabling the definition of generic functions by induction on the universe. Dybjer and Setzer [1999] gave a systematic study of universes as *inductive-recursive types*. Meanwhile, outside of the context of type theory Böhm and Berarducci [1985] introduced the idea of writing generic programs as functions over codes representing a datatype, which was further developed by Bird et al. [1996], Jay [1995], Jansson and Jeuring [1997] (PolyP), and Hinze and Jeuring [2003] (Generic Haskell). These two ideas were combined by Benke et al. [2003], who gave the first real application of a type-theoretic universe to the implementation of datatype-generic programs. Writing datatype-generic programs idea proved to be one of the most successful applications of full-spectrum dependent types, and was further developed by Morris [2007], Altenkirch et al. [2006], Morris et al. [2009], and Weirich and Casinghino [2010].

Chapman et al. [2010] introduced a *closed* dependent type theory where all datatypes are given by a code in a universe, including the type of codes itself, which formed the basis of the Epigram 2 prototype. This approach was extended in the work by Évariste Dagand [Dagand and McBride 2012; Évariste Dagand 2013], who gives an algorithm for *elaborating* a datatype definition to a code in the universe, which can be compared to our own algorithm for computing the code of a given Agda datatype through reflection.

Andjelkovic [2011] solves the problem that some generic operations (e.g. decidable equality) cannot be defined for all types in the universe by indexing the universe by the kind of features it uses. In contrast, we take a more flexible approach by emitting a (possibly unsatisfiable) constraint specific to the generic construction.

Recent attempts at defining a practical encoding of datatypes in Agda include the work by Diehl and Sheard [Diehl 2017; Diehl and Sheard 2013, 2014], Sijsling [2016] and effectfully [effectfully 2016a,b]. Each of these implementations come with their own set of limitations.

- The work of Sijsling [2016] can only encode datatypes living in Set_0. Parameters and indices are part of the encoding, but indices cannot depend on parameters. Only first-order inductive arguments are permitted. Because this work is focusing on ornaments, only a generic fold is implemented, and it operates on the encoded datatypes. It was never made available as a library of reusable components.
- The generic-elim library [Diehl and Sheard 2014] is slightly more permissive regarding the level at which datatypes can be defined, but does not attempt to bridge the gap between concrete

datatypes and their encoded counterpart. Its encoding of datatypes does not allow higher-order inductive arguments. Generic constructions are difficult to use and descriptions have to be constructed by hand as no macro is provided to derive the encoding of existing Agda datatypes.
- effectfully's *generic* library [effectfully 2020] is the most successful attempt at enabling datatype-generic programming in Agda, and what we base our work upon. If its encoding is the most expressive of the three, it cannot be used in the safe fragment of Agda. Only two datatype-generic constructions are provided. Because parameters and indices are not part of the encoding, reflection has to be used in order to implement usable generic constructions on concrete datatypes. Since it uses deep conversion functions between datatypes and their encoding, constructions suffer from poor computational behavior and get stuck on open terms.

This library is our attempt at resolving these long-standing limitations.

6 CONCLUSION AND FUTURE WORK

Using a universe for datatype-generic programming without reflection is one of the flagship applications of dependently typed languages. At the same time, the proliferation of new datatypes that is encouraged by these languages means having easy access to generic constructions is essential. Past attempts at making these universes available to users have largely failed to gain traction, either because they do not cover a wide enough set of datatypes, because they incur a heavy encoding overhead, or because they are just not set up to be usable 'out of the box' and rely on unsafe code. The library we present in this paper takes one further step along the path of resolving these issues and making datatype-generic programming available to regular Agda users. Its interface strives to be easy to use and work for a general class of datatypes. At the same time we provide many abstractions to let users define their very own datatype-generic constructions, including the generation of constraints that must be satisfied for a specific generic construction. Nevertheless, some rough edges remain to be taken care of, which we discuss below.

Universe-polymorphism. While our encoding does support datatypes defined at any universe level, we currently do not support universe-polymorphic datatypes, that is, datatypes *quantified* over universe levels. This however does not prevent one from using our generic constructions, as it is possible to define a family of descriptions for any level. Still, our deriveDesc macro will fail when applied to datatypes where some parameters are universe levels. We think a tangible solution could be to extend the macro to support the following pattern where the description of V is derived at an arbitrary universe level a:

```
data V {a} (A : Set a) : Set a
VD : ∀ {a} → HasDesc (V {a})
VD {a} = deriveDesc (V {a})
```

Once this is implemented, it should become possible to test our library on the plethora of datatypes defined in Agda's standard library, and assess how many of them fall in the set of datatypes that can be encoded in our universe.

Nested inductive families with explicit positivity annotations. An attentive reader might have noticed that our encoding, much like the one of effectfully [2020] or Chapman et al. [2010], does not support nested inductive types other than tuples. This is quite unfortunate as types like the following are common practice:

```
data Tree (A : Set) : Set where
  node : (x : A) (xs : List (Tree A)) → Tree A
```

The only way to describe such a datatype in our encoding is to rely on higher-order inductive arguments:

```
data Tree’ (A : Set) : Set where
  node : (x : A) (n : Nat) (xs : Fin n → Tree’ A) → Tree’ A
```

The issue comes from the fact that if we allow any type family F : Set → Set to be used for inductive arguments in the encoding, Agda cannot enforce that said F is strictly positive in its argument, and *rightly* prevents us from definiting the fixed point μ of the interpretation. An easy way out is to add existing common strictly positive functors in the encoding, such as lists and vectors, to at least offer some form of nesting the same way tuples of inductive arguments are supported. But this would require every datatype-generic construction to be updated for every new constructor added. Another more flexible approach would be to add *explicit polarity annotations* in Agda itself, in order to make it apparent that any such F in the encoding must be strictly-positive, just by looking at its type.

A more precise encoding of telescopes for more constructions. While we already showcase some generally useful constructions, the library could be extended further. Deriving the binary parametricity relation for a given datatype [Bernardy et al. 2010] would be a good candidate. Another possible direction would be to build further on our generic construction of the no-confusion property to implement a generic proof-relevant unification algorithm such as the one presented by Cockx and Devriese [2018]. However despite our best efforts, we were unable to implement some useful constructions such as a datatype-generic map and its associated laws. This stems from how simplistic and permissive our encoding of parameters is. Notably, once a parameter is added in a telescope, the encoding communicates no information as to how this parameter is being used in the rest of the telescope. The same is true inside constructor descriptions, where it’s impossible to distinguish between parameters used positively or negatively in the types introduced. While we are interested in finding an encoding of telescope accounting for this valuable information, it is unclear whether such an encoding would be practical enough to implement datatype-generic constructions.

More general classes of datatypes. Orthogonally, it could also be interesting to extend our approach to encodings of more general classes of inductive datatypes, such as inductive-recursive types [Diehl 2017; Dybjer and Setzer 1999, 2003], inductive-inductive types [Forsberg 2013; Forsberg and Setzer 2010], and quotient inductive-inductive types [Kaposi et al. 2019]. While there is a tradeoff to be made between the generality of the encoding and the ease of implementing new generic constructions, this can be largely mitigated by the ability to emit an impossible constraint for datatypes that are not supported by a particular generic construction. Another approach could be to abandon the goal of finding one encoding to rule them all, but accept that some encodings are more suitable than others for different classes of datatypes. The responsibility would fall on the user to use the appropriate encoding for the task at hand.

REFERENCES

David Abrahams and Aleksey Gurtovoy. 2004. *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond.* Pearson Education.

Agda Development Team. 2021. *Agda 2.6.2 documentation.* https://agda.readthedocs.io/en/v2.6.2/ Accessed [2021/07/10].

Thorsten Altenkirch, Conor McBride, and Peter Morris. 2006. Generic Programming with Dependent Types. In *Datatype-Generic Programming - International Spring School, SSDGP 2006, Nottingham, UK, April 24-27, 2006, Revised Lectures (Lecture Notes in Computer Science, Vol. 4719)*, Roland Carl Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring (Eds.). Springer, 209–257. https://doi.org/10.1007/978-3-540-76786-2_4

Stevan Andjelkovic. 2011. *A family of universes for generic programming.* Master's thesis. https://publications.lib.chalmers.se/records/fulltext/146810.pdf

Marcin Benke, Peter Dybjer, and Patrik Jansson. 2003. Universes for Generic Programs and Proofs in Dependent Type Theory. *Nord. J. Comput.* 10, 4 (2003), 265–289.

Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. 2010. Parametricity and dependent types. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 345–356. https://doi.org/10.1145/1863543.1863592

Richard S. Bird, Oege de Moor, and Paul F. Hoogendijk. 1996. Generic Functional Programming with Types and Relations. *Journal of Functional Programming* 6, 1 (1996), 1–28.

Edwin C. Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2021.9

Corrado Böhm and Alessandro Berarducci. 1985. Automatic Synthesis of Typed Lambda-Programs on Term Algebras. *Theoretical Computer Science* 39 (1985), 135–154.

James Chapman, Pierre Évariste Dagand, Conor McBride, and Peter Morris. 2010. The gentle art of levitation. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 3–14. https://doi.org/10.1145/1863543.1863547

David Christiansen and Edwin Brady. 2016. Elaborator reflection: extending Idris in Idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 284–297. https://doi.org/10.1145/2951913.2951932

Jesper Cockx and Dominique Devriese. 2018. Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory. *Journal of Functional Programming* 28 (2018). https://doi.org/10.1017/S095679681800014X

Pierre-Evariste Dagand and Conor McBride. 2012. Elaborating Inductive Definitions.

Larry Diehl. 2017. *Fully Generic Programming over Closed Universes of Inductive-Recursive Types.* Ph. D. Dissertation. Portland State University. https://pdxscholar.library.pdx.edu/cgi/viewcontent.cgi?article=4656&context=open_access_etds

Larry Diehl and Tim Sheard. 2013. Leveling up dependent types: generic programming over a predicative hierarchy of universes. In *Proceedings of the 2013 ACM SIGPLAN workshop on Dependently-typed programming, DTP@ICFP 2013, Boston, Massachusetts, USA, September 24, 2013*, Stephanie Weirich (Ed.). ACM, 49–60. https://doi.org/10.1145/2502409.2502414

Larry Diehl and Tim Sheard. 2014. Generic constructors and eliminators from descriptions: type theory as a dependently typed internal DSL. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*, José Pedro Magalháes and Tiark Rompf (Eds.). ACM, 3–14. https://doi.org/10.1145/2633628.2633630

Peter Dybjer and Anton Setzer. 1999. A Finite Axiomatization of Inductive-Recursive Definitions. In *Typed Lambda Calculi and Applications, 4th International Conference, TLCA 99, L Aquila, Italy, April 7-9, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1581)*, Jean-Yves Girard (Ed.). Springer, 129–146. http://link.springer.de/link/service/series/0558/bibs/1581/15810129.htm

Peter Dybjer and Anton Setzer. 2003. Induction-recursion and initial algebras. *Annals of Pure and Applied Logic* 124, 1-3 (2003), 1–47. https://doi.org/10.1016/S0168-0072(02)00096-9

effectfully. 2016a. Deriving eliminators of described data types. http://effectfully.blogspot.com/2016/06/deriving-eliminators-of-described-data.html

effectfully. 2016b. Descriptions. http://effectfully.blogspot.com/2016/04/descriptions.html

effectfully. 2016c. Emulating cumulativity in Agda. http://effectfully.blogspot.com/2016/07/cumu.html

effectfully. 2020. *Generic.* https://github.com/effectfully/Generic

Lucas Escot and Jesper Cockx. 2022. *Generics, a library for datatype-generic programming in Agda.* https://doi.org/10.5281/zenodo.6767057

Fredrik Nordvall Forsberg. 2013. *Inductive-inductive definitions.* Ph. D. Dissertation. Swansea University, UK. https://cronfa.swan.ac.uk/Record/cronfa43083 British Library, EThOS.

Fredrik Nordvall Forsberg and Anton Setzer. 2010. Inductive-Inductive Definitions. In *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6247)*, Anuj Dawar and Helmut Veith (Eds.). Springer, 454–468. https://doi.org/10.1007/978-3-642-15205-4_35

Ralf Hinze and Johan Jeuring. 2003. Generic Haskell: Practice and Theory. In *Generic Programming - Advanced Lectures (Lecture Notes in Computer Science, Vol. 2793)*, Roland Carl Backhouse and Jeremy Gibbons (Eds.). Springer, 1–56. http://springerlink.metapress.com/openurl.asp?genre=article&amp;issn=0302-9743&amp;volume=2793&amp;spage=1

LLC Jane Street Group. 2018. ppxlib's user manual. https://github.com/ocaml-ppx/ppx_deriving

Patrik Jansson and Johan Jeuring. 1997. Polyp - A Polytypic Programming Language. In *POPL*. 470–482. https://doi.org/10.1145/263699.263763

C. Barry Jay. 1995. A Semantics for Shape. *Science of Computer Programming* 25, 2-3 (1995), 251–283.

Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report.* Cambridge University Press.

Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. 2019. Constructing quotient inductive-inductive types. *Proceedings of the ACM on Programming Languages* 3 (2019). https://dl.acm.org/citation.cfm?id=3290315

Steve Klabnik and Carol Nichols. 2019. *The Rust Programming Language (Covers Rust 2018).* No Starch Press.

Per Martin-Löf. 1984. *Intuitionistic type theory.* Studies in proof theory, Vol. 1. Bibliopolis.

Conor McBride. 2002. Elimination with a Motive. In *Types for Proofs and Programs*, Paul Callaghan, Zhaohui Luo, James McKinna, Robert Pollack, and Robert Pollack (Eds.).

Conor McBride. 2013. Dependently typed metaprogramming (in Agda). *Lecture Notes* (2013).

Conor McBride, Healfdene Goguen, and James McKinna. 2004. A Few Constructions on Constructors. In *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3839)*, Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner (Eds.). Springer, 186–200. https://doi.org/10.1007/11617990_12

Peter Morris, Thorsten Altenkirch, and Neil Ghani. 2009. A Universe of Strictly Positive Families. *Int. J. Found. Comput. Sci.* 20, 1 (2009), 83–107. https://doi.org/10.1142/S0129054109006462

Peter W. J. Morris. 2007. *Constructing Universes for Generic Programming.* Ph. D. Dissertation. University of Nottingham, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.519405 British Library, EThOS.

Bengt Nordström, Kent Petersson, and Jan M Smith. 1990. *Programming in Martin-Löf's type theory.* Vol. 200. Oxford University Press.

Tim Sheard and Simon L. Peyton Jones. 2002. Template meta-programming for Haskell. *SIGPLAN Notices* 37, 12 (2002), 60–75. https://doi.org/10.1145/636517.636528

Yorick Sijsling. 2016. *Generic programming with ornaments and dependent types.* Master's thesis.

Stephanie Weirich and Chris Casinghino. 2010. Arity-generic datatype-generic programming. In *Proceedings of the 4th ACM Workshop Programming Languages meets Program Verification, PLPV 2010, Madrid, Spain, January 19, 2010*, Cormac Flanagan and Jean-Christophe Filliâtre (Eds.). ACM, 15–26. https://doi.org/10.1145/1707790.1707799

Pierre Évariste Dagand. 2013. *A cosmology of datatypes : reusability and dependent types.* Ph. D. Dissertation. University of Strathclyde, Glasgow, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.605921 British Library, EThOS.