Datatype-Generic Programming Meets Elaborator Reflection

HSIANG-SHANG KO, LIANG-TING CHEN, and TZU-CHI LIN, Institute of Information Science, Academia Sinica, Taiwan

Datatype-generic programming is natural and useful in dependently typed languages such as Agda. However, datatype-generic libraries in Agda are not reused as much as they should be, because traditionally they work only on datatypes decoded from a library's own version of datatype descriptions; this means that different generic libraries cannot be used together, and they do not work on native datatypes, which are preferred by the practical Agda programmer for better language support and access to other libraries. This paper presents a framework using Agda's elaborator reflection to instantiate datatype-generic programs as, and for, a useful range of native datatypes and functions —including universe-polymorphic ones— in programmer-friendly and customisable forms. Thanks to the power of elaborator reflection, generic programs do not need to be drastically rewritten compared to their traditional forms, making it easy to adapt existing generic libraries and develop new ones. We expect that datatype-generic libraries built with our framework —being interoperable with native entities— will finally be suitable for the toolbox of the practical Agda programmer.

1 INTRODUCTION

Parametrised by datatype structure, datatype-generic programs [Gibbons 2007] are ideal library components since they can be instantiated for a usually wide range of datatypes, including user-defined ones as long as their structures are recognisable by the datatype-generic programs. Particularly in dependently typed programming [Stump 2016; Brady 2017; Kokke et al. 2020], datatype-genericity has long been known to be naturally achievable [Benke et al. 2003; Altenkirch and McBride 2003], and is even more useful for organising indexed datatypes with intrinsic constraints and their operations. However, there is hardly any datatype-genericity in, for example, the Agda standard library, which instead contains duplicated code for similar datatypes and functions. The existing dependently typed datatype-generic libraries [McBride 2011, 2014; Dagand and McBride 2014; Diehl and Sheard 2016; Ko and Gibbons 2017; Allais et al. 2021] —mostly in Agda, which will be our default language— are not reused as much as they should be either. What is going wrong?

The major problem, we argue, is the lack of interoperability. The prevalent approach to datatype-generic programming in Agda (recapped in Section 2) is to construct a family of datatype descriptions and then decode the descriptions to actual datatypes via some least fixed-point operator μ. Generic programs take descriptions as parameters and work only on datatypes decoded from descriptions. Although this approach is theoretically rooted in the idea of universe à la Tarski [Martin-Löf 1975, 1984] and serves as a simulation of a more recent theory of datatypes [Chapman et al. 2010] (discussed in Section 7), it is not what we want: Generic libraries usually use their own version of datatype descriptions and are incompatible with each other, so only one library can be chosen at a time, which is unreasonable. Moreover, decoded datatypes are essentially segregated from native datatypes, and there is no point for the Agda programmer to abandon most of the language support and libraries developed for native datatypes in exchange for one generic library.

So what do we want from datatype-generic libraries? We want to write our own native datatypes and then instantiate generic programs for them. And in a dependently typed setting, we should be able to instantiate theorems (and, in general, constructions) about native datatypes and functions too. For a standard example, from the List datatype, we want to derive not only its fold operator

Authors' address: Hsiang-Shang Ko, joshko@iis.sinica.edu.tw; Liang-Ting Chen, liang.ting.chen.tw@gmail.com; Tzu-Chi Lin, vik@iis.sinica.edu.tw, Institute of Information Science, Academia Sinica, 128 Academia Road, Section 2, Nankang, Taipei, Taiwan, 115201.

2022. 2475-1421/2022/0-ART0 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
foldr : {A : Set ℓ} {B : Set ℓ'} → (A → B → B) → B → List A → B
foldr f e []       = e
foldr f e (a :: as) = f a (foldr f e as)
```

but also theorems about foldr, such as the following 'fold fusion' theorem (which allows us to optimise the composition of a foldr and a function h as a single foldr):

```
foldr-fusion : {A : Set ℓ} {B : Set ℓ'} {C : Set ℓ''} (h : B → C)
               {e : B} {f : A → B → B} {e' : C} {f' : A → C → C} →
               (he : h e ≡ e') (hf : ∀ a b c → h b ≡ c → h (f a b) ≡ f' a c)
               (as : List A) → h (foldr f e as) ≡ foldr f' e' as
foldr-fusion h he hf []       = he
foldr-fusion h he hf (a :: as) = hf a _ _ (foldr-fusion h he hf as)
```

Also important (especially in a dependently typed setting) is the ability to derive new datatypes — the standard example is the derivation of the vector datatype from list length,

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

length : {A : Set ℓ} → List A → ℕ
length []       = zero
length (a :: as) = suc (length as)
```

and subsequently we want to derive constructions about vectors too. What we want is conceptually simple but immediately useful in practice: automated generation of native entities that had to written manually —including all the entities shown above— from datatype-generic programs.

Luckily, datatype-generic programming has a long history of development in Haskell [Löh 2004; Magalhães 2012], where we can find much inspiration for our development in Agda. Generic programs in Haskell have always been instantiated for native datatypes, so the interoperability problem in Agda does not exist there. However, generic program instantiation in Haskell traditionally proceeds by inserting conversions back and forth between native and generic representations, causing a serious efficiency problem. The conversions are even more problematic in Agda because their presence would make it unnecessarily complicated to reason about instantiated functions. The Haskell community addressed the conversion problem using compiler optimisation [Magalhães 2013] and, more recently, staging [Pickering et al. 2020]. Unfortunately, compiler optimisation does not work for us because instantiated functions are reasoned about even before they are compiled, and they need to be as clean as hand-written code right after instantiation; this need could be met by staging, which is not available in dependently typed languages though.

Luckily again, in Agda there is something that can take the place of staging for generic program instantiation: elaborator reflection (inspired by Idris [Christiansen and Brady 2016]), through which the Agda metaprogrammer has access to operations for elaborating the surface language to the core. It turns out that datatype-generic programming and elaborator reflection are a perfect match in Agda. For example, to express dependency in types, datatype descriptions are usually higher-order and can be somewhat difficult to manipulate, but our transformation from descriptions to native datatypes is surprisingly natural thanks to the 'local variable creation' technique [Nanevski and Pfenning 2005; Schürmann et al. 2005], which is easily implemented using a few reflection primitives. Moreover, there is no need to drastically alter the form of generic programs (such as adding staging annotations) because they can be straightforwardly instantiated with (open-term) normalisation, also a reflection primitive.

We have developed a framework in Agda where datatype-generic programs can be instantiated as, and for, a useful range of native datatypes and functions through elaborator reflection. We do not need radically new datatype-generic programming techniques, but do need to adapt our

```
data ConDs (I : Set) : Set₁ where
  []  :                                  ConDs I
  _::_ : ConD I → ConDs I →              ConDs I
data ConD (I : Set) : Set₁ where
  ι : I                                  → ConD I
  σ : (A : Set) → (A → ConD I)           → ConD I
  ρ : RecD I →                 ConD I    → ConD I
data RecD (I : Set) : Set₁ where
  ι : I                                  → RecD I
  π : (A : Set) → (A → RecD I)           → RecD I
```

```
⟦_⟧Cs : ConDs I → (I → Set) → (I → Set)
⟦ []      ⟧Cs X i = ⊥
⟦ D :: Ds ⟧Cs X i = ⟦ D ⟧C X i ⊎ ⟦ Ds ⟧Cs X i
⟦_⟧C : ConD I → (I → Set) → (I → Set)
⟦ ι j     ⟧C X i = i ≡ j
⟦ σ A D   ⟧C X i = Σ[ a ∶ A ] ⟦ D a ⟧C X i
⟦ ρ D E   ⟧C X i = ⟦ D ⟧R X × ⟦ E ⟧C X i
⟦_⟧R : RecD I → (I → Set) → Set
⟦ ι i     ⟧R X   = X i
⟦ π A D   ⟧R X   = (a : A) → ⟦ D a ⟧R X
```

Fig. 1. A basic version of datatype descriptions and their base functor semantics

datatype descriptions —restricted to inductive families [Dybjer 1994] in this paper— to support commonly used Agda features, in particular universe polymorphism (Section 3). Our generic programs instantiate to native entities that are close to hand-written forms, and work on existing native entities —whose forms can be flexibly customised— through 'connections' to their generic counterparts (Section 4). The instantiation macros are a new and natural use case of elaborator reflection (Section 5); more generally, we give a Cook's tour of Agda's elaborator reflection (which is less documented), and promote the local name creation technique for handling higher-order syntax. As a demo, we adapt some existing generic constructions to our framework (Section 6). We expect that this work will facilitate the development of practical datatype-generic libraries in Agda, and provide motivations for theoretical investigations (Section 7). Our Agda code is available at https://github.com/Zekt/Type-Embellishment.

2 A RECAP OF DATATYPE-GENERIC PROGRAMMING

We start from a recap of standard datatype-generic programming in a dependently typed setting. In Section 2.1 we fix on a first-class representation of datatype definitions (restricted to inductive families [Dybjer 1994] in this paper). Then we review (F-)algebras in Section 2.2, the kind of generic program that our presentation will focus on (but not the only kind we use). The representations in this section are close to but not the final version, which we will develop in Sections 3 and 4.

2.1 Datatype Descriptions

There have been quite a few variants of datatype encoding [Altenkirch et al. 2007; Chapman et al. 2010; McBride 2011; Dagand and McBride 2014; Ko and Gibbons 2017]; here we use a three-layered version that closely follows the structure of an Agda datatype definition (comparable to de Vries and Löh's [2014] encoding). As a small running example, consider this accessibility datatype:

```
data Acc< : ℕ → Set where
  acc : (n : ℕ) (as : (m : ℕ) (lt : m < n) → Acc< m) → Acc< n
```

The first layer is the list of constructors, which for $\mathsf{Acc}_<$ consists of only **acc**; the type of **acc** has two fields n and as, which constitute the second layer; the type of the field as is described in the third layer as it ends with the recursive occurrence $\mathsf{Acc}_< m$, in front of which there are function arguments m and lt. Corresponding to the three layers, we use three datatypes of 'descriptions' ConDs, ConD, and RecD in Figure 1 —all parametrised by an index type I— to encode datatype definitions. For example, $\mathsf{Acc}_<$ is described by

```
Acc<D : ConDs ℕ
Acc<D = (σ ℕ (λ n → ρ (π ℕ (λ m → π (m < n) (λ lt → ι m))) (ι n))) :: []
```

Inhabitants of ConDs I are just lists of constructor (type) descriptions of type ConD I. Inhabitants of ConD I are also list-like: the elements can either be the type of a non-recursive field, marked by $\boldsymbol{\sigma}$, or describe a recursive occurrence, marked by $\boldsymbol{\rho}$, and the 'lists' end with $\boldsymbol{\iota}$. Different from ordinary lists, in the case of $\boldsymbol{\sigma}\ A\ D$ a new variable of type A is brought into the context of D (for example, in the type of **acc**, the field n appears in the type of as); this is done by making D a function from A, using the host language's function space to extend the context — we will continue to use this technique heavily in Section 3.[1] The $\boldsymbol{\iota}$ at the end of a ConD I should specify the index targeted by the constructor (for example, the final n in the type of **acc**). Inhabitants of RecD I use the same structure to describe dependent function types ending with a recursive occurrence.

A couple of syntax declarations will make descriptions slightly easier to write and read:

$$\textbf{syntax}\ \boldsymbol{\pi}\ A\ (\lambda\ a \to D) = \boldsymbol{\pi}[\,a : A\,]\ D; \quad \textbf{syntax}\ \boldsymbol{\sigma}\ A\ (\lambda\ a \to D) = \boldsymbol{\sigma}[\,a : A\,]\ D$$

For example, $\mathsf{Acc}_<\mathsf{D}$ can be rewritten as $(\boldsymbol{\sigma}[\,n : \mathbb{N}\,]\ \boldsymbol{\rho}\ (\boldsymbol{\pi}[\,m : \mathbb{N}\,]\ \boldsymbol{\pi}[\,lt : m < n\,]\ \boldsymbol{\iota}\ m)\ (\boldsymbol{\iota}\ n)) :: []$.

In the standard recipe, a description $D : \mathsf{ConDs}\ I$ is converted to a type family $\mu\ D : I \to \mathsf{Set}$ by taking the least fixed point of the base functor $[\![D]\!]_{\mathsf{Cs}} : (I \to \mathsf{Set}) \to (I \to \mathsf{Set})$:

$$\begin{aligned}&\textbf{data}\ \mu\ (D : \mathsf{ConDs}\ I) : I \to \mathsf{Set}\ \textbf{where}\\ &\quad \textbf{con} : \forall\ \{i\} \to [\![D]\!]_{\mathsf{Cs}}\ (\mu\ D)\ i \to \mu\ D\ i\end{aligned}$$

For example, we can redefine $\mathsf{Acc}_<$ as $\mu\ \mathsf{Acc}_<\mathsf{D} : \mathbb{N} \to \mathsf{Set}$, whose inhabitants are now constructed by the generic constructor **con**. Specified by the definition of the base functor $[\![D]\!]_{\mathsf{Cs}}$ in Figure 1,[2] the argument of **con** encodes the choice of a constructor and the arguments of the chosen constructor in a sum-of-products structure; for example, in Agda it is customary to use a pattern synonym [Pickering et al. 2016] to define **acc** in terms of **con**,

$$\textbf{pattern}\ \textbf{acc}\ n\ as = \textbf{con}\ (\textbf{inl}\ (n\ ,\ as\ ,\ \textbf{refl}))$$

where the arguments n and as of **acc** are collected in a tuple (product structure), tagged by **inl** (left injection into a sum type), and finally wrapped up with **con** as an inhabitant of $\mu\ \mathsf{Acc}_<\mathsf{D}\ n$. In general, when there are multiple constructors, the injection parts will look like **inl** …, **inr** (**inl** …), **inr** (**inr** (**inl** …)), etc, specifying the constructor choice in Peano-style. The equality proof **refl** at the end of the tuple needs a bit more explanation: in the type of **con**, the index i is universally quantified, which seems to suggest that we could construct inhabitants of $\mu\ D\ i$ for any i, but the equality proof forces i to be n, the index targeted by **acc**.

2.2 Algebras as Generic Programs

Now we can write programs on $\mathsf{Acc}_<$, for example its fold operator:

$$\begin{aligned}&\mathsf{foldAcc}_< : \{P : \mathbb{N} \to \mathsf{Set}\} \to (\forall\ n \to (\forall\ m \to m < n \to P\ m) \to P\ n) \to\\ &\qquad\qquad \forall\ \{n\} \to \mathsf{Acc}_<\ n \to P\ n\\ &\mathsf{foldAcc}_<\ p\ (\textbf{acc}\ n\ as) = p\ n\ (\lambda\ m\ lt \to \mathsf{foldAcc}_<\ p\ (as\ m\ lt))\end{aligned}$$

However, the point of using described datatypes such as $\mu\ \mathsf{Acc}_<\mathsf{D}$ is that we do not have to write $\mathsf{foldAcc}_<$ ourselves but can simply derive it as an instantiation of a generic grogram. The class of

[1] The computation power of the host language's function space has been better utilised in the datatype-generic programming literature (for example by McBride [2011, Section 2.1]), but we will refrain from abusing the function space in the descriptions we write for tasks beyond context extension, keeping our descriptions in correspondence with native datatypes. In general, if there are abuses, they will be detected at the meta-level (Section 5.2).

[2] $\bot$ is the empty type with no constructors. $A \uplus B$ is the sum of the types A and B with constructors $\textbf{inl} : A \to A \uplus B$ and $\textbf{inr} : B \to A \uplus B$. $\Sigma[\,a : A\,]\ B$ is a dependent pair type, where $\Sigma[\,a : A\,]$ binds the variable a, which can appear in B; the pair constructor _,_ associates to the right. Free variables in types (such as I in the types of $[\![\_]\!]_{\mathsf{Cs}}$, $[\![\_]\!]_{\mathsf{C}}$, and $[\![\_]\!]_{\mathsf{R}}$) are implicitly universally quantified.

```
fmap_Cs : (D : ConDs I) → (∀ {i} → X i → Y i) → ∀ {i} → ⟦D⟧_Cs X i → ⟦D⟧_Cs Y i
fmap_Cs (D :: Ds) f (inl xs)  = inl (fmap_C  D  f xs)
fmap_Cs (D :: Ds) f (inr xs)  = inr (fmap_Cs Ds f xs)

fmap_C  : (D : ConD I)  → (∀ {i} → X i → Y i) → ∀ {i} → ⟦D⟧_C  X i → ⟦D⟧_C  Y i
fmap_C  (ι i    ) f eq        = eq
fmap_C  (σ A D) f (a , xs)    = a , fmap_C (D a) f xs
fmap_C  (ρ D E) f (xs , xs')  = fmap_R D f xs , fmap_C E f xs'

fmap_R  : (D : RecD I)  → (∀ {i} → X i → Y i) → ⟦D⟧_R X → ⟦D⟧_R Y
fmap_R  (ι i    ) f x         = f x
fmap_R  (π A D) f xs          = λ a → fmap_R (D a) f (xs a)
```

Fig. 2. Datatype-generic functorial maps of base functors

generic programs we will focus on in this paper is '(F-)algebras' [Bird and de Moor 1997] (where the functor F is always some base functor $[\![D]\!]_{Cs}$ in this paper), whose type is defined by

```
Alg : ConDs I → (I → Set) → Set
Alg D X = ∀ {i} → ⟦D⟧_Cs X i → X i
```

Algebras are useful because they are the interesting part of a fold function: By a 'fold function' we mean a function defined recursively on an argument of some datatype by (i) pattern-matching the argument with all possible constructors, (ii) applying the function recursively to all the recursive fields, and (iii) somehow computing the final result from the recursively computed sub-results and the non-recursive fields. For example, foldAcc$_<$ is a fold function, and so are a lot of common functions such as list length. The first two steps are the same for all fold functions on the same datatype, whereas the third step is customisable and represented by an algebra, whose argument of type $[\![D]\!]_{Cs}\ X\ i$ represents exactly the input of step (iii). We can define a generic fold operator that expresses the computation pattern of fold functions and can be specialised with an algebra,

```
{-# TERMINATING #-}
fold : (D : ConDs I) → Alg D X → ∀ {i} → μ D i → X i
fold D f (con ds) = f (fmap_Cs D (fold D f) ds)
```

where fmap$_{Cs}$ is the functorial map for $[\![D]\!]_{Cs}$ (defined in Figure 2),[3] used here to apply fold $D\ f$ to the recursive fields in ds. (The unsafe TERMINATING pragma is not be a problem because our work does not use fold.) Libraries may provide generic programs in the form of algebras parametrised by descriptions, and the user gets a fold function for their datatype by applying fold to an algebra specialised to the description of the datatype. For example, by specialising a generic program in Section 6.1, we get an algebra (with some parameters of its own)

```
foldAcc<Alg : {P : ℕ → Set} → (∀ n → (∀ m → m < n → P m) → P n) → Alg Acc<D P
foldAcc<Alg p (inl (n , ps , refl)) = p n ps
```

which we then use to specialise fold to get foldAcc$_<$:

```
foldAcc< : {P : ℕ → Set} → (∀ n → (∀ m → m < n → P m) → P n) →
           ∀ {n} → Acc< n → P n
foldAcc< p = fold Acc<D (foldAcc<Alg p)
```

[3] For most of the generic programs in this paper we will provide only a sketch, because they are not too different from those in the literature. But as a more detailed example, the functorial map (Figure 2) is a typical generic program: The functorial map should apply a given function f to all the recursive fields in a sum-of-products structure while leaving everything else intact, and it does so by analysing the input description layer by layer — fmap$_{Cs}$ keeps the choices of **inl** or **inr**, fmap$_C$ keeps the σ-fields and ι-equalities, and finally fmap$_R$ applies f to the recursive fields (of type $X\ i$ for some i) pointwise.

Being able to treat folds generically means that we can write generic programs whose types have the form $\forall\,\{i\} \to \mu\ D\ i \to X\ i$, but this is not enough when, for example, we want to prove generic theorems by induction on $d : \mu\ D\ i$, in which case the types take the more complex form $\forall\,\{i\}\ (d : \mu\ D\ i) \to P\ d$ (where $P : \forall\,\{i\} \to \mu\ D\ i \to \mathsf{Set}$). Therefore we have another set of definitions for generic induction, corresponding to the scheme of elimination rules of inductive families [Dybjer 1994, Section 3.3]. The technical details of generic induction are omitted from the presentation, however, since the treatment is largely standard (closely following, for example, McBride [2011]), and our metaprograms (Section 5) work for fold and induction in the same way.

3 DATATYPE PARAMETERS AND UNIVERSE POLYMORPHISM

The datatype descriptions presented in Section 2 missed a couple of important features — it was probably tempting to generalise $\mathsf{Acc}_<$ to a version parametrised by a type A and a binary relation R on A, and moreover, by the levels of the universes in which A and R reside:[4]

$$\begin{array}{l}\textbf{data}\ \mathsf{Acc}\ \{\ell\ \ell' : \mathsf{Level}\}\ \{A : \mathsf{Set}\ \ell\}\ (R : A \to A \to \mathsf{Set}\ \ell')\ :\ A \to \mathsf{Set}\ (\ell \sqcup \ell')\ \textbf{where}\\ \quad \mathsf{acc}\ :\ (x : A) \to ((y : A) \to R\ y\ x \to \mathsf{Acc}\ R\ y) \to \mathsf{Acc}\ R\ x\end{array}$$

We ought to extend our datatype descriptions to express this kind of parametric and universe-polymorphic datatypes given their prevalence in Agda codebases. Conceptually this is straightforward: parameters are just variables in the context which can be referred to by the index type, datatype level, and constructor types, and we know an easy way to extend the context — just use the host language's function space. So a parametrised and universe-polymorphic datatype could be described by a parameter type P, a parametrised index type $I : (p : P) \to \mathsf{Set}\ (\ell_I\ p)$ (where $\ell_I : P \to \mathsf{Level}$), a parametrised datatype level $\ell_D : P \to \mathsf{Level}$, and a parametrised list of constructor descriptions $(p : P) \to \mathsf{ConDs}\ (I\ p)$ (where ConDs needs to be enriched with levels, which we will do in Section 3.3). For example, we could describe Acc with

$$\begin{array}{l}P \;=\; \Sigma[\,\ell : \mathsf{Level}\,]\ \Sigma[\,\ell' : \mathsf{Level}\,]\ \Sigma[\,A : \mathsf{Set}\ \ell\,]\ (A \to A \to \mathsf{Set}\ \ell')\\ I \;=\; \lambda\ (\_\,,\_\,,A\,,\_) \to A \qquad \ell_I \;=\; \lambda\ (\ell\,,\_) \to \ell \qquad \ell_D \;=\; \lambda\ (\ell\,,\ell'\,,\_) \to \ell \sqcup \ell'\end{array}$$

and a parametrised description that looks like $\mathsf{Acc}_<\mathsf{D}$. The actual description of Acc will be given in Section 3.4, but before that, there are some practical issues about level quantification (Section 3.1) and representation of parameters and indices (Section 3.2) to deal with.

3.1 Level Parameters

Unfortunately, we have in fact already bumped into the limitation of Agda's current design of universe polymorphism, where only finite levels can be dealt with uniformly via quantification over Level. Depending on whether a described datatype is universe-polymorphic or not, its parameter type may reside in a universe with a finite or infinite level: a non-universe-polymorphic parameter type, for example $\Sigma[\,A : \mathsf{Set}\,]\ (A \to A \to \mathsf{Set})$, resides in Set_1, which has a finite level, whereas the parameter type P defined above for Acc —call it P_{Acc} for short— cannot have type $\mathsf{Set}\ \ell_P$ for any $\ell_P : \mathsf{Level}$ because the type of A is $\mathsf{Set}\ \ell$ where ℓ can be arbitrarily large, so the type/kind of P_{Acc} has to be Set_ω, the first universe with an infinite level. Generic programs taking descriptions with parameters as input have to quantify the level of P, but currently Agda does not allow such quantification. This is one of the many problems created by the mismatch between the range of levels we need to handle and the limited power of level quantification; another is that the usual

[4] Agda's finite universe levels have a type $\mathsf{Level} : \mathsf{Set}$. We can use the primitives $\mathsf{lzero} : \mathsf{Level}$ and $\mathsf{lsuc} : \mathsf{Level} \to \mathsf{Level}$ to construct finite levels in the same way as we construct natural numbers, but we cannot pattern-match levels with lzero and lsuc. There is also an operator $\_\sqcup\_ : \mathsf{Level} \to \mathsf{Level} \to \mathsf{Level}$ that computes the maximum of two levels.

```
mutual
  data Tel : Level → Setω where
    []   : Tel lzero
    _::_ : (A : Set ℓ) (T : A → Tel ℓ′) → Tel (ℓ ⊔ ℓ′)
    _⧺_  : (T : Tel ℓ) (U : ⟦T⟧T → Tel ℓ′) → Tel (ℓ ⊔ ℓ′)

⟦_⟧T : Tel ℓ → Set ℓ
⟦[]⟧T      = ⊤
⟦A :: T⟧T  = Σ[a : A ] ⟦T a⟧T
⟦T ⧺ U⟧T   = Σ[t : ⟦T⟧T] ⟦U t⟧T
```

Fig. 3. (Tree-shaped) telescopes and their semantics as nested Σ-types

universe-polymorphic Σ-type former —with which we can only construct Σ-types with finite levels— is actually not enough for defining P_{Acc}; and one more will be mentioned in Section 3.2.

To avoid some of the problems (including the two mentioned above), we make a simplifying assumption, which holds for common universe-polymorphic datatypes: we assume that there is a list of level parameters separate from other ordinary parameters, and only the level parameters are used to achieve universe polymorphism. More formally, to describe a datatype, we start with a number $n : \mathbb{N}$ of level parameters, from which we can compute a type $\mathsf{Level} \hat{} n$ of tuples of n levels (as defined by $A \hat{} \mathbf{zero} = \top$ and $A \hat{} (\mathbf{suc}\ n) = A \times (A \hat{} n)$).[5] Parameterised by $\ell s : \mathsf{Level} \hat{} n$, the rest of the description can be written more succinctly as $\lambda\ \ell s \to (\ell_D , \ell_P , \ell_I , P , I , D)$ where ℓ_D is the datatype level, $P : \mathsf{Set}\ \ell_P$ the ordinary parameter type, $I : P \to \mathsf{Set}\ \ell_I$ the parametrised index type, and $D : (p : P) \to \mathsf{ConDs}\ (I\ p)$ the parametrised list of constructors. Note that the datatype level ℓ_D depends only on the level parameters ℓs, not the ordinary parameter of type P; moreover, given ℓs, the type P (which is $\Sigma[\, A : \mathsf{Set}\ \ell\,]\ (A \to A \to \mathsf{Set}\ \ell')$ in the case of Acc) always has a finite level now, avoiding the two problems in the previous paragraph.

3.2 Telescopes

At some point we will need to convert a description to a datatype definition, and it would be unsatisfactory in practice if the parameter and index types in the datatype definition were not in the conventional curried form. When currying, the encoding of multiple types in one nested Σ-type is ambiguous — how do we know whether a Σ-type is supposed to be interpreted as two types, with the latter depending on the former, or just one type? A natural solution is to use telescopes [de Bruijn 1991] to represent lists of parameter or index types, as shown in Figure 3. Again we use the host language's function space to bring variables of the types in the front of a telescope into the context of the rest of the telescope. Besides the usual cons constructor '::', we also include a constructor '⧺' for appending telescopes (which requires indexed induction-recursion [Dybjer and Setzer 2006] to define), making our telescopes tree-shaped; the reason will be clear when we reach Section 6. The index ℓ in the type $\mathsf{Tel}\ \ell$ of a telescope T is the maximum level appearing in T. This level is important since it is the universe level of the type $[\![\, T \,]\!]_{\mathsf{T}}$, which is a nested Σ-type inhabited by tuples whose components have the types in T.

A couple of syntax declarations will make telescopes slightly easier to write and read:

$$\mathbf{syntax}\ \_::\_\ A\ (\lambda\ x \to T) = [\, x : A \,]\ T\,; \quad \mathbf{syntax}\ \_⧺\_\ T\ (\lambda\ t \to U) = \|[\, t : T \,]\|\ U$$

For example, the parameters of Acc can be represented as $[\, A : \mathsf{Set}\ \ell\,]\ [\, R : (A \to A \to \mathsf{Tel}\ \ell')\,]\ []$ instead of $\mathsf{Set}\ \ell :: (\lambda\ A \to (A \to A \to \mathsf{Set}\ \ell') :: (\lambda\ R \to []))$.

From a telescope T it is straightforward to compute a curried function type $\mathsf{Curried}_{\mathsf{T}}\ T\ X$ which has arguments with the types in T, and ends with a given type $X : [\![\, T \,]\!]_{\mathsf{T}} \to \mathsf{Set}\ \ell'$ that can refer to all the arguments (collectively represented as a tuple of type $[\![\, T \,]\!]_{\mathsf{T}}$):

[5] $\top$ is the unit type with one constructor **tt**.

```
record DataD : Setω where field
  #levels : ℕ
  applyL  : Level ^ #levels → PDataD

record PDataD : Setω where field
  alevel {plevel} {ilevel} : Level
  {struct}                 : ConBs
  level-ineq : maxMap max-π struct ⊔
               maxMap max-σ struct
                 ⊑ alevel ⊔ ilevel
  Param      : Tel plevel
  Index      : ⟦Param⟧T → Tel ilevel
  applyP     : (ps : ⟦Param⟧T) → ConDs ⟦Index ps⟧T struct

data ConDs (I : Set ℓI) : ConBs → Setω where
  []  :                                  ConDs I []
  _::_ : ConD I cb → ConDs I cbs → ConDs I (cb :: cbs)

data ConD (I : Set ℓI) : ConB → Setω where
  ι : I                                  → ConD I []
  σ : (A : Set ℓ) → (A → ConD I cb)      → ConD I (inl ℓ :: cb)
  ρ : RecD I rb →      ConD I cb         → ConD I (inr rb :: cb)

data RecD (I : Set ℓI) : RecB → Setω where
  ι : I                                  → RecD I []
  π : (A : Set ℓ) → (A → RecD I rb)      → RecD I (ℓ :: rb)
```

Fig. 4. Parametric and universe-polymorphic datatype descriptions (all five layers), where RecB = List Level, ConB = List (Level ⊎ RecB), and ConBs = List ConB; modifications to Figure 1 are highlighted.

```
Curried_T : (T : Tel ℓ) → (⟦ T ⟧T → Set ℓ′) → Set (ℓ ⊔ ℓ′)
Curried_T []       X = X tt
Curried_T (A :: T) X = (a : A)          → Curried_T (T a) (λ t → X (a , t))
Curried_T (T +: U) X = Curried_T T (λ t → Curried_T (U t) (λ u → X (t , u)))
```

It is also straightforward to convert between this curried function type and its uncurried counterpart with the functions $\mathsf{curry}_\mathsf{T} : ((t : \llbracket T \rrbracket_\mathsf{T}) \to X\ t) \to \mathsf{Curried}_\mathsf{T}\ T\ X$ and $\mathsf{uncurry}_\mathsf{T}$ in the opposite direction (whose definitions are omitted). With these, we will be able to compute curried forms of parameters and indices when they appear in types (such as the type of the fold operator of Acc).

Incidentally, if we attempt a similar construction for Level ^ n (which can be viewed as a kind of specialised telescope) to produce curried forms of level parameters as well,

```
Curried_L : (n : ℕ) {f : Level ^ n → Level} → ((ℓs : Level ^ n) → Set (f ℓs)) → Set {! !}
Curried_L zero    X = X tt
Curried_L (suc n) X = (ℓ : Level) → Curried_L n (λ ℓs → X (ℓ , ℓs))
```

we will not be able to fill in the hole '{! !}' since it should be a finite level when n is zero (meaning that there is no level quantification), or ω when n is non-zero, going beyond the current capabilities of Agda's universe polymorphism. We will rely on elaborator reflection to deal with level parameters in Section 5.

3.3 Universe-Polymorphic Descriptions

It remains to adapt the description datatypes ConDs, ConD, and RecD from Section 2 for universe polymorphism. A first instinct might be copying what has been done for Tel (as constructor descriptions can be viewed as a slightly more complex kind of telescopes), enriching the Set-arguments to Set ℓ and perhaps indexing the datatypes with the maximum level, but this is not enough: the range of definitions depending on Tel (such as $\llbracket \_ \rrbracket_\mathsf{T}$ and $\mathsf{Curried}_\mathsf{T}$) is limited and requires only the computation of the maximum level, so indexing suffices; on the other hand, generic libraries built on the description datatypes may construct whatever they want from descriptions, and the need for non-trivial level computation will naturally arise if those constructions are universe-polymorphic. For example, in Sections 6.2 and 6.3 we will compute new universe-polymorphic datatypes from old ones, and will need to specify the new levels in terms of the old ones (and even reason about them). For a concrete example we can look at now, consider how the type of a base functor $\llbracket\ D\ \rrbracket_\mathsf{Cs}$ should be enriched: One place where we use the base functor is the type of an algebra $\{i : I\} \to \llbracket D \rrbracket_\mathsf{Cs}\ X\ i \to X\ i$ where $X : I \to \mathsf{Set}\ \ell$ is the result type, which can have any

level depending on what the algebra computes, so ℓ should be universally quantified in the type of $[\![D]\!]_{\mathsf{Cs}}$. But then, what should the level of the type $[\![D]\!]_{\mathsf{Cs}}\ X\ i$ be? This level —call it ℓ'— needs to be computed from ℓ and the structure of D, and the computation is non-trivial — for example, if D is [], then $[\![D]\!]_{\mathsf{Cs}}\ X\ i = \bot$, in which case ℓ' is simply lzero; if D is non-empty, then ℓ may or may not appear in ℓ', depending on whether there is a constructor with a ρ-field or not.

To allow level computation to be performed as freely as possible, we choose to index the description datatypes with as much useful information as possible, as shown in Figure 4. The index in the type of a description is a list which not only contains the levels of the fields but also encodes the description constructors used. Starting from the simplest RecD datatype, we index it with RecB = List Level, recording the levels of the π-fields. For ConD, the index type is ConB = List (Level ⊎ RecB), whose element sum type is used to record whether a field is σ or ρ. Finally, ConDs is indexed with ConBs = List ConB, collecting information from all the constructors into one list. With some helper functions, which constitute a small domain-specific language for datatype level computation, we can now specify the output level of $[\![\_]\!]_{\mathsf{Cs}}$:

```
[[_]]Cs : {I : Set ℓI} → ConDs I cbs → (I → Set ℓ) →
          (I → Set (maxMap max-π cbs ⊔ maxMap max-σ cbs ⊔
                    maxMap (hasRec? ℓ) cbs ⊔ hasCon? ℓI cbs))
[[ [] ]]Cs X i = ⊥
[[ D :: Ds ]]Cs X i = [[ D ]]C X i ⊎ [[ Ds ]]Cs X i
```

In prose, the output level is the maximum among the maximum level of the π-fields, the maximum level of the σ-fields, ℓ if the description has a ρ-field, and ℓ_I if the description has a constructor.

For our constructions, the approach works surprisingly well (even though the level expressions may look somewhat scary sometimes): we are able to write fully universe-polymorphic types while keeping almost all of the programs as they were — for example, the universe-polymorphic program of $[\![\_]\!]_{\mathsf{Cs}}$ is exactly the same as the non-universe-polymorphic one in Section 2. To see how the universe-polymorphic version of $[\![\_]\!]_{\mathsf{Cs}}$ is type-checked, we need to show a couple of definitions:

```
maxMap : (A → Level) → List A → Level
maxMap f []       = lzero
maxMap f (a :: as) = f a ⊔ maxMap f as

hasCon? : Level → ConBs → Level
hasCon? ℓ = maxMap (λ _ → ℓ)
```

It is easy to see that the output level in the $[\![\,[]\,]\!]_{\mathsf{Cs}}$ case is lzero, which is indeed the level of $\bot$. In the $[\![D :: Ds]\!]_{\mathsf{Cs}}$ case where D : ConD I cb and Ds : ConDs I cbs, the output level expands to

```
max-π cb ⊔ max-σ cb ⊔ hasRec? ℓ cb ⊔ ℓI ⊔
maxMap max-π cbs ⊔ maxMap max-σ cbs ⊔ maxMap (hasRec? ℓ) cbs ⊔ hasCon? ℓI cbs
```

where the first line is the level of $[\![D]\!]_{\mathsf{C}}\ X\ i$ and the second line is inductively the level of $[\![Ds]\!]_{\mathsf{Cs}}\ X\ i$, and indeed the level of the sum type is their maximum. It may appear that we skipped several steps applying the associativity and commutativity of '⊔', but in fact these properties (along with some others) are built into Agda's definitional equality on Level, so the definition of $[\![\_]\!]_{\mathsf{Cs}}$ type-checks without any manual proofs about levels.

3.4 Packing Up

We are now ready to pack what we have developed in this section into two new layers of datatype descriptions in Figure 4, mostly corresponding to the representation given in Section 3.1. The outermost layer DataD contains the number of level parameters and brings those level parameters into the context for the next layer PDataD (for 'parametrised datatype descriptions' or 'pre-datatype descriptions'), which contains the ordinary parameter types in the Param telescope (with maximum

level plevel), the index types in the parametrised Index telescope (with maximum level ilevel), and the parametrised list of constructor descriptions (whose structure is recorded in struct). Before explaining the rest of the PDataD fields, it may be helpful to see an example first: the Acc datatype can now be described by

```
AccD : DataD
AccD = record {#levels = 2; applyL = λ (ℓ , ℓ′ , _) → record
  {alevel    = ℓ′; level-ineq = refl
  ; Param    = [A : Set ℓ] [R : (A → A → Set ℓ′)] []
  ; Index    = λ (A , R , _) → [_ : A] []
  ; applyP   = λ (A , R , _) → (σ[x : A] ρ (π[y : A] π[_ : R y x] ι (y , tt)) (ι (x , tt))) :: []}}
```

When accessing the fields in the nested DataD and PDataD structures, the postfix projection syntax works better, as shown in the following definitions of base functor lifted to the DataD layer (which we will use later):

```
⟦_⟧D : (D : DataD) → ∀ {ℓs ps} →
       let I = ⟦D .applyL ℓs .Index ps⟧T in (I → Set ℓ) → (I → Set _)
⟦D⟧D {ℓs} {ps} = ⟦D .applyL ℓs .applyP ps⟧Cs
fmapD : (D : DataD) → ∀ {ℓs ps} →
        let I = ⟦D .applyL ℓs .Index ps⟧T in {X : I → Set ℓX} {Y : I → Set ℓY} →
        (∀ {i} → X i → Y i) → ∀ {i} → ⟦D⟧D X i → ⟦D⟧D Y i
fmapD D {ℓs} {ps} = fmapCs (D .applyL ℓs .applyP ps)
```

What remain to be explained are the PDataD fields alevel and level-ineq, which make sure that a corresponding datatype definition would pass Agda's universe checker. Here we are using the simpler datatype level–checking rule employed when Agda's `--without-K` option [Cockx et al. 2016] is turned on: the level of a datatype should at least be the maximum level of its index types, which is ilevel in our descriptions. If there are more components in the datatype level, they are specified in alevel, and the final datatype level is alevel ⊔ ilevel. The datatype level is not uniquely determined by the content of the datatype —for example, we could define alternative versions of natural numbers at any level— but must be no less than the level of any **π**- or **σ**-field of the constructors; this is enforced by level-ineq, where the relation $\ell \sqsubseteq \ell'$ is defined by $\ell \sqcup \ell' \equiv \ell'$. With level-ineq, we could even define a universe-polymorphic version of the μ operator from Section 2, but that is not the road we are going to take.

4 CONNECTING GENERIC AND NATIVE ENTITIES

Instead of a generic μ operator (Section 2), we will rely on Agda's elaborator reflection (Section 5) to manufacture a native datatype N from a description D : DataD (Section 3.4). Subsequently we may need to compute from D a new description that refers to N and its constructors. For example, in Section 6.3 we will define a datatype-generic predicate All P stating that a given predicate P holds for all the elements in a container-like structure; for lists, All specialises to

```
data ListAll {A : Set ℓ} (P : A → Set ℓ′) : List A → Set (ℓ ⊔ ℓ′) where
  []  :                                    ListAll P []
  _::_ : ∀ {a} → P a → ∀ {as} → ListAll P as → ListAll P (a :: as)
```

whose description can be computed from the description of List. Note that the index type of ListAll refers to List —a native datatype— and the indices targeted by the ListAll constructors refer to the native List constructors. These native entities need to be provided as additional input to the generic

```
DataT : DataD → Set_ω
DataT D = ∀ ℓs ps →
  let D_P = D .applyL ℓs
  in (is : D_P .Index ps) →
     Set (D_P .alevel ⊔ D_P .ilevel)

record DataC (D : DataD) (N : DataT D) : Set_ω where field
  toN      : ⟦D⟧_D (N ℓs ps) is → N ℓs ps is
  fromN    : N ℓs ps is → ⟦D⟧_D (N ℓs ps) is
  fromN-toN : (ns : ⟦D⟧_D (N ℓs ps) is) → fromN (toN ns) ≡ ns
  toN-fromN : (n : N ℓs ps is) → toN (fromN n) ≡ n
```

Fig. 5. Datatype connections

construction of All to allow the latter to specialise to the description of ListAll. More generally, if we provide enough structure about native datatypes so that what we can do with them are more or less the same as what we can do with those datatypes manufactured with μ, then we should be able to adapt generic programs that assume the presence of μ to work on these native datatypes instead. Such structure will be defined in Section 4.1, and analogously for fold functions in Section 4.2.

4.1 Datatype Connections

When μ was present, generic programs only needed to take a description D : DataD as input, and the corresponding native datatype would simply be μ D. Without μ, a corresponding native datatype N needs to be passed as an additional argument, and the first issue is the type of N: the native datatype is usually in a curried form, but it is easier for generic programs to handle an uncurried form, which can be computed by DataT D as defined in Figure 5. Regardless of how many parameters and indices there actually are, this uncurried form always represents level parameters, ordinary parameters, and indices as three arguments ℓs, ps, and is, presenting a uniform view to generic programs. The conversion from a curried form to the uncurried form is purely cosmetic and can be done with a wrapper function, for example,

```
AccT : DataT AccD
AccT _ (A , R , _) (as , _) = Acc R as
```

Note that AccT allows the form of the native datatype to be customised: we can change the order and visibility of the arguments (for example, the visibility of A is set to implicit in Acc) as long as we change AccT accordingly. Also, corresponding to the **con** constructor of μ, we need a function toN to construct inhabitants of N, and moreover, we need to perform pattern matching, which can be simulated by an inverse fromN of toN. These are packed into the record type DataC of 'datatype connections' in Figure 5, replacing μ's functionalities. (Strictly speaking, the inverse property fromN-toN here is only propositional whereas for **con** it is definitional, but this does not pose a problem for our examples in Section 6.) An inhabitant of DataC D N performs invertible conversion between the branches of the sum structure in D with the constructors of N, and the conversion is highly mechanical — for example,

```
AccC : DataC AccD AccT
AccC = record {toN      = λ {(inl (x , as , refl)) → acc x as       }
              ; fromN    = λ {(acc x as)            → inl (x , as , refl)}
              ; fromN-toN = λ {(inl (x , as , refl)) → refl          }
              ; toN-fromN = λ {(acc x as)            → refl          }}
```

Note that the order and visibility of constructor arguments can be customised as well.

The introduction of DataC supports a symmetric architecture where generic and native entities may grow separately but can be kept in sync (reminiscent of 'delta-based bidirectional transformations' [Abou-Saleh et al. 2018, Section 3.3]): we may compute a new description from an old one and then manufacture a native datatype from the new description, or write a native datatype and then derive its description; in either case, a connection is established between the generic and

```
record FoldP : Set_ω where field
  {Desc}   : DataD
  {Native} : DataT Desc
  Con      : DataC Desc Native
  #levels  : ℕ
  level    : Level ^ #levels → Level ^ (Desc .#levels)
  applyL   : ∀ ℓs → PFoldP (Desc .applyL (level ℓs))

record PFoldP (D : PDataD) : Set_ω where field
  {plevel} {clevel} : Level
  Param   : Tel plevel
  param   : ⟦Param⟧T → ⟦D .Param⟧T
  Carrier : ∀ ps → ⟦D .Index (param ps)⟧T → Set clevel
  applyP  : ∀ ps → Alg (D .applyP (param ps)) (Carrier ps)

Alg : ConDs I cbs → (I → Set ℓ) → Set _
Alg D X = ∀ {i} → ⟦D⟧Cs X i → X i
```

Fig. 6. Fold programs (parametrised algebras)

```
FoldT : FoldP → Set_ω
FoldT F = ∀ ℓs ps {is} →
  let open FoldP F; open PFoldP (F .applyL ℓs)
  in Native (level ℓs) (param ps) is →
     Carrier ps is

record FoldC (F : FoldP) (f : FoldT F) : Set_ω where field
  equation :
    ∀ {ℓs ps is} → let open FoldP F; open PFoldP (F .applyL ℓs) in
    (ns : ⟦Desc⟧D (Native (level ℓs) (param ps)) is) →
    f ℓs ps (Con .toN ns) ≡ applyP ps (fmap_D Desc (f ℓs ps) ns)
```

Fig. 7. Fold connections

native entities at the end. This architecture generalises the standard one involving μ, where D has a connection only with μ D, whereas in our architecture, connections can be established between any pair of description and datatype as long as they correspond. In particular, the forms of native datatypes and constructors (curried versus uncurried forms, order and visibility of arguments, etc) are not tightly coupled with descriptions (especially datatype-generically computed ones, which usually have prescribed forms) and can be customised by the programmer, which is vital in practice.

4.2 Fold Connections

Following the same architecture, we are also going to connect algebras with native fold functions. In general, algebras can be parametrised like foldAcc$_<$Alg from Section 2, and first we should give them a proper representation: analogous to DataD and PDataD (Figure 4), we use two layers FoldP (for 'fold programs') and PFoldP —defined in Figure 6— to store respectively the level parameters and the ordinary parameters. There are some additional fields that require explanation: FoldP is designed to contain sufficient information for manufacturing a corresponding native fold function. The fold function needs a type, which refers to the native datatype on which the fold function operates, so FoldP includes a field Con : DataC connecting the datatype description Desc on which the algebra operates to a Native datatype, enabling us to compute the type of the fold function using FoldT in Figure 7.[6] In the definition of FoldT, we also see that the fields level and param are used to compute the parameters for the native datatype argument from the parameters of the fold function. So, given F : FoldP, it can be connected to some f : FoldT F, but what should the connection be? Since f is supposed to replace an instantiation of the generic fold operator, what we need to know about f is that it satisfies a suitably instantiated version of the defining equation of fold. This equation constitutes the only field of the record type FoldC in Figure 7.

Incidentally, although we do not present the details of generic induction, the definitions are largely the same as what we have formulated for folds above, including IndP, IndT, IndC, etc. When we get to examples that require induction in Section 6, it should suffice to think of those generic programs as a more complex kind of parametrised algebras.

[6] Agda's **open** statement can be used to bring the fields of an inhabitant of a record type into scope — for example, the name Native in the definition of FoldT stands for F .Native because of **open** FoldP F. Moreover, an **open** statement can be used in a **let**-expression to limit its effect to the body of the **let**-expression.

In this paper we are only interested in manufacturing native fold functions from FoldP and establishing FoldC at the end, leaving the opposite direction as future work. As a concrete example, let us manufacture the universe-polymorphic fold operator for Acc from

```
foldAccP : FoldP
foldAccP = record {Con = AccC; #levels = 3
  ;level  = λ (ℓ'' , ℓ , ℓ' , _) → (ℓ , ℓ' , tt)
  ;applyL = λ (ℓ'' , ℓ , ℓ' , _) → record
   {Param = [A : Set ℓ] [R : (A → A → Set ℓ')] [P : (A → Set ℓ'')]
            [p : (∀ x → (∀ y → R y x → P y) → P x)] []
   ; param = λ (A , R , P , p , _) → A , R , tt
   ; Carrier = λ (A , R , P , p , _) (x , _) → P x
   ; applyP = λ {(A , R , P , p , _) (inl (x , ps , refl)) → p x ps}}}
```

(which is an instantiation of a generic program in Section 6.1). Along the way we will develop some generic facilities for mechanising the manufacturing process. First we need a curried type for the fold function, which can be computed by a variant of FoldT that uses $\mathsf{Curried}_\mathsf{T}$ (Section 3.2):

```
FoldNT : (F : FoldP) (ℓs : Level ^ (F .#levels)) → Set _
FoldNT F ℓs = let open FoldP F; open PFoldP (F .applyL ℓs) in
  Curried_T Param λ ps → Curried_T (Desc .applyL (level ℓs) .Index (param ps)) λ is →
  Native (level ℓs) (param ps) is → Carrier ps is
```

As explained in Sections 3.1 and 3.2, we have to treat the level parameters specially and cannot just curry them like what we have done with the ordinary parameters and indices. After normalising $\forall\,\{\ell s\} \to \mathsf{FoldNT}\ \mathsf{foldAccP}\ \ell s$ and currying the level parameters, we get the type

```
foldAcc : ∀ {ℓ'' ℓ ℓ'} (A : Set ℓ) (R : A → A → Set ℓ') (P : A → Set ℓ'')
          (p : ∀ x → (∀ y → R y x → P y) → P x) → ∀ x → Acc R x → P x
```

The definition of foldAcc should satisfy the equation of FoldC foldAccP foldAccT, but this equation does not work directly as a definition because toN is not a constructor. We can, however, change toN on the left-hand side to fromN on the right-hand side to get a definition, which we write as

```
foldAcc A R P p x a = fold-base foldAccP foldAcc A R P p x a
```

where fold-base generically expresses the computation pattern of fold functions in the usual non-recursive form that abstracts the recursive call as an extra argument *rec*:

```
fold-base : (F : FoldP) → ∀ {ℓs} → FoldNT F ℓs → FoldNT F ℓs
fold-base F {ℓs} rec =
  let open FoldP F; open PFoldP (F .applyL ℓs) in curry_T λ ps → curry_T λ is →
  applyP ps ∘ fmap_D Desc (λ {is} → uncurry_T (uncurry_T rec ps) is) ∘ Con .fromN
```

This definition of foldAcc, albeit one deemed non-terminating by Agda, implies the FoldC.equation because of the inverse property DataC.fromN-toN. To turn this into a valid definition, we pattern-match the variable *a* with all the possible constructors, although there is only one in this case:

```
foldAcc A R P p .x (acc x as) = fold-base foldAccP foldAcc A R P p x (acc x as)   (1)
```

Now normalise the right-hand side,

```
foldAcc A R P p .x (acc x as) = p x (λ y lt → foldAcc A R P p y (as y lt))   (2)
```

and this final definition can be directly shown to satisfy the connecting equation

```
data Term : Set where
  unknown : Term
  lit     : (l : Literal)                          → Term
  agda-sort : (s : Sort)                           → Term
  pi      : (a : Arg Type) (b : Abs Type)          → Term
  lam     : (v : Visibility) (t : Abs Term)        → Term
  var     : (i : Nat)      (xs : Args Term)        → Term
  con     : (c : Name)     (xs : Args Term)        → Term
  def     : (f : Name)     (xs : Args Term)        → Term
  pat-lam : (cs : Clauses) (xs : Args Term)        → Term
  meta    : (x : Meta)     (xs : Args Term)        → Term

data Sort : Set where
  set     : (t : Term) → Sort
  lit     : (n : Nat)  → Sort
  prop    : (t : Term) → Sort
  propLit : (n : Nat)  → Sort
  inf     : (n : Nat)  → Sort
  unknown : Sort

data Arg (A : Set) : Set where
  arg : (i : ArgInfo) (x : A) → Arg A

data Abs (A : Set) : Set where
  abs : (s : String) (x : A) → Abs A
```

Fig. 8. Reflected expressions

```
data Pattern where
  con    : (c : Name) (ps : Patterns)
           → Pattern
  proj   : (f : Name) → Pattern
  var    : (i : ℕ)    → Pattern
  absurd : (i : ℕ)    → Pattern
  lit    : (l : Literal) → Pattern
  dot    : (t : Term) → Pattern
```

Fig. 9. Reflected patterns

```
data Literal : Set where
  nat    : (n : Nat)    → Literal
  word64 : (n : Word64) → Literal
  float  : (x : Float)  → Literal
  char   : (c : Char)   → Literal
  string : (s : String) → Literal
  name   : (x : Name)   → Literal
  meta   : (x : Meta)   → Literal
```

```
postulate
  Name : Set
  Meta : Set

Args A    = List (Arg A)
Type      = Term
Telescope = List (String × Arg Type)
Names     = List Name
Patterns  = List Pattern
```

Fig. 10. Other types and abbreviations for reflection

```
foldAccC : FoldC foldAccP foldAccT
foldAccC = record {equation = λ {(inl (x , as , refl)) → refl}}
```

where foldAccT _ (A , R , P , p , _) {x , _} = foldAcc A R P p x is a wrapper function. (The inverse property DataC.fromN-toN does not appear in the proof, but we need it at the meta-level to argue generically that the proof always works.)

5 ESTABLISHING CONNECTIONS USING ELABORATOR REFLECTION

To automate the mechanical constructions in Section 4, we use Agda's elaborator reflection to define a set of macros (i) to translate between fully typed higher-order representations used by generic programs and uni-typed first-order representations used by the reflection framework in Section 5.2; (ii) to generate connections by synchronising two different types of representations in Section 5.3; (iii) to partially evaluate generic programs (Section 4.2) and derive definitions that are comparable, if not identical, to hand-written definitions in Section 5.4. Regarding elaborator reflection itself, we briefly introduce its basic design in Section 5.1, and leave explanations of reflection primitives to later sub-sections when needed. Due to space restrictions, we will only be able to provide sketches in most of this section. Examples of using the macros are given in Section 6.

5.1 Elaborator Reflection in Agda

The reflection API includes the elaborator monad TC, a set of TC computations, and datatypes —Term, Pattern, Literal, Clause, and Definition (Figures 8 to 12)— reflecting the core language where every expression is in weak head normal form and every application is in spine-normal form. The type Arg decorates a type with an ArgInfo about visibility (being implicit or not) and modality; the type Abs with a String as the binder's name. For brevity, we often suppress i : ArgInfo and a binder's name s, so for example we write **pi** a b (a reflected Π-type) for **pi** (**arg** i a) (**abs** s b).

```
data Clause : Set where
  clause : (Δ : Telescope) (lhs : Patterns) (rhs : Term)
         → Clause
  absurd-clause : (Δ : Telescope) (lhs : Patterns) → Clause
Clauses = List Clause
```

Fig. 11. Clauses

```
data Definition : Set where
  data-type : (#ps : ℕ) (cs : Names)   → Definition
  data-cons :           (d  : Name)    → Definition
  function  :           (cls : Clauses) → Definition
  ...
```

Fig. 12. A snippet of reflected declarations

The quotation of an expression e can be obtained by **quoteTerm** e and the resolved unique name for a definition f or a constructor by **quote** f. For example, **quoteTerm acc** is **con** (**quote acc**) [], where the empty list [] indicates that no arguments follow the **acc** constructor.

The TC monad (short for 'type-checking monad' [Agda Team 2022a]) stores states needed for elaboration such as the context of the call site, the scope of names with its definition, the set of metavariables and so forth. A macro f is a definition of type $A_1 \to \ldots \to \mathsf{Term} \to \mathsf{TC}\ \top$ declared with keyword **macro**. When executed during elaboration, the call site of f becomes a metavariable x supplied as the last argument of f for manipulation inside f. A minimal example is

```
macro give = unify
```

where we declare the primitive unify : Term → Term → TC ⊤ unifying two expressions as a macro. Elaborating give e would splice the given expression e : Term in place of the call, if Agda did nothing to its argument e. In fact, Agda quotes macro arguments of types Term and Name upon invocation, so elaborating give e amounts to running unify (**quoteTerm** e) (**meta** x []). Afterwards, the call site becomes e (and is elaborated again). In general, we can compute whatever expression we need inside a macro and then place it at the call site by unifying it with x.

5.2 Translating Higher-Order Representations with Local Variable Creation

This section's main task is to translate DataD, a fully typed higher-order representation, into the reflected language to declare native datatypes. The reflected language is, by contrast, a uni-typed first-order representation using de Bruijn indices and *not* hygiene, posing a challenge. Rather than presenting the full detail, it suffices to see how telescopes (Figure 3) are handled to get the essence of the translation. For example, the tree-shaped telescope ‖[(A , _) : [_ : Set] []‖ [_ : ($A \to A$)] [] of type Tel (lsuc lzero) should be translated to this (flattened) list of reflected types

```
'Set :: pi (var 0 []) (var 1 []) :: [] : Telescope        (3)
```

where the type Telescope is defined in Figure 10, and the variables **var** 0 [] and **var** 1 [] in **pi** both refer to the quotation 'Set = **agda-sort** (**lit** 0) of Set.

One obvious approach is to analyse the quotation of T : Tel ℓ. Such a macro needs to analyse abstract syntax trees *modulo judgemental equality* — it has to check which case is being analysed by reducing, say, a reflected expression **def** (**quote** f) xs for a definition f to one of the three cases **con** (**quote** Tel.[]) [], **con** (**quote** Tel._::_) xs, and **con** (**quote** Tel._++_) xs. This approach is error-prone, and moreover, Agda fails to check if the macro terminates or not.

Instead, let us pattern-match against T : Tel ℓ. The case for the empty telescope [] is simple to define but the other two cases A :: T and U ++ V seem impossible:

```
fromTel : Tel ℓ → TC Telescope
fromTel []       = return []
fromTel (A :: T) = ... fromTel (T ?) ...
fromTel (U ++ V) = ... fromTel (V ?) ...
```

```
exCxtTel : (T : Tel ℓ) (f : ⟦ T ⟧T → TC A) → TC A
exCxtTel []       f = f tt
exCxtTel (A :: T) f = exCxtT A λ _ x →
  exCxtTel (T x) λ t → f (x , t)
exCxtTel (T ++ U) f = exCxtTel T λ t →
  exCxtTel (U t) λ u → f (t , u)
```

Fig. 13. Context extension by T : Tel ℓ

```
exCxtℓs : (n : ℕ) (f : Level ^ n → TC A) → TC A
exCxtℓs zero    f = f tt
exCxtℓs (suc n) f = exCxtT Level λ _ ℓ →
  exCxtℓs n λ ℓs → f (ℓ , ℓs)
```

Fig. 14. Context extension by Level's

Note that T is a function from A, and V a function from $[\![U]\!]_{\mathsf{T}}$, for some *arbitrary* A and U; how do we give their arguments? We solve this problem by creating a *local variable*. The TC monad stores the context during elaboration, which can be extended by a variable of a given type to run a TC computation locally, using the primitive extendContext : Type → TC A → TC A. The first argument of extendContext is a reflected type, which should be the quotation of the actual value of A, which is not known until elaboration and thus cannot be obtained by **quoteTerm**; to obtain the quotation, we use the primitive quoteTC : A → TC Term. Then, in the second argument of extendContext, the local variable can be actually brought into scope by another primitive unquoteTC : Term → TC A which unquotes a given reflected expression within the TC monad. The above construction amounts to a TC computation

```
exCxtT : (B : Set ℓ) → (Type → B → TC A) → TC A
exCxtT B f = do ‘B ← quoteTC B
                extendContext ‘B (unquoteTC (var 0 []) >>= λ x → f ‘B x)
```

which creates a local variable x of type B for use in a TC computation f.

As for $[\![U]\!]_{\mathsf{T}}$, if we merely created a local variable $u : [\![U]\!]_{\mathsf{T}}$, then each reference to a component of u would be formed by projections fst and snd. For example, instead of (3) we would have

```
‘Set :: pi (def (quote fst) ((var 0 []) :: [])) (def (quote fst) ((var 1 []) :: [])) :: []
```

To eliminate projections, we create a list of local variables for each type in U : Tel ℓ as a tuple and give it to a TC computation (Figure 13). The last two cases of fromTel can then be defined by

```
fromTel (A :: T) = do
  exCxtT A λ ‘A x → do
    Γ ← fromTel (T x)
    return (‘A :: Γ)

fromTel (U ++ V) = do Γ ← fromTel U
                      exCxtTel U λ u → do
                        Δ ← fromTel (V u)
                        return (Γ ++ Δ)
```

As long as local variables are not pattern-matched, the computation can proceed. Indeed, we are exploiting the fact that our representations are used as if they are higher-order abstract syntax!

For the rest of the task we can only provide a sketch. To handle D : DataD, a TC computation defineByDataD is defined using the same local variable creation technique, but to apply D .applyL : Level ^ #levels → PDataD we need a variant exCxtℓs (Figure 13) which extends the context by a list of Level variables. To actually define a datatype described by D : DataD, we have extended Agda's **unquoteDecl** mechanism to allow macros to declare datatypes, and can write

```
unquoteDecl data d constructor c₁ … cₙ = defineByDataD D d (c₁ :: … :: cₙ :: [])
```

to introduce a datatype d and its constructors $c_1, \ldots, c_n$ into the scope. (The declaration form is somewhat verbose, but is chosen so that Agda's scope- and type-checking can be kept unchanged.)

Conversely, we also have a macro genDataD that expands to a description of a native datatype N. This direction is syntactical and unsurprising though — for example, telescopes are handled by

```
to‘Tel : Telescope → Term
to‘Tel = foldr (λ ‘A ‘T → ‘A ‘:: ‘T) ‘[]
```

where ‘[] = **con** (**quote** Tel.[]) [] and ‘A ‘:: ‘T = **con** (**quote** Tel._::_) (‘A :: **lam** ‘T :: []).

5.3 Generating Wrappers and Connections

Our next task is generating DataT, DataC, and FoldC (Figures 5 and 7). Their main ingredients are functions, which we define by pattern-matching λ-expressions, reflected as **pat-lam** in Figure 8. We will focus on DataT and omit DataC and FoldC, which are handled similarly. For example, we should generate the datatype wrapper AccT as

```
λ {{ℓ , ℓ′ , tt} (A , R , tt) (as , tt) → Acc {ℓ} {ℓ′} {A} R as}
```

where all the implicit arguments are given. To construct the **pat-lam**, we need to construct a clause whose left-hand side consists of patterns in the form of uncurried tuples of variables (which in general can be tree-shaped), and whose right-hand side is the native datatype applied to a list of curried arguments. Moreover, we need to assign the right visibilities to the arguments of Acc.

The macro genDataT that expands to a suitable pattern-matching λ-expression is

```
macro genDataT : DataD → Name → Term → TC ⊤
      genDataT D d hole = do ‘D ← quoteωTC D
                             checkType hole (def (quote DataT) (‘D :: []))
                             uncurryDataD D d >>= unify hole
```

where uncurryDataD : DataD → Name → TC Term does the real work (explained below), before which the primitive checkType : Term → Type → TC Term effectively provides a type annotation so that, for example, we can write AccT = genDataT AccD Acc without giving its type.

As for uncurryDataD, first we need to explain how clauses are given. A clause takes the form $\Delta \vdash \overline{p} \hookrightarrow e$ (Figure 11) where $\overline{p}$ is a list of patterns and e a reflected expression; the types of the variables in $\overline{p}$ are specified in the context Δ. It may appear that the context Δ needs to be fully specified beforehand, but actually it is not the case. This is because the reflected language plays the dual role of unchecked input and checked output of the elaborator [Cockx and Abel 2020]: the context of a checked clause is fully specified, whereas the context of an unchecked clause, which has the form $\overline{p} \hookrightarrow e$, can simply be filled with **unknown**'s. So uncurryDataD only needs to give $\overline{p}$ and e for each clause: Each pattern in $\overline{p}$ is computed from a Tel by labelling the elements with de Bruijn indices and replacing every ‘::’ or ‘++’ with a (reflected) pair constructor and ‘[]’ with the quotation of **tt**. And e is computed by applying the native datatype d to a list of de Bruijn indices paired with their visibilities, which are obtained with the help of the primitives getDefinition : Name → TC Definition and getType : Name → TC Type.

5.4 Instantiating Generic Functions with Normalisation

Finally, we define a TC computation defineFold (Figure 15) to instantiate a given F : FoldP as a native function f by (i) generating the instantiated type using FoldNT, (ii) generating a clause for each constructor of the datatype specified in F, and (iii) normalising these clauses.

For step (i), if we simply normalise, say, ∀ {ℓs} → FoldNT foldAccP ℓs, then we will have

```
∀ {ℓs : Level ^ 3} (A : Set (fst (snd ℓs))) (R : A → A → Set (fst (snd (snd ℓs)))) → …
```

instead of the preferred curried form shown in Section 4.2. As discussed in Section 3.2, we have to separately curry the level parameters with reflection after normalising the rest of the type, namely FoldNT F ℓs. But where does the variable ℓs come from? We reuse exCxtℓ (Figure 14) to extend the context with (F .#levels) many Level variables and collect those variable in a tuple ℓs, so that

```
defineFold : FoldP → Name → TC ⊤
defineFold F f = let open FoldP F in do
  -- step (i)
  `type ← addLevels #levels <$> exCxtℓs #levels λ ℓs → normalise =<< quoteTC (FoldNT F ℓs)
  declareDef f `type
  -- step (ii)
  cls ← caseM (getDefinition =<< FoldPToNativeName F) of λ
    { (data-type pars cs) → exCxtℓs #levels λ ℓs → forM cs (conClause pars #levels)
    ; _                   → typeError [] }
  -- step (iii)
  defineFun f =<< mapRHS normalise cls
```

Fig. 15. Definition of the TC computation defineFold

the expression FoldNT F ℓs makes sense and can be normalised with the primitive normalise : Term → TC Term. It remains to add quantification over the same number of Level variables we extend with (which is carried out by addLevels). Finally, we declare the type of f as the one we just generated using the primitive declareDef : Name → Type → TC ⊤.

For step (ii), we use the primitive getDefinition : Name → TC Definition to get the list cs of constructor names of the datatype F .Native, and generate the clause for each constructor. Abstracted from (1), each clause has the form

$$\Delta \vdash \overline{\ell}\ \overline{p}\ \overline{x}\ (c_i\ \overline{a}) \hookrightarrow \text{fold-base}\ F\ \{\ell_1, \ldots, \ell_n, \text{tt}\}\ f\ \overline{p}\ \overline{x}\ (c_i\ \overline{a}) \tag{4}$$

where $\overline{\ell}$ is the list of level parameters, $\overline{p}$ the ordinary parameters, $\overline{x}$ the indices, and $\overline{a}$ the constructor arguments. In general, the context Δ needs to be given because it will be used by the elaborator to determine the types of the variables when the right-hand side of (4) —call it e— is normalised. But in this case only the length of Δ needs to be specified, since all types will be determined upon synthesising the type of e, which is the first step of normalisation. Moreover, the patterns $\overline{x}$ are forced since the values of indices is determined by pattern-matching with $c_i\ \overline{a}$. It follows that $\overline{x}$ can be given as $\overline{\textbf{unknown}}$ on both sides. Therefore, we only need to generate a simpler clause

$$\overline{\ell}\ \overline{p}\ \overline{.\textbf{unknown}}\ (c_i\ \overline{a}) \hookrightarrow \text{fold-base}\ F\ \{\ell_1, \ldots, \ell_n, \text{tt}\}\ f\ \overline{p}\ \overline{\textbf{unknown}}\ (c_i\ \overline{a}) \tag{5}$$

for each constructor c_i.

For step (iii), we normalise the right-hand side of (5) within a context extended with variables from the left-hand side of (5) and obtain the list of clauses with normalised expressions on the right-hand side as the result of mapRHS normalise cls. Then, we define the function f by the primitive defineFun : Name → List Clause → TC ⊤ with the clauses we just obtained.

6 EXAMPLES

As a demo of our framework, here we provide some samples of generic constructions that should have been made available to the Agda programmer. To be more precise, these constructions are not new (or not too novel compared to those in the literature), but they have not been in the main toolbox of the Agda programmer, who prefers to work with native datatypes and functions; our framework makes it possible to instantiate these constructions for native entities. We will omit the details except those related to the design of our framework, and briefly discuss mechanisms that could make these constructions more convenient to use.

6.1 Fold Operators

The generic program that instantiates to fold operators on native datatypes is given the type

```
fold-operator : (C : DataC D N) → FoldP
```

As an example of instantiating the generic program, suppose that we have written the datatype Acc manually, and want to derive its fold operator. First we generate the description AccD by the macro genDataD Acc, and then the datatype connection AccC by genDataC AccD (genDataT AccD Acc). Now fold-operator AccC is exactly the fold program foldAccP, and the fold operator/function foldAcc can be manufactured from foldAccP by

```
unquoteDecl foldAcc = defineFold foldAccP foldAcc
```

and connected with foldAccP by genFoldC foldAccP foldAcc. (The macros are manually invoked for now, but the process could be streamlined as, say, pressing a few buttons of an editor.)

Let us look at fold-operator in a bit more detail. The ordinary parameters of fold-operator C are mainly a $[\![D]\!]_{\mathsf{D}}$-algebra in a curried form, so the work of fold-operator is purely cosmetic: at type level, compute the types of a curried algebra, which are the curried types of the constructors of D where all the recursive fields are replaced with a given carrier, and at program level, uncurry a curried algebra. The level parameters of fold-operator C include those of D and one more for the carrier X appearing in the Param telescope shown below, which also contains the ordinary parameters of D and a curried algebra represented as a telescope computed (by FoldOpTel) from the list of constructor descriptions in D:

```
fold-operator {D} C .applyL (ℓ , ℓs) .Param
  = let D_P = D .applyL ℓs in ‖[ ps : D_P .Param ]‖ [ X : Curried_T (D_P .Index ps) (λ _ → Set ℓ) ]
                                FoldOpTel (D_P .applyP ps) (uncurry_T X)
```

The type of X is in a curried form, which is then uncurried for FoldOpTel and other parts of the definition of fold-operator, for example the Carrier field:

```
fold-operator C .applyL (ℓ , ℓs) .Carrier = λ (ps , X , calgs) → uncurry_T X
```

This is a recurring pattern (which we first saw in Section 4.1): curried forms are exposed to the library user, whereas uncurried forms are processed by generic programs. The pattern is also facilitated by the telescope-appending constructor, which appears in Param above (disguised with the syntax ‖[…]‖): the parameters are instantiated in a curried form for the library user, but for generic programs they are separated into three groups ps, X, and $calgs$, making it convenient to refer to each group like in Carrier above.

It would not be too interesting if we could only manufacture functions but not prove some of their properties. For fold operators, one of the most useful theorems is the fusion theorem [Bird and de Moor 1997], of which foldr-fusion in Section 1 is an instance. The interface to the theorem is

```
fold-fusion : (C : DataC D N) (fC : FoldC (fold-operator C) f) → IndP
```

where the fold connection fC is used to quantify over functions f that are fold operators of N. Although the version of foldr in Section 1 is not manufactured by defineFold from the fold program foldListP = fold-operator ListC (where ListC is the datatype connection for List) and the arguments of foldr are in a different order from that specified by foldListP, there is no problem instantiating fold-fusion for foldr: in this case we can still manually write a wrapper foldrT = λ _ ((A , _) , B , e , f , _) → foldr f e : FoldT foldListP and manufacture a fold connection foldrC by genFoldC′ foldListP foldrT, and then fold-fusion ListC foldrC instantiates to foldr-fusion.

In general, if the library user is not satisfied with the form of a manufactured function (argument order, visibility, etc), they can print the function definition, change it to a form they want, and connect the customised version back to the library in the same way as we treated foldr. This customisation can be tiresome if it has to be done frequently, however, and there should be ways to

get the manufactured forms right most of the time. We have implemented a cheap solution where argument name suggestions (for definition-printing) and visibility specifications are included in FoldP (and IndP) and processed by relevant components such as $\mathsf{Curried}_\mathsf{T}$, and the solution works well for our small selection of examples. More systematic solutions are probably needed for larger libraries though, for example name suggestion based on machine learning [Alon et al. 2019] and visibility calculation by analysing which arguments can be inferred by unification.

6.2 Algebraic Ornamentation

Ornaments [McBride 2011] are descriptions of relationships between two structurally similar datatype descriptions, one of which has more information than the other. For example, after computing the descriptions ListD and NatD of List and $\mathbb{N}$ using genDataD, the following ornament states that List —having an additional element field— is a more informative version of $\mathbb{N}$:

```
ListO : DataO ListD NatD
ListO = record {level = λ _ → tt; applyL = λ (ℓ , _) → record
  {param = λ _ → tt; index = λ _ _ → tt; applyP = λ _ → ι :: ⁜ (Δ[_] ρ ι ι) :: ⁜ []}}
```

Do not worry about the details — the point here is that it is not difficult to write ornaments between concrete datatypes (and it will be even easier if there is a (semi-)automatic inference algorithm [Ringer et al. 2019] or an editing interface showing two datatypes side by side and allowing the user to mark their differences intuitively).

The first thing we can derive from an ornament is a forgetful function; in the case of ListO, the derived forgetful function is list length, which discards the additional element field. More can be derived from special kinds of ornaments, with a notable example being 'algebraic ornaments'. In our formulation, given a fold program F : FoldP we can compute a more informative version of the description F .Desc and an ornament between them:

```
AlgD : FoldP → DataD ;   AlgO : (F : FoldP) → DataO (AlgD F) (F .Desc)
```

The new datatype described by AlgD F has the parameters of F and an extra index that is the result of the fold corresponding to F. For example, the following datatype of 'algebraic lists' [Ko 2021] can be obtained by applying the macro defineByDataD to the description AlgD (fold-operator ListC):

```
data AlgList {A : Set ℓ} {B : Set ℓ′} (e : B) (f : A → B → B) : B → Set (ℓ ⊔ ℓ′) where
  []  :                                AlgList e f  e
  _::_ : (a : A) → ∀ {b} → AlgList e f b → AlgList e f (f a b)
```

But it is usually easier to program with more specialised datatypes such as Vec, which is a standard example of a datatype computed by algebraic ornamentation (over List), using the forgetful function derived from ListO, namely length. From the algebraic ornament between Vec and List, in addition to a forgetful function fromV we can also derive its inverse toV and the inverse properties:

```
fromV : Vec A n → List A
toV   : (as : List A) → Vec A (length as)

from-toV : (as : List A)   → fromV (toV as) ≡ as
to-fromV : (as : Vec A n) → (length (fromV as) , toV (fromV as)) ≡⟨ Σ ℕ (Vec A) ⟩ (n , as)
```

Note that, besides a new datatype Vec, we have derived an isomorphism between List A and $\Sigma\ \mathbb{N}\ (\mathsf{Vec}\ A)$ from the ornament ListO, allowing us to promote a natural number n to a list if a vector of type Vec A n can be supplied (or the other way around). In general, every ornament gives rise to such a 'promotion isomorphism' [Ko and Gibbons 2013]. A more interesting and notable example is the conversion between extrinsically and intrinsically typed λ-terms [Kokke et al. 2020]:

```
data Λ : Set where
  var : ℕ → Λ
  app : Λ → Λ → Λ
  lam : Λ → Λ

data Ty : Set where
  base : Ty
  _⇒_ : Ty → Ty → Ty
```

```
data _⊢_ : List Ty → Ty → Set where
  var : Γ ∋ τ → Γ ⊢ τ
  app : Γ ⊢ σ ⇒ τ → Γ ⊢ σ → Γ ⊢ τ
  lam : σ :: Γ ⊢ τ → Γ ⊢ σ ⇒ τ

data _⊢_:_ : List Ty → Λ → Ty → Set where
  var : (i : Γ ∋ τ) → Γ ⊢ var (to ℕ i) : τ
  app : Γ ⊢ t : σ ⇒ τ → Γ ⊢ u : σ → Γ ⊢ app t u : τ
  lam : σ :: Γ ⊢ t : τ → Γ ⊢ lam t : σ ⇒ τ
```

(The list membership relation '_∋_' will be defined in Section 6.3.) Write an ornament between the datatypes Λ and '_⊢_' of untyped and intrinsically typed λ-terms, and we get the typing relation '_⊢_:_' and an isomorphism between $\Gamma \vdash \tau$ and $\Sigma[\, t : \Lambda \,]\ \Gamma \vdash t : \tau$ for free, allowing us to promote an untyped term t to an intrinsically typed one if a typing derivation for t can be supplied.

We have deliberately omitted the types of the generic programs because they are somewhat verbose, making generic program invocation less cost-effective — for example, the generic programs proving the inverse properties need connections for the original and the new datatypes, the fold used to compute the algebraic ornament, and the 'from' and 'to' functions. In general, we should seek to reduce the cost of invoking generic programs. We have implemented a smaller-scale solution where generic programs use Agda's instance arguments [Devriese and Piessens 2011] to automatically look for the connections and other information they need, and the solution works well — for example, to-fromV can be derived by supplying just the names Vec and List. Larger-scale solutions such as instantiating the definitions in a parametrised module all at once may be required in practice.

Finally, we should briefly mention how AlgD handles universe polymorphism. Given F : FoldP, the most important change from F .Desc to AlgD F is adding a suitably typed **σ**-field (for example, the field b in AlgList) in front of every **ρ**-field; this is mirrored in the computation of the struct field of AlgD F from that of F .Desc, primarily using the function

```
algConB : Level → ConB → ConB
algConB ℓ []             = []
algConB ℓ (inl ℓ′ :: cb) = inl ℓ′ :: algConB ℓ cb
algConB ℓ (inr rb :: cb) = inl (max-ℓ rb ⊔ ℓ) :: inr rb :: algConB ℓ cb
```

(where max-ℓ rb is the maximum level in rb). Subsequently we need to prove level-ineq for AlgD F, which requires non-trivial reasoning and involves properties about algConB such as max-σ (algConB ℓ cb) ≡ max-π cb ⊔ max-σ cb ⊔ hasRec? ℓ cb. The reasoning is not difficult, but is probably one of the first examples that require non-trivial reasoning about universe levels.

6.3 Predicates on Simple Containers

There are not too many generic programs that work without assumptions on the datatypes they operate on; with dependent types, such assumptions can be formulated as predicates on datatype descriptions. As a simple example, below we characterise a datatype N as a 'simple container' type by marking some fields of its description as elements of some type X, and then derive predicates All P and Any P on N lifted from a predicate P on X, stating that P holds for all or one of the elements in an inhabitant of N. For example, the ListAll datatype (Section 4) is an instance of All.

The definition of simple containers (in several layers) is shown in Figure 16. The top layer SC on DataD only quantifies over the level parameters, and the main definition is at the next layer SC_P on PDataD: First is the element type El, which can refer to the ordinary parameters. Then in pos we assign a Bool to every **σ**-field indicating whether it is an element or not. More precisely,

```
SC : DataD → Set_ω
SC D = ∀ {ℓs} → SC_P (D .applyL ℓs)

SC_B : ConB → Set
SC_B = ListAll (λ {(inl _) → Bool
                  ; (inr _) → ⊤})

record SC_P (D : PDataD) : Set_ω where field
  {elevel} : Level
  El   : ⟦D .Param⟧_T → Set elevel
  pos  : ListAll SC_B (D .struct)
  coe  : (ps : ⟦D .Param⟧_T) → SC_Cs (D .applyP ps) pos (El ps)

SC_Cs : {I : Set ℓ_I} → ConDs I cbs → ListAll SC_B cbs → Set ℓ → Set_ω
SC_Cs []       _        X = ⊤
SC_Cs (D :: Ds) (s :: ss) X = SC_C D s E × SC_Cs Ds ss X

SC_C : {I : Set ℓ_I} → ConD I cb → SC_B cb → Set ℓ → Set_ω
SC_C (ι i    ) _        X = ⊤
SC_C (σ A D) (true  :: s) X = ((_ , A) ≡Σ[ℓ : Level] Set ℓ (_ , X)) × ((a : A) → SC_C (D a) s X)
SC_C (σ A D) (false :: s) X =                                          (a : A) → SC_C (D a) s X
SC_C (ρ D E) (_     :: s) X =                                                    SC_C  E     s X
```

Fig. 16. The simple-container predicate on datatype descriptions

the assignments are performed on the struct field of the description, and might not make sense since any σ-field could be marked with **true**, not just those of type El. However, the coe field of SC_P makes sure that the types of the fields marked with **true** are equal to El; subsequently, when a generic program sees such a field, it can use the equality to coerce the type of the field to El.

The All predicate is simpler since it is just the datatype created along with a promotion isomorphism (Section 6.2). For example, to derive ListAll, we mark the element field of List in an SC ListD structure, from which we can compute a more informative ListWP datatype that requires every element a to be supplemented with a proof of $P\ a$:

```
data ListWP {A : Set ℓ} (P : A → Set ℓ′) : Set (ℓ ⊔ ℓ′) where
  []  :                                  ListWP P
  ⟨_,_⟩::_ : (a : A) → P a → ListWP P → ListWP P
```

Then the ornament between ListWP and List gives rise to ListAll and an isomorphism between ListWP P and $\Sigma\ (\mathsf{List}\ A)\ (\mathsf{ListAll}\ P)$, allowing us to convert between a list of pairs of an element and a proof (ListWP P) and a pair of a list of elements (List A) and a list of proofs (ListAll P).

The Any predicate is more interesting since its structure is rather different from that of the original datatype, although in the case of List, the Any structure happens to degenerate quite a bit:

```
data ListAny {A : Set ℓ} (P : A → Set ℓ′) : List A → Set (ℓ ⊔ ℓ′) where
  here  : ∀ {a as} → P a          → ListAny P (a :: as)
  there : ∀ {a as} → ListAny P as → ListAny P (a :: as)
```

The list membership relation $xs \ni x$ used in Section 6.2, defined by $\mathsf{ListAny}\ (\lambda\ y \to x \equiv y)\ xs$, is a special case. In general, a proof of Any P is a path pointing to an element satisfying P, and we can write a generic lookup function that follows a path to retrieve the element it points to (resembling Diehl and Sheard's [2016] construction) — for ListAny, this lookup function specialises to

```
lookupListAny : {A : Set ℓ} {P : A → Set ℓ′} {as : List A} → ListAny P as → Σ A P
lookupListAny (here  p) = _ , p
lookupListAny (there i) = lookupListAny i
```

A path can be regarded as an (enriched) natural number that instructs the lookup function to stop (**here**/**zero**) or go further (**there**/**suc**) — that is, there is an ornament between Any and $\mathbb{N}$, allowing us to derive a forgetful function to$\mathbb{N}$ that computes the length of a path. Moreover, a path should specify which element it points to if stopping, or which sub-tree to go into if going further, so the

numbers of **here** and **there** constructors are exactly the numbers of element positions and recursive fields respectively. For example, the Any predicate for the datatype of balanced 2-3 trees below (taken from McBride [2014]) would have three **here** constructors and five **there** constructors:

```
data B23T : Height → Value → Value → Set where    -- both Height and Value are ℕ
  node₀ : {{l ⩽ r}}                                          → B23T  zero    l r
  node₂ : (x   : Value) → B23T h l x → B23T h x r            → B23T (suc h) l r
  node₃ : (x y : Value) → B23T h l x → B23T h x y → B23T h y r → B23T (suc h) l r
```

It is nice not having to write B23TAny and lookupB23TAny by hand.

7 DISCUSSION

Interoperability between generic and native entities. Our framework makes generic constructions parametrised by our version of datatype descriptions interoperable with native datatypes and functions. Generic libraries using their own descriptions could achieve the same by duplicating our framework, but there are alternatives: In Haskell, Magalhães and Löh [2014] provide automatic conversions between the datatype representations of several libraries —targeting different families of datatypes— and a representative representation, through which native datatypes are connected to all the libraries at once; our descriptions (possibly with extensions to support mutual recursion, induction-recursion, etc) can serve as a representative representation. Moreover, in a dependently typed setting it is possible to attain better uniformity and reusability by defining families of datatypes as predicates on a single representation like in Section 6.3, in contrast to a simply typed setting where different families of datatypes need to be modelled by separate representations.

Optimisation of datatype-generic programs. A traditional way to instantiate a generic program is to compose the program with conversions between a native datatype and its generic description. If the generic program was defined on datatypes decoded by the μ operator and then instantiated by the recursive conversion between the native and μ-decoded datatypes, the conversion overhead would be roughly the same as that of unoptimised Haskell generic programs, and had to be eliminated. Rather than optimising the composition of several recursive functions, the Haskell community has employed a shallow encoding and studied relevant optimisation/specialisation (see, for example, de Vries and Löh [2014, Section 5.1]); this encoding is the basis of our design. Below we discuss previous attempts at optimising instantiated programs using the shallow encoding.

Recent work using staging [Yallop 2017; Pickering et al. 2020] eliminates performance overheads by generating native function definitions that are almost identical to hand-written ones. There is, however, no implementation of staging in existing dependently typed languages, so we cannot compare them properly on the same ground. But it shares a similar purpose with our framework of generating function definitions containing neither generic representations nor conversions, making them comparable regardless of the specific languages they work in.

We compare staging with our framework from the view of partial evaluation [Jones et al. 1993]. A partial evaluator takes a general program and known parts of its input, and generates a program that takes the remaining input; the resulting program is extensionally equal to —and usually more optimised than— the general program partially applied to the known input. Our macro defineFold (Section 5.4) is a partial evaluator which specialises a generic program (general program) to a given datatype description (known input). Indeed, it has been observed that we can perform partial evaluation in functional languages by normalisation [Filinski 1999], which defineFold does.

Similar to a partial evaluator, a staged generic program is a more specialised program-generator — the generic/general program to be specialised has been fixed. However, staging requires manually inserting staging annotations; this puts burdens on the programmer and mixes a part of the

instantiation process with generic definitions. Our approach separates how we instantiate generic programs (by metaprograms) from how we define them (as algebras). As a result, our generic programs are annotation-free, making them easier to write and read. On the other hand, staging has its own benefit: Pickering et al. [2020, Section 4.1] provide principles followed by the programmer to avoid generic representations from appearing in specialised programs; without staging annotations, it seems difficult to formulate similar principles in our setting. But we can explore alternative approaches — for example, Alimarine and Smetsers [2004] give theorems guaranteeing that generic representations can be removed from instantiated programs with suitable types by normalisation.

There have also been attempts using compiler optimisation [de Vries 2004; Magalhães 2013], which are less relevant to our work as explained in Section 1. However, as generic programs become more complex, we may need more advanced code generation techniques that can be borrowed from compiler optimisation, possibly through extensions to elaborator reflection.

Analysis of higher-order encodings. The local variable creation technique has been used extensively to analyse higher-order encodings, but it has to be used with caution. Agda's elaborator checks if a local variable escapes its context in the result of extendContext during the runtime of elaboration.[7] This is similar to the nu constructor of the typed tactic language Mtac [Ziliani et al. 2015], which checks the escape of local variables during tactic execution. Indeed, Chen [2019] implemented Mtac's nu constructor by elaborator reflection in the same way. On the contrary, the ν-operator for *local name creation* in a modal calculus [Nanevski and Pfenning 2005] and a two-level functional language [Schürmann et al. 2005] statically ensures that local names cannot escape their scope.

Our higher-order encodings are not actually higher-order abstract syntax [Harper et al. 1993], since exotic terms such as $\mathsf{Bool} :: \lambda\,\{\mathbf{false} \longrightarrow [\,A : \mathsf{Set}\,]\,[]; \mathbf{true} \longrightarrow [\,P : (\mathbb{N} \longrightarrow \mathsf{Set})\,]\,[]\}$ are not excluded by the typing discipline. As argued by McBride [2011], exotic terms, which we do not use, could be exploited for richer expressiveness, but then it is harder to analyse and deviates too much from Agda's current datatype declarations, which our framework targets.

Dependently typed datatype-generic libraries. Our framework makes it possible and worthwhile to develop datatype-generic libraries for wider and practical use in Agda. The change to traditional Agda generic programs required by our framework is a mild generalisation from the operators μ and fold to datatype and fold connections capturing the behaviour of the operators, so it is easy to adapt existing libraries to our framework as well as develop new ones. There are still many opportunities to explore for dependently typed datatype-generic libraries: Even for recursion schemes [Yang and Wu 2022], a standard datatype-generic example, we can start supplying theorems about them like in Section 6.1. Datatypes derived from others, such as the lifted predicates All and Any in Section 6.3, are also common and should be treated generically (as opposed to manually duplicating an instance for each datatype as in the standard library [Agda Team 2022b]). Domain-specific organisation of datatypes with intrinsic invariants is another important goal, for which ornaments [McBride 2011] still have much potential (although the community has focussed mostly on lifting ornaments to programs and proofs [Dagand and McBride 2014; McDonell et al. 2016; Williams and Rémy 2018; Ringer et al. 2019]). For example, Section 6.2 mentions that the relationship between intrinsically and extrinsically typed λ-terms can be captured as an ornament, whose properties and derived constructions should be formulated generically for reuse in developments of typed embedded languages — a direction already proved fruitful by Allais et al. [2021].

Code generation versus first-class datatypes. Similar to staged approaches [Yallop 2017; Pickering et al. 2020], our framework instantiates generic programs by generating code separately for each

[7] In previous versions of Agda, the existing de Bruijn indices in an extended context were not weakened [Agda Issue 2019] and the check was not implemented in full until the recent version 2.6.2 of Agda [Agda Issue 2020].

native instance. A potential problem is code duplication, on which we take a conservative position: while we cannot solve the problem, which is inherent in languages with datatype declarations, we do alleviate it a little by removing the overhead of manual instantiation and maintaining explicit connections between generic and instantiated entities, which generic libraries can exploit. A possible solution to the problem was proposed by Chapman et al. [2010] in the form of a more radically redesigned type theory where datatype declarations are replaced with first-class descriptions, and the μ operator becomes the exclusive built-in mechanism for manufacturing datatypes. Generic programs in this theory are directly computable and do not require instantiation. However, there have been no subsequent developments of the theory, in particular a practical implementation. While waiting for better languages to emerge, it is also important to enable the development and practical use of datatype-generic libraries as soon as possible, which our framework does.

Foundations. While the use of universe polymorphic-datatypes is convenient in practice, its theory is, surprisingly, an unexplored territory, and the notion of universe polymorphism is less studied than it should be. To the best of our efforts, we did not find any type theory backing Agda's current design of first-class universe levels and universe polymorphism. Even worse, without universe cumulativity, subject reduction does not hold for expressions in Set_ω [Agda Issue 2022]. Kovács [2022] initiated a theoretical framework for first-class universe levels which is able to model features like bounded universe polymorphism, but its metatheory is bare-bones and lacks, for example, an elaboration algorithm needed for implementation. A type theory of first-class datatypes [Chapman et al. 2010] requires internal universe-polymorphic encodings, but most of internal encodings [Dybjer and Setzer 2006; Nordvall Forsberg 2013; Kaposi and Kovács 2020] do not encode universe polymorphism.

Due to the nature of uni-typed encodings, our experience with Agda's elaborator reflection was painful, especially in contrast to datatype-generic programming. Elaborator reflection is a useful paradigm but it has not been specified in theory, and to use it the understanding of the inner workings of elaborator is much needed. More importantly, the correctness of a macro can only be verified externally at best, but it is internally guaranteed by typed metaprogramming like staging. Christiansen and Brady [2016] argued that programs in the reflected elaborator are shorter and simpler than typed metaprograms, but they also admitted the additional cost is for the mandatory correctness. Hopefully, the best of two worlds could be combined. Say, suppose that we have a type $\mathsf{TTerm}\ A$ of A-typed reflected expressions. Normalisation (by evaluation) always works on well-typed expressions, so the type of normalise could be $\mathsf{TTerm}\ A \to \mathsf{TTerm}\ A$; type checking transforms a possibly ill-formed expression to a typed expression if successful, so the type of checkType could be $\mathsf{Term} \to (A : \mathsf{Set}\ \ell) \to \mathsf{TC}\ (\mathsf{TTerm}\ A)$. Typed reflected expressions also benefit efficiency, since they need not be elaborated again.

We hope that our framework can serve as inspiration and a call for a foundation for universes and metaprogramming not only for theoretical interests but also for practical needs.

ACKNOWLEDGMENTS

The work is supported by the Ministry of Science and Technology of Taiwan under grant MOST 109-2222-E-001-002-MY3.

REFERENCES

Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. 2018. Introduction to Bidirectional Transformations. In *International Summer School on Bidirectional Transformations (SSBX) 2016.* Lecture Notes in Computer Science, Vol. 9715. Springer, Chapter 1, 1–28. https://doi.org/10.1007/978-3-319-79108-1_1

Agda Issue. 2019. Wrong de Bruijn Indices for Reflected Variables inside an Extended Context. https://github.com/agda/agda/issues/3831 Accessed: 2022-03-02.

Agda Issue. 2020. Get Rid of UNREACHABLE on Escaping Variable. https://github.com/agda/agda/pull/4898 Accessed: 2022-03-02.

Agda Issue. 2022. Loss of Subject Reduction with Setω. https://github.com/agda/agda/issues/5810 Accessed: 2022-03-02.

The Agda Team. 2022a. Agda 2.6.2.1 Documentation. https://agda.readthedocs.io/en/v2.6.2.1/. Accessed: 2022-02-23.

The Agda Team. 2022b. The Agda Standard Library (Version 1.7.1). https://github.com/agda/agda-stdlib/tree/v1.7.1. Accessed: 2022-02-27.

Artem Alimarine and Sjaak Smetsers. 2004. Optimizing Generic Functions. In *International Conference on Mathematics of Program Construction (MPC) (Lecture Notes in Computer Science, Vol. 3125)*. Springer, 16–31. https://doi.org/10.1007/978-3-540-27764-4_3

Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2021. A Type- and Scope-Safe Universe of Syntaxes with Binding: Their Semantics and Proofs. *Journal of Functional Programming* 31 (2021), e22:1–51. https://doi.org/10.1017/S0956796820000076

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning Distributed Representations of Code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 40:1–29. https://doi.org/10.1145/3290353

Thorsten Altenkirch and Conor McBride. 2003. Generic Programming within Dependently Typed Programming. In *Generic Programming (IFIP — The International Federation for Information Processing, Vol. 115)*. Springer, 1–20. https://doi.org/10.1007/978-0-387-35672-3_1

Thorsten Altenkirch, Conor McBride, and Peter Morris. 2007. Generic Programming with Dependent Types. In *International Spring School on Datatype-Generic Programming (SSDGP) 2006*. Lecture Notes in Computer Science, Vol. 4719. Springer, 209–257. https://doi.org/10.1007/978-3-540-76786-2_4

Marcin Benke, Peter Dybjer, and Patrik Jansson. 2003. Universes for Generic Programs and Proofs in Dependent Type Theory. *Nordic Journal of Computing* 10, 4 (2003), 265–289. https://www.mimuw.edu.pl/~ben/Papers/universes.pdf

Richard Bird and Oege de Moor. 1997. *Algebra of Programming*. Prentice-Hall.

Edwin Brady. 2017. *Type-Driven Development with Idris*. Manning Publications.

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. 2010. The Gentle Art of Levitation. In *International Conference on Functional Programming (ICFP)*. ACM, 3–14. https://doi.org/10.1145/1863543.1863547

Liang-Ting Chen. 2019. Monadic Typed Tactic Programming by Reflection (Extended Abstract). In *Workshop on Type-Driven Development (TyDe)*. https://tydeworkshop.org/2019-abstracts/paper20.pdf

David Christiansen and Edwin Brady. 2016. Elaborator Reflection: Extending Idris in Idris. In *International Conference on Functional Programming (ICFP)*. ACM, 284–297. https://doi.org/10.1145/3022670.2951932

Jesper Cockx and Andreas M. Abel. 2020. Elaborating Dependent (Co)pattern Matching: No Pattern Left Behind. *Journal of Functional Programming* 30 (2020), e2:1–43. https://doi.org/10.1017/S0956796819000182

Jesper Cockx, Dominique Devriese, and Frank Piessens. 2016. Eliminating Dependent Pattern Matching without K. *Journal of Functional Programming* 26 (2016), e16:1–40. https://doi.org/10.1017/S0956796816000174

Pierre-Évariste Dagand and Conor McBride. 2014. Transporting Functions across Ornaments. *Journal of Functional Programming* 24, 2–3 (2014), 316–383. https://doi.org/10.1017/S0956796814000069

N. G. de Bruijn. 1991. Telescopic Mappings in Typed Lambda Calculus. *Information and Computation* 91, 2 (1991), 189–204. https://doi.org/10.1016/0890-5401(91)90066-b

Edsko de Vries and Andres Löh. 2014. True Sums of Products. In *Workshop on Generic Programming (WGP)*. ACM, 83–94. https://doi.org/10.1145/2633628.2633634

Martijn de Vries. 2004. *Specializing Type-Indexed Values by Partial Evaluation*. Master's thesis. University of Groningen. https://fse.studenttheses.ub.rug.nl/8943/

Dominique Devriese and Frank Piessens. 2011. On the Bright Side of Type Classes: Instance Arguments in Agda. In *International Conference on Functional Programming (ICFP)*. ACM, 143–155. https://doi.org/10.1145/2034773.2034796

Larry Diehl and Tim Sheard. 2016. Generic Lookup and Update for Infinitary Inductive-Recursive Types. In *Workshop on Type-Driven Development (TyDe)*. ACM, 1–12. https://doi.org/10.1145/2976022.2976031

Peter Dybjer. 1994. Inductive Families. *Formal Aspects of Computing* 6, 4 (1994), 440–465. https://doi.org/10.1007/BF01211308

Peter Dybjer and Anton Setzer. 2006. Indexed Induction-Recursion. *Journal of Logic and Algebraic Programming* 66, 1 (2006), 1–49. https://doi.org/10.1016/j.jlap.2005.07.001

Andrzej Filinski. 1999. A Semantic Account of Type-Directed Partial Evaluation. In *International Symposium on Principles and Practice of Declarative Programming (PPDP)*. Springer, 378–395. https://doi.org/10.1007/10704567_23

Jeremy Gibbons. 2007. Datatype-Generic Programming. In *International Spring School on Datatype-Generic Programming (SSDGP) 2006*. Lecture Notes in Computer Science, Vol. 4719. Springer, 1–71. https://doi.org/10.1007/978-3-540-76786-2_1

Robert Harper, Furio Honsell, and Gordon D. Plotkin. 1993. A Framework for Defining Logics. *Journal of the ACM* 40, 1 (1993), 143–184. https://doi.org/10.1145/138027.138060

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall. https://www.itu.dk/people/sestoft/pebook/

Ambrus Kaposi and András Kovács. 2020. Signatures and Induction Principles for Higher Inductive-Inductive Types. *Logical Methods in Computer Science* 16, 1 (2020), 10:1–30. https://doi.org/10.23638/LMCS-16(1:10)2020

Hsiang-Shang Ko. 2021. Programming Metamorphic Algorithms: An Experiment in Type-driven Algorithm Design. *The Art, Science, and Engineering of Programming* 5, 2 (2021), 7:1–34. https://doi.org/10.22152/programming-journal.org/2021/5/7

Hsiang-Shang Ko and Jeremy Gibbons. 2013. Modularising Inductive Families. *Progress in Informatics* 10 (2013), 65–88. https://doi.org/10.2201/NiiPi.2013.10.5

Hsiang-Shang Ko and Jeremy Gibbons. 2017. Programming with Ornaments. *Journal of Functional Programming* 27 (2017), e2:1–43. https://doi.org/10.1017/S0956796816000307

Wen Kokke, Philip Wadler, and Jeremy G. Siek. 2020. Programming Language Foundations in Agda (Version 20.07). http://plfa.inf.ed.ac.uk/20.07/

András Kovács. 2022. Generalized Universe Hierarchies and First-Class Universe Levels. In *Conference on Computer Science Logic (CSL) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 216)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 28:1–17. https://doi.org/10.4230/LIPIcs.CSL.2022.28

Andres Löh. 2004. *Exploring Generic Haskell*. Ph. D. Dissertation. Utrecht University. https://www.andres-loeh.de/ExploringGH.pdf

José Pedro Magalhães. 2012. *Less Is More: Generic Programming Theory and Practice*. Ph. D. Dissertation. Utrecht University. https://dreixel.net/research/pdf/thesis.pdf

José Pedro Magalhães. 2013. Optimisation of Generic Programs through Inlining. In *Symposium on Implementation and Application of Functional Languages (IFL) (Lecture Notes in Computer Science, Vol. 8241)*. Springer, 104–121. https://doi.org/10.1007/978-3-642-41582-1_7

José Pedro Magalhães and Andres Löh. 2014. Generic Generic Programming. In *International Symposium on Practical Aspects of Declarative Languages (PADL)*. Springer, 216–231. https://doi.org/10.1007/978-3-319-04132-2_15

Per Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium '73 (Studies in Logic and the Foundations of Mathematics, Vol. 80)*. Elsevier, 73–118. https://doi.org/10.1016/S0049-237X(08)71945-1

Per Martin-Löf. 1984. *Intuitionistic Type Theory*. Bibliopolis, Napoli.

Conor McBride. 2011. Ornamental Algebras, Algebraic Ornaments. https://personal.cis.strath.ac.uk/conor.mcbride/pub/OAAO/LitOrn.pdf

Conor McBride. 2014. How to Keep Your Neighbours in Order. In *International Conference on Functional Programming (ICFP)*. ACM, 297–309. https://doi.org/10.1145/2628136.2628163

Trevor L. McDonell, Timothy A. K. Zakian, Matteo Cimini, and Ryan R. Newton. 2016. Ghostbuster: A Tool for Simplifying and Converting GADTs. In *International Conference on Functional Programming (ICFP)*. ACM, 338–350. https://doi.org/10.1145/3022670.2951914

Aleksandar Nanevski and Frank Pfenning. 2005. Staged Computation with Names and Necessity. *Journal of Functional Programming* 15, 6 (2005), 893–939. https://doi.org/10.1017/S095679680500568X

Fredrik Nordvall Forsberg. 2013. *Inductive-inductive definitions*. Ph. D. Dissertation. Swansea University.

Matthew Pickering, Gergo Gergő Érdi, Simon Peyton Jones, and Richard A. Eisenberg. 2016. Pattern Synonyms. In *International Symposium on Haskell*. ACM, 80–91. https://doi.org/10.1145/2976002.2976013

Matthew Pickering, Andres Löh, and Nicolas Wu. 2020. Staged Sums of Products. In *International Symposium on Haskell*. ACM, 122–135. https://doi.org/10.1145/3406088.3409021

Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2019. Ornaments for Proof Reuse in Coq. In *International Conference on Interactive Theorem Proving (ITP) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 141)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 26:1–19. https://doi.org/10.4230/LIPIcs.ITP.2019.26

Carsten Schürmann, Adam Poswolsky, and Jeffrey Sarnat. 2005. The ∇-Calculus. Functional Programming with Higher-Order Encodings. In *International Conference on Typed Lambda Calculi and Applications (TLCA) (Lecture Notes in Computer Science, Vol. 3461)*. Springer. https://doi.org/10.1007/11417170_2

Aaron Stump. 2016. *Verified Functional Programming in Agda*. ACM Books. https://doi.org/10.1145/2841316

Thomas Williams and Didier Rémy. 2018. A Principled Approach to Ornamentation in ML. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 21:1–30. https://doi.org/10.1145/3158109

Jeremy Yallop. 2017. Staged Generic Programming. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 29:1–29. https://doi.org/10.1145/3110273

Zhixuan Yang and Nicolas Wu. 2022. Fantastic Morphisms and Where to Find Them: A Guide to Recursion Schemes. arXiv:2202.13633.

Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. 2015. Mtac: A Monad for Typed Tactic Programming in Coq. *Journal of Functional Programming* 25 (2015), e12:1–59. https://doi.org/10.1017/S0956796815000118