Datatype-Generic Programming Meets Elaborator Reflection

HSIANG-SHANG KO, LIANG-TING CHEN, and TZU-CHI LIN, 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. Based on elaborator reflection, we present a framework in Agda featuring a set of general metaprograms for instantiating datatype-generic programs as, and for, a useful range of native datatypes and functions —including universe-polymorphic ones— in programmer-friendly and customisable forms. We expect that datatype-generic libraries built with our framework will be more attractive to the practical Agda programmer. As the elaborator reflection features used by our framework become more widespread, our design can be ported to other languages too.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Data types and structures**.

Additional Key Words and Phrases: datatype-generic programming, dependently typed programming, inductive families, universe polymorphism, elaborator reflection, metaprogramming

ACM Reference Format:
Hsiang-Shang Ko, Liang-Ting Chen, and Tzu-Chi Lin. 2022. Datatype-Generic Programming Meets Elaborator Reflection. *Proc. ACM Program. Lang.* 6, ICFP, Article 98 (August 2022), 29 pages. https://doi.org/10.1145/3547629

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?

One major problem, we argue, is the lack of interoperability. The prevalent approach to datatype-generic programming in Agda 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

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.

2475-1421/2022/8-ART98
https://doi.org/10.1145/3547629

Section 7.3.1), 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

```
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)
```

Note that both foldr and foldr-fusion are *fully universe-polymorphic* (otherwise they would be inconvenient to use). Also important (especially in a dependently typed setting) is the ability to derive new datatypes — the standard example is the derivation of vectors from list length,

```
data Vec (A : Set ℓ) : ℕ → Set ℓ where            length : {A : Set ℓ} → List A → ℕ
  []  :                         Vec A zero         length []       = zero
  _∷_ : A → ∀ {n} → Vec A n → Vec A (suc n)        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 current dependently typed languages though.

Luckily again, in Agda there is a mechanism 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, in addition to the usual metaprogramming features such as quoting and unquoting. In fact, elaborator reflection is powerful enough for the metaprogrammer to develop general facilities for practical datatype-generic programming. Like in Haskell, we can quote programmer-defined datatypes as descriptions for processing by generic programs; conversely, newly computed descriptions can be unquoted as programmer-friendly datatypes rather than decoded using a fixed-point operator, and functions can be defined by unquoting too. Besides these standard tasks, elaborator reflection achieves more and serves our purpose nicely: To express dependency in types, descriptions are usually higher-order and can be difficult to manipulate, but our metaprogram that manufactures datatypes from descriptions 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 elaborator reflection primitives that interact with the context during type checking. Moreover, we can specialise generic programs to more efficient forms straightforwardly by using (open-term) normalisation —also an elaborator reflection primitive— to perform some of the computation early during elaboration. (By contrast, to achieve a similar effect with staging, it would be necessary to add annotations to generic programs, making generic libraries harder to develop.)

Based on elaborator reflection, 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 in programmer-friendly and customisable forms. Central to the framework is a set of general metaprograms performing transformations between native and generic entities, for example deriving descriptions from datatypes and manufacturing datatypes from descriptions. These metaprograms are general in the sense that they are decoupled from generic libraries, and can be independently maintained and widely reused; the decoupling also allows generic libraries to be developed or adapted from old ones in largely the same, traditional way, without having to deal with native entities themselves. To interface with the metaprograms, generic libraries should (either directly or indirectly) target the datatype descriptions and function representations provided by our framework, which are expressive: we support inductive families [Dybjer 1994] and both fold and inductive functions, all of which can be parametrised and universe-polymorphic.

We expect that datatype-generic libraries built with our framework will be more attractive to the practical Agda programmer. As the elaborator reflection features used by our framework become more widespread, our design will be portable to other languages too (in particular dependently typed ones) — for example, if a metaprogramming system includes a normalisation operation (like Idris), it will immediately gain the ability to optimise definitions. Moreover, Agda's currently unique design of universe polymorphism —where universe levels are made explicit and first-class— plays an important role in our universe-polymorphic datatype descriptions, and our work serves as a practical justification for further investigation into such design [Kovács 2022].

For the rest of the paper: After recapping standard datatype-generic programming (Section 2) and refining and replacing some definitions for our framework (Section 3), we present some of our metaprograms that showcase the power of elaborator reflection (Section 4). To simplify the presentation, up to this point we assume Set : Set and introduce only a slimmed-down version of our framework. Then, leaving Set : Set behind, we sketch how the full framework supports universe polymorphism (Section 5), and give a demo of the framework using some existing generic constructions (Section 6). Finally we conclude with some discussions (Section 7). Our code is available on Zenodo at https://doi.org/10.5281/zenodo.6603498.

2 A RECAP OF DATATYPE-GENERIC PROGRAMMING

We start from a recap of standard datatype-generic programming (in a dependently typed setting). The core idea of datatype-genericity is to encode datatype definitions as *descriptions*, which can take a variety of forms but should be some kind of first-class data on which computation can be

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

performed. Programs that manipulate datatype descriptions in some way are dubbed '(datatype-) generic programs', and can perform constructions tailored for individual datatypes by analysing input descriptions or produce new datatypes by computing output descriptions. The power of generic programs depends crucially on the range of datatypes encoded by the chosen descriptions. In Section 2.1 we fix on a class of descriptions covering inductive families [Dybjer 1994] in the form of sums of products. 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).

2.1 Datatype Descriptions

There have been quite a few variants of datatype descriptions [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 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 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. (Generic programs can then perform constructions depending on the number of constructors, the types of fields, the indices of recursive occurrences, etc.) For example, 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 σ, or describe a recursive occurrence, marked by ρ, and the 'lists' end with ι. Different from ordinary lists, in the case of $\sigma\ A\ D$ 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.[1] The ι at the end

[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 4.2).

```
fmapCs : (D : ConDs I) → (∀ {i} → X i → Y i) → ∀ {i} → ⟦D⟧Cs X i → ⟦D⟧Cs Y i
fmapCs (D :: Ds) f (inl xs)  = inl (fmapC  D f xs)
fmapCs (D :: Ds) f (inr xs)  = inr (fmapCs Ds f xs)

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

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

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

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:

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

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

A description D : ConDs I is converted to a datatype $\mu\ D : I \to \mathrm{Set}$ for day-to-day programming by taking the least fixed point of the base functor $[\![D]\!]_{\mathrm{Cs}} : (I \to \mathrm{Set}) \to (I \to \mathrm{Set})$:

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

For example, we can redefine $\mathrm{Acc}_<$ as $\mu\ \mathrm{Acc}_<\mathrm{D} : \mathbb{N} \to \mathrm{Set}$, whose inhabitants are now constructed by the generic constructor **con**. Specified by the definition of the base functor $[\![D]\!]_{\mathrm{Cs}}$ 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**,

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

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 $\mu\ \mathrm{Acc}_<\mathrm{D}\ n$. 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 $\mathrm{Acc}_<$, for example its fold operator:

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

[2] $\bot$ is the empty type with no constructors. $A \uplus B$ is the sum of the types A and B with constructors **inl** : $A \to A \uplus B$ and **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 $[\![\_]\!]_{\mathrm{Cs}}$, $[\![\_]\!]_{\mathrm{C}}$, and $[\![\_]\!]_{\mathrm{R}}$) are implicitly universally quantified.

However, the point of using decoded datatypes such as μ Acc$_<$D is that we do not have to write foldAcc$_<$ ourselves but can simply derive it as an instantiation of a generic grogram. The class of 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 $[\![D]\!]_{\mathrm{Cs}}$ in this paper), whose type is defined by

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

where the result type X is traditionally called the 'carrier' of the algebra. Algebras are the interesting part of a 'fold function', by which 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 $[\![D]\!]_{\mathrm{Cs}}\ X\ i$ represents exactly the input of step (iii). We can define a generic fold operator that expresses the computation pattern of fold functions,

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

where fmap$_{\mathrm{Cs}}$ is the functorial map for $[\![D]\!]_{\mathrm{Cs}}$ (defined in Figure 2),[3] used here to apply fold D f to the recursive fields in ds.[4] 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} (p : ∀ 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} (p : ∀ n → (∀ m → m < n → P m) → P n) →
           ∀ {n} → Acc< n → P n
foldAcc< p = fold Acc<D (foldAcc<Alg p)
```

Being able to treat folds generically means that we can write generic programs whose types have the form ∀ {i} → μ D i → X i, but this is not enough when, for example, we want to prove generic theorems by induction on d : μ D i, in which case the types take the more complex form ∀ {i} (d : μ D i) → P d (where P : ∀ {i} → μ D i → 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 4) work for fold and induction in the same way.

[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$_{\mathrm{Cs}}$ keeps the choices of **inl** or **inr**, fmap$_{\mathrm{C}}$ keeps the σ-fields and ι-equalities, and finally fmap$_{\mathrm{R}}$ applies f to the recursive fields (of type X i for some i) pointwise.
[4]Agda's termination checker cannot verify that this generic fold operator is terminating, hence the unsafe TERMINATING pragma. This is not a problem for us because instead of using this fold operator, we will only manufacture fold functions on specific datatypes such as foldAcc$_<$, which do pass the termination check.

```
mutual
  data Tel : Set where                              ⟦_⟧ᴛ : Tel → Set
    []    : Tel                                     ⟦[]      ⟧ᴛ = ⊤
    _::_  : (A : Set) (T :   A     → Tel) → Tel     ⟦A :: T ⟧ᴛ = Σ[ a :   A    ] ⟦T a⟧ᴛ
    _++_  : (T : Tel) (U : ⟦T⟧ᴛ → Tel) → Tel        ⟦T ++ U⟧ᴛ = Σ[ t : ⟦T⟧ᴛ] ⟦U t⟧ᴛ
```

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

3 PARAMETERS AND CONNECTIONS

To make our framework expressive enough for practical use, we need a couple more features. The first feature is datatype parameters. For example, it was probably tempting to generalise $\mathsf{Acc}_<$ (Section 2.1) to a version parametrised by a type A and a binary relation R on A:

```
data Acc {A : Set} (R : A → A → Set) : A → Set where
  acc : (x : A) → ((y : A) → R y x → Acc R y) → Acc R x
```

The second feature is a data structure —which we call datatype *connections*— that abstracts and replaces the generic μ operator (although similar structures have long been present in the Haskell tradition of datatype-generic programming). Here is a motivating example: From a description D, we will use a metaprogram to manufacture a native datatype N (in place of $\mu\ D$). 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} (P : A → Set) : List A → Set where
  []   :                                               ListAll P []
  _::_ : ∀ {a} → P a → ∀ {as} → ListAll P as → ListAll P (a :: as)
```

whose description can be computed from the description of List. If μ were in use, then N would simply be $\mu\ D$, whose constructor would also be known to be **con**; without μ, however, N and its constructors have to be provided as additional input to the generic construction of All to allow the latter to specialise to the description of ListAll. In general, datatype connections capture the extensional behaviour of datatypes generically such that what we can do with connections are more or less the same as what we can do with those datatypes manufactured with μ. Thus, by replacing μ with datatype connections, we can easily adapt generic programs that assume the presence of μ to work on native datatypes instead. The extensionality also allows us to customise the forms of native datatypes flexibly as long as they still behave the same.

After treating datatype parameters and connections respectively in Sections 3.1 and 3.2, we will also define parametrised algebras and their connections with fold functions in Section 3.3. In this section it may appear that, to use our framework, the programmer needs to write the additional components such as descriptions and connections, which are usually tedious, but keep in mind that we will be able to generate them automatically by metaprograms (Section 4).

3.1 Parametrised Datatype Descriptions and Telescopes

It is conceptually straightforward to encode a parametrised datatype, since parameters are just variables in the context which can be referred to by the index type and constructor types, and we know an easy way to extend the context — just use the host language's function space. So, traditionally, a parametrised datatype could be encoded by a parameter type $P : \mathsf{Set}$, a parametrised index type $I : P \to \mathsf{Set}$, and a parametrised list of constructor descriptions $(p : P) \to \mathsf{ConDs}\ (I\ p)$. For example, we could encode Acc with $P = \Sigma[A : \mathsf{Set}]\ (A \to A \to \mathsf{Set})$, $I = \lambda\ (A\ ,\ \_) \to A$, and a parametrised description that looks like $\mathsf{Acc}_<\mathsf{D}$.

```
Curried_T : (T : Tel) → (⟦ T ⟧_T → Set) → Set
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)))
```

Fig. 4. Curried function types from telescopes

```
record PDataD : Set where field
  Param  : Tel
  Index  : ⟦ Param ⟧_T → Tel
  applyP : (ps : ⟦ Param ⟧_T) →
           ConDs ⟦ Index ps ⟧_T
```

```
⟦_⟧_PD : (D : PDataD) → ∀ {ps} →
         let I = ⟦ D .Index ps ⟧_T in (I → Set) → (I → Set)
⟦ D ⟧_PD {ps} = ⟦ D .applyP ps ⟧_Cs

fmap_PD : (D : PDataD) → ∀ {ps} {X Y : ⟦ D .Index ps ⟧_T → Set} →
          (∀ {i} → X i → Y i) → ∀ {i} → ⟦ D ⟧_PD X i → ⟦ D ⟧_PD Y i
fmap_PD D {ps} = fmap_Cs (D .applyP ps)
```

Fig. 5. Parametrised datatype descriptions and lifted base functors

The encoding above works well in principle, but there is one refinement that can make the encoding work better in practice. We will eventually need to convert a description to a datatype, and it would be unsatisfactory if the parameter and index types in the datatype were not in the conventional curried form. To make this currying easier, we introduce *telescopes* [de Bruijn 1991] to represent lists of parameter or index types, as shown in Figure 3; also shown is the semantics of a telescope $\llbracket T \rrbracket_{\mathsf{T}}$, which is a nested Σ-type inhabited by tuples whose components have the types in T.[5] 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. This allows us to combine tuples $t : \llbracket T \rrbracket_{\mathsf{T}}$ and $u : \llbracket U\ t \rrbracket_{\mathsf{T}}$ directly into $(t\ ,\ u) : \llbracket T \mathbin{+\!\!+} U \rrbracket_{\mathsf{T}}$, from which we can still easily retrieve t and u because we did not insist on flattening $T \mathbin{+\!\!+} U$ (to use only '::') and re-associating the possibly deeply nested tuple (t, u) to the right — we will see how this structure is useful to generic libraries when we reach Section 6.1. A couple of syntax declarations will make telescopes slightly easier to write and read:

```
syntax _::_ A (λ x → T) = [ x : A ] T ;    syntax _++_ T (λ t → U) = [[ t : T ]] U
```

For example, the parameters of Acc can be written as $[A : \mathsf{Set}]\ [R : (A \to A \to \mathsf{Set})]\ []$ instead of $\mathsf{Set} :: (\lambda\ A \to (A \to A \to \mathsf{Set}) :: (\lambda\ R \to []))$. From a telescope T it is straightforward to compute a curried function type $\mathsf{Curried}_{\mathsf{T}}\ T\ X$ (Figure 4) which has arguments with the types in T, and ends with a given type $X : \llbracket T \rrbracket_{\mathsf{T}} \to \mathsf{Set}$ that can refer to all the arguments (collectively represented as a tuple of type $\llbracket T \rrbracket_{\mathsf{T}}$). 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 from the presentation).

With telescopes, we can now define the type PDataD of parametrised datatype descriptions in Figure 5, refining the parameter and index sets P and I into telescopes. For example, the Acc datatype can now be described by

```
AccD : PDataD
AccD = record
  { Param  = [ A : Set ] [ R : (A → A → Set) ] []
  ; 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))) :: [] }
```

[5] ⊤ is the unit type with one constructor **tt**.

```
PDataT : PDataD → Set
PDataT D = ∀ ps (is : ⟦D .Index ps⟧T) → Set

record PDataC (D : PDataD) (N : PDataT D) : Set where field
  toN        : ⟦D⟧PD (N ps) is → N ps is
  fromN      : N ps is          → ⟦D⟧PD (N ps) is
  fromN-toN  : (ns : ⟦D⟧PD (N ps) is) → fromN (toN ns) ≡ ns
  toN-fromN  : (n  : N ps is)           → toN (fromN n)  ≡ n
```

Fig. 6. Datatype connections

When accessing the fields in the PDataD structures, the postfix projection syntax works better, as shown in the definitions of base functor lifted pointwise to the PDataD layer in Figure 5 (which we will use later).

3.2 Datatype Connections

When μ was present, generic programs only needed to take a description *D* : PDataD 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 PDataT *D* as defined in Figure 6. Regardless of how many parameters and indices there actually are, this uncurried form always represents parameters and indices as two arguments *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 : PDataT 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 PDataC of datatype *connections* in Figure 6, replacing the functionalities of μ. (A fine difference between PDataC and μ is that 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 PDataC *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 : PDataC 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 here.

The introduction of PDataC 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 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

```
record PFoldP : Set where field
  {Desc}   : PDataD
  {Native} : PDataT Desc
  Con      : PDataC Desc Native
  Param    : Tel
  param    : ⟦Param⟧T → ⟦Desc .Param⟧T
  Carrier  : ∀ ps → ⟦Desc .Index (param ps)⟧T → Set
  applyP   : ∀ ps → Alg (Desc .applyP (param ps))
                        (Carrier ps)
```

```
PFoldT : PFoldP → Set
PFoldT F = ∀ ps {is} →
  let open PFoldP F in Native (param ps) is → Carrier ps is
record PFoldC (F : PFoldP) (f : PFoldT F) : Set where field
  equation :
    ∀ {ps is} → let open PFoldP F in
    (ns : ⟦Desc⟧PD (Native (param ps)) is) →
    f ps (Con .toN ns) ≡ applyP ps (fmapPD Desc (f ps) ns)
```

Fig. 7. Parametrised fold programs (algebras) and connections

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.

3.3 Parametrised Fold Programs and Fold Connections

Like descriptions, algebras can also be parametrised — in fact, foldAcc$_<$Alg is Section 2 is already an algebra with two parameters P and p. Analogous to PDataD (Figure 5), we use the type PFoldP (for 'parametrised fold programs') defined in Figure 7 to store an algebra with its parameter telescope and carrier. There are some more fields that require explanation: PFoldP 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 PFoldP includes a field Con : PDataC Desc Native 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 PFoldT in Figure 7.[6] What remains to be explained is the field param, which, as can be seen in the definition of PFoldT, is used to compute the parameters for the native datatype from the parameters of the fold function. For example, the fold operator of Acc

```
foldAcc : {A : Set} {R : A → A → Set} {P : A → Set}
          (p : ∀ x → (∀ y → R y x → P y) → P x) → ∀ {x} → Acc R x → P x
foldAcc p (acc x as) = p x (λ y lt → foldAcc p (as y lt))
```

is encoded as the fold program (which ignores parameter visibility)

```
foldAccP : PFoldP
foldAccP = record
  {Con     = AccC
  ; Param  = [A : Set] [R : (A → A → Set)] [P : (A → Set)]
               [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}}
```

Following the same architecture for datatypes, we are also going to connect algebras with native fold functions. Here a fold function f : PFoldT F corresponding to some F : PFoldP is supposed to replace an instantiation of the generic fold operator using the algebra in F, so what we need to

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

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 PFoldC in Figure 7. Proofs of the equation are usually by definition — for example, foldAccP and foldAcc are connected by

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

where foldAccT $(A , R , P , p , \_)$ = foldAcc p is a wrapper function.

Although we do not present the details of generic induction, the definitions are largely the same as those for folds, 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.

4 ESTABLISHING CONNECTIONS USING ELABORATOR REFLECTION

As explained at the end of Section 3.2, our framework can be thought of as keeping native and generic entities 'in sync' through connections. This syncing can be tedious: whenever we write a native datatype N, we need to produce its description D, a wrapper T around N, and a connection between D and T; conversely, whenever we compute a new datatype description, we also need to produce the corresponding native datatype, wrapper, and connection; and the same goes for instantiating generic programs as native functions. Fortunately, such tasks can be automated by a set of metaprograms supplied by our framework.

Our metaprograms are based on Agda's elaborator reflection (Section 4.1), which provides a few important features that make some of the tasks much easier to accomplish than with traditional metaprogramming. We will focus on two particularly noteworthy examples (Sections 4.2 and 4.3): The first is the translation from our datatype descriptions —which use a higher-order representation of binders— to the elaborator reflection API's first-order representations —which use de Bruijn indices— for generating a native datatype. Surprisingly, we are able to avoid the tedious and error-prone manipulation of de Bruijn indices using the 'local variable creation' technique [Nanevski and Pfenning 2005; Schürmann et al. 2005], which is easily supported by elaborator reflection. The second is the instantiation of fold programs as native fold functions. The elaborator reflection API exposes open-term normalisation as a primitive, which we can directly use for non-trivial computation such as expanding generic definitions of function types and partially evaluating function bodies, without having to implement heavyweight term transformations in the metaprogram.

4.1 Elaborator Reflection in Agda

The elaborator reflection API is based on a set of datatypes reflecting Agda's core language. Among these datatypes, the one we will see most frequently is Term, the datatype of first-order representations of core expressions. The quotation of an expression e can be obtained as **quoteTerm** e : Term; this syntax makes it easy to produce concrete examples of Term — for example, we may evaluate **quoteTerm** $((A$: Set$)$ → Vec A **zero**$)$ and get[7]

```
pi (agda-sort (lit 0)) (def (quote Vec) (var 0 [] :: con (quote zero) [] :: [])) : Term
```

where the structure of the type expression is turned into the composition of several of the Term constructors, namely

- **pi** : Term → Term → Term, which represents a dependent function type,
- **agda-sort** : Sort → Term, where Sort is the datatype representing sorts such as Set and Set ℓ,

[7]In fact there is some additional information embedded in Terms such as argument visibility and binder names, which we suppress in our presentation for brevity.

- **def** : Name → List Term → Term and **con** : Name → List Term → Term, representing the application of a top-level definition or constructor —referred to by a quoted name of the form **quote** n : Name in this example— to a list of arguments,
- **var** : $\mathbb{N}$ → List Term → Term, which is similar to **def** and **con** except that the first argument is a variable in the form of a de Bruijn index.

The central component of elaborator reflection is the elaborator monad TC, short for 'Type Checking', which stores states needed for elaboration such as the context of the call site, the scope of names with its definition, and the set of metavariables. Metaprograms take the form of TC computations, and have access to a set of primitive operations used during elaboration — for example, the primitive unify : Term → Term → TC ⊤ unifies two given terms and solves some of the metavariables (thereby changing the elaborator state).

At elaboration time, we can run a metaprogram and splice an expression into the source code. A more convenient way to do so is to use macros, a special kind of metaprograms of type A_1 → ⋯ → A_n → Term → TC ⊤ declared with the keyword **macro** and called with the first n arguments. During elaboration, the call site of a macro M becomes a metavariable x, which is represented as **meta** x [] : Term and supplied as the last argument of M for manipulation inside M. A minimal example is

macro give = unify

which is called with one argument of type Term. Elaborating give e will splice the given expression e in place of the call: If a macro argument has type Term or Name, the expression supplied for the argument in a macro call will be automatically quoted by **quoteTerm** or **quote**, 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.

Another way to use a metaprogram is to compute a top-level function by

unquoteDecl f = ⋯

where introduces the function name f into the scope. The right-hand side invokes a metaprogram, which needs to take the name f as an argument, so that —analogously to how we write top-level functions by hand— it can declare the type of f using declareDef : Name → Term → TC ⊤ and give the definition of f using defineFun : Name → List Clause → TC ⊤ (where Clause is the datatype of reflected function clauses).

We have extended the **unquoteDecl** mechanism to allow metaprograms to define datatypes as well. The extended syntax is

unquoteDecl data d **constructor** c_1 … c_n = ⋯

which introduces the names of a datatype d and its constructors $c_1, \ldots, c_n$ into the scope. The definitions of d and $c_1, \ldots, c_n$ are supplied by the metaprogram using the new primitives declareData and defineData (whose details are omitted from the presentation).

4.2 Translating Higher-Order Representations with Local Variable Creation

Our first task is to translate PDataD, a fully typed higher-order representation, into the reflected language to define native datatypes. The reflected language is, by contrast, a uni-typed first-order representation using de Bruijn indices and *not* hygienic, 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 → A)] [] of type Tel should be translated to this flattened list of reflected types

```
‘Set ∷ pi (var 0 []) (var 1 []) ∷ [] : Telescope                                    (1)
```

where the type Telescope = List Term is a part of the elaborator reflection API (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 the input Tel. 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 analyse the input Tel by pattern matching. The case for the empty telescope [] is simple to define but the other two cases A ∷ T and U ++ V seem impossible:

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

Note that T is a function from A, and V a function from $⟦ U ⟧_T$, 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 : Term → TC A → TC A. The first argument of extendContext is a reflected type, which should be the quotation of the actual value of A, but this value 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 the 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) → (Term → 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. This function can be generalised to extend the context with a telescope:

```
exCxtTel : (T : Tel) (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))
```

As for $⟦ U ⟧_T$, if we merely created a local variable $u : ⟦ U ⟧_T$, then each reference to a component of u would be formed by projections fst and snd. For example, instead of (1) we would have

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

To eliminate projections, we use exCxtTel to create a list of local variables for each type in U : Tel as a tuple. The last two cases of fromTel can then be defined by

```
fromTel (A ∷ T)  =  do              fromTel (U ++ V)  =  do Γ ← fromTel U
  exCxtT A λ ‘A x → do                                    exCxtTel U λ u → do
    Γ ← fromTel (T x)                                       Δ ← fromTel (V u)
    return (‘A ∷ Γ)                                         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!

The approach scales well for our metaprogram that defines a native datatype from a description. Conversely, we also have a macro that expands to a description of a given native datatype. This direction is syntactical and unsurprising though — for example, telescopes are handled simply 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 :: []) (using **lam** : Term → Term to construct a reflected λ-expression).

4.3 Instantiating Fold Programs with Normalisation

Our second task is to write a metaprogram definePFold (Section 4.3.2) to manufacture a native fold function from a PFoldP. Before writing definePFold, we will work out a concrete example manually and develop some generic definitions (Section 4.3.1), which definePFold will use. We will not show the full detail of definePFold since it requires a more extensive understanding of the elaborator reflection API, but the metaprogram is essentially just a careful formalisation of the manual process.

4.3.1 Instantiation by Hand. As a concrete example, let us manufacture from the fold program foldAccP in Section 3.3 a variant of the fold operator for Acc (for demonstration purposes whose arguments are all made explicit to avoid the complication of handling argument visibility):

```
foldAcc′ : (A : Set) (R : A → A → Set) (P : A → Set)
           (p : ∀ x → (∀ y → R y x → P y) → P x) → ∀ x → Acc R x → P x
foldAcc′ A R P p .x (acc x as) = p x (λ y lt → foldAcc′ A R P p y (as y lt))
```

First we need a curried type for the fold function, which can be computed by a variant of PFoldT that uses Curried$_\mathsf{T}$ (Figure 4):

```
PFoldNT : PFoldP → Set
PFoldNT F = let open PFoldP F in
  Curried_T Param λ ps → Curried_T (Desc .Index (param ps)) λ is →
  Native (param ps) is → Carrier ps is
```

Simply normalising PFoldNT foldAccP gives the type of foldAcc′.

As for the definition of foldAcc′, it should satisfy the equation of PFoldC foldAccP foldAccT′ (Figure 7) —where foldAccT′ (A , R , P , p , _) {x , _} = foldAcc′ A R P p x is a wrapper function— but the 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 : PFoldP) → PFoldNT F → PFoldNT F
fold-base F rec =
  let open PFoldP F in curry_T λ ps → curry_T λ is →
  applyP ps ∘ fmap_PD Desc (λ {is} → uncurry_T (uncurry_T rec ps) is) ∘ Con .fromN
```

```
definePFold : PFoldP → Name → TC ⊤
definePFold F f = do
    -- step (i)
    `type ← normalise =<< quoteTC (PFoldNT F)
    declareDef f `type
    -- step (ii)
    cls ← caseM (getDefinition =<< PFoldPToNativeName F) of λ
            { (data-type pars cs) → forM cs (conClause pars)
            ; _                   → typeError [] }
    -- step (iii)
    defineFun f =<< mapRHS normalise cls
```

Fig. 8. The metaprogram definePFold for generating a top-level fold function from a fold program

This definition of foldAcc′, albeit one deemed non-terminating by Agda, implies the PFoldC.equation because of the inverse property PDataC.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:

$$\text{foldAcc}'\ A\ R\ P\ p\ .x\ (\textbf{acc}\ x\ as)\ =\ \text{fold-base foldAccP foldAcc}'\ A\ R\ P\ p\ x\ (\textbf{acc}\ x\ as) \tag{2}$$

Now normalise the right-hand side:

$$\text{foldAcc}'\ A\ R\ P\ p\ .x\ (\textbf{acc}\ x\ as)\ =\ p\ x\ (\lambda\ y\ lt\ \to\ \text{foldAcc}'\ A\ R\ P\ p\ y\ (as\ y\ lt))$$

In general, this final definition will pass the termination check if Con .fromN works normally by breaks its input into structurally smaller pieces, in which case the fmap_{PD} Desc part in fold-base applies *rec* to those smaller pieces. And the connecting equation always holds definitionally:

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

(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.)

4.3.2 Instantiation by a Metaprogram. Now we formalise the manual process above as a TC computation definePFold (Figure 8), which instantiates a given F : PFoldP as a native function f by (i) generating the instantiated type using PFoldNT, (ii) generating a clause for each constructor of the datatype specified in F, and (iii) normalising these clauses.

Step (i) is straightforward because the type of f can be obtained directly by normalising PFoldNT F. We use quoteTC to turn PFoldNT F into a Term, normalise it, and finally declare the normalised term as the type of f using declareDef.

For step (ii), we use the primitive getDefinition : Name → TC Definition (where Definition is the datatype of reflected definitions of datatypes, record types, functions, etc) to get the list *cs* of constructor names of the datatype F .Native, and generate a clause for each constructor. First we need to explain how clauses are handled in a bit more detail: A clause takes the form $\Delta \vdash \overline{p} \hookrightarrow e$ 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 unsolved metavariables represented by **unknown** : Term.

Back to the clauses we should generate for f: Abstracted from (2), each clause has the form

$$\Delta \vdash \overline{p}\ \overline{x}\ (c_i\ \overline{a}) \hookrightarrow \text{fold-base}\ F\ f\ \overline{p}\ \overline{x}\ (c_i\ \overline{a}) \tag{3}$$

where $\overline{p}$ the 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 (3) 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 the right-hand side of (3), 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

$$\overline{p}\ \overline{.\textbf{unknown}}\ (c_i\ \overline{a}) \hookrightarrow \mathsf{fold\text{-}base}\ F\ f\ \overline{p}\ \overline{\textbf{unknown}}\ (c_i\ \overline{a}) \tag{4}$$

for each constructor c_i.

For step (iii), we normalise the right-hand side of (4) within a context extended with variables from the left-hand side of (4) and obtain the list of clauses with normalised expressions on the right-hand side as the result of mapRHS normalise *cls*. Finally, we define f by the primitive defineFun, supplying the clauses we just constructed.

Following the same approach to generating functions, the remaining components of our framework, namely PDataT, PDataC, PFoldT, and PFoldC, can also be automatically generated from native datatypes and functions by metaprograms.

5 UNIVERSE POLYMORPHISM

We have assumed a single universe Set satisfying Set : Set in our presentation so far, but it is well known that this assumption leads to logical inconsistency, to avoid which we must use a hierarchy of universes $\mathsf{Set}_0 : \mathsf{Set}_1 : \mathsf{Set}_2 : \cdots$ instead [Martin-Löf 1975]. General library components are usually reusable across the hierarchy — for example, the most general form of Acc is

```
data Acc {ℓ ℓ′ : Level} {A : Set ℓ} (R : A → A → Set ℓ′) : A → Set (ℓ ⊔ ℓ′) where
  acc : (x : A) → ((y : A) → R y x → Acc R y) → Acc R x
```

where the universe levels ℓ and ℓ' in the types of A and R can be arbitrary. Our framework ought to support such *universe-polymorphic* datatypes given their prevalence in Agda.

Agda supports universe polymorphism through a currently unique system where (finite) universe levels are made explicit and first-class by giving them a type Level : Set. Levels can be constructed with the primitives lzero : Level and lsuc : Level → Level in the same way as natural numbers are constructed, but we cannot pattern-match levels with lzero and lsuc. There is also an operator _⊔_ : Level → Level → Level that computes the maximum of two levels. First-class levels make it convenient to encode universe-polymorphic datatypes — just put levels inside datatype descriptions. It is also possible to compute new datatypes where the universe levels are the results of non-trivial computation, which can be reasoned about. Besides showing that it is possible to deal with universe-polymorphic entities generically, our encoding in this section also serves as a practical justification of (Agda's) first-class universe levels, as we make essential use of the capabilities of computing and reasoning about levels internally.

Our framework makes a simplifying assumption that holds for common universe-polymorphic datatypes (for example Acc above): we assume that there is a list of level parameters separate from other ordinary parameters, and only these level parameters are involved in universe polymorphism. Under this assumption, to describe a possibly universe-polymorphic datatype, we start with a number $n : \mathbb{N}$ of level parameters, from which we can compute a type Level ^ n of tuples of n levels as defined by A ^ **zero** = ⊤ and A ^ (**suc** n) = A × (A ^ n), and then provide a function of type Level ^ n → PDataD, which brings n level parameters into the scope of the definition of a PDataD. We create a new description layer DataD for level parametrisation, which is shown in Figure 9 along with the existing four layers adapted to accommodate levels, to be explained

```
record DataD : Setω where field
  #levels : ℕ
  applyL  : Level ^ #levels → PDataD
record PDataD : Set ω 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 → Set ω where
  []   :                                ConDs I []
  _::_ : ConD I cb → ConDs I cbs → ConDs I (cb :: cbs)
data ConD (I : Set ℓI) : ConB → Set ω 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 → Set ω where
  ι : I                              → RecD I []
  π : (A : Set ℓ) → (A → RecD I rb) → RecD I (ℓ :: rb)
```

Fig. 9. Universe-polymorphic datatype descriptions (all five layers), where RecB = List Level, ConB = List (Level ⊎ RecB), and ConBs = List ConB (Modifications to Figures 1 and 5 are highlighted.)

```
mutual
  data Tel : Level → Set ω where                       ⟦_⟧T : Tel ℓ → Set ℓ
    []   : Tel lzero                                   ⟦[]     ⟧T = ⊤
    _::_ : (A : Set ℓ) (T : A → Tel ℓ′) → Tel (ℓ ⊔ ℓ′)  ⟦A :: T⟧T = Σ[a : A ] ⟦T a⟧T
    _++_ : (T : Tel ℓ) (U : ⟦T⟧T → Tel ℓ′) → Tel (ℓ ⊔ ℓ′) ⟦T ++ U⟧T = Σ[t : ⟦T⟧T] ⟦U t⟧T
```

Fig. 10. Universe-polymorphic telescopes (Modifications to Figure 3 are highlighted.)

below in Sections 5.1 to 5.3. The rest of our framework are similarly adapted, so there are also definitions of DataT, DataC, FoldP, FoldT, and FoldC, which we omit from the presentation. Finally, the metaprograms in Section 4 are extended to treat level parameters, but due to the current limit of Agda's universe polymorphism, the treatment is different from that of ordinary parameters, and is briefly sketched in Section 5.4.

5.1 Telescopes

Before we adapt the descriptions, telescopes need to be adapted too, as shown in Figure 10. The core change is made to the type of the first argument of '::', from Set to Set ℓ. We also add an index of type Level to Tel such that T : Tel ℓ implies that the maximum level appearing in T is ℓ; this maximum level is important since it is the universe level of the type $[\![T]\!]_{\mathsf{T}}$. Since the elements of a telescope can now be Sets of arbitrary finite levels, the level of the type of the telescope itself has to be greater than all those levels, and is set as the first infinite level ω here.

5.2 Sum-of-Products Descriptions

Next we adapt the description datatypes ConDs, ConD, and RecD. 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 $[\![\_]\!]_{\mathsf{T}}$ and Curried$_{\mathsf{T}}$) is limited and requires only the computation of the maximum level, so indexing suffices; on the other hand, generic libraries may construct whatever they want from descriptions, and the need for non-trivial level computation will naturally arise if those constructions are universe-polymorphic — 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 $[\![D]\!]_{\mathsf{Cs}}$ should be enriched: One place where we use the base functor is the type of an algebra $\forall\, \{i\} \rightarrow [\![D]\!]_{\mathsf{Cs}}\ X\ i \rightarrow X\ i$ where $X : I \rightarrow$ Set ℓ 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 (Figure 9). 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 Figure 1. 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        hasCon? : Level → ConBs → Level
maxMap f []        = lzero                   hasCon? ℓ = maxMap (λ _ → ℓ)
maxMap f (a :: as) = f a ⊔ maxMap f as
```

It is easy to see that the output level in the $[\![\,[\,]\,]\!]_{\mathsf{Cs}}$ case is lzero, which is indeed the level of ⊥. 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.

5.3 Ordinary Parameters

The changes to PDataD (Figure 9) should be mostly unsurprising except the new 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 (Section 2), so even the traditional approach to datatype-genericity could be extended to incorporate universe polymorphism. In general, the ability to manipulate and reason about levels internally is probably crucial to datatype-genericity, because computation of universe-polymorphic datatype descriptions —in particular the levels in the descriptions— can be arbitrarily complex, and it may no longer be feasible to infer levels or check level constraints automatically as can be done for specific datatypes in languages with typical ambiguity [Sozeau and Tabareau 2014].

5.4 Level Parameters

Conceptually, level parameters could be treated in the same way as ordinary ones — in particular, Level ^ *n* could be thought of as a kind of specialised telescope, so we could start with a construction similar to Curried_T (Figure 4), computing a curried function type with *n* level quantifications:

```
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))
```

However, the hole '?' is problematic 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, but currently Agda's universe polymorphism supports only finite levels. To produce curried types with level quantifications, we have to operate at the meta-level (evading the type checker) and generate the level quantifications in their syntax trees in relevant metaprograms such as defineFold, of which definePFold (Figure 8) is a cut-down version. The need for the special treatment is one reason that we separate level parameters from ordinary ones instead of allowing the two kinds of parameters to mix freely.

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 possible 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 : 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 from Acc its description AccD, a wrapper AccT, and a datatype connection AccC between AccD and AccT by some macros:

```
AccD = genDataD Acc ;   AccT = genDataT AccD Acc ;   AccC = genDataC AccD AccT
```

We can then use the connection AccC to instantiate fold-operator to the fold program

```
foldAccP = fold-operator AccC
```

from which the fold operator/function foldAcc can be manufactured by using the **unquoteDecl** syntax (Section 4.1) to introduce the name foldAcc into the scope, and then invoking the metaprogram defineFold with the fold program and the newly introduced name:

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

Finally, we generate a fold connection between foldAccP and foldAcc by

```
foldAccC = genFoldC foldAccP foldAcc
```

which can be used to instantiate other generic programs (such as fold-fusion below) for foldAcc. The macros are manually invoked for now and might be invoked with wrong arguments, but usually the error messages are reasonable because the macros are typed — for example, checking genDataC AccD Acc will give rise to a standard error message saying that Acc does not have the type DataT AccD. (Perhaps the instantiation process could eventually be streamlined as, say, pressing a few buttons of an editor, which displays only valid options.)

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*. Suppose that we want to instantiate the theorem for the foldr operator on List in Section 1. Note that this version of foldr is different from the one manufactured by defineFold from the fold program foldListP = fold-operator ListC, where ListC is the datatype connection for List; in particular, the arguments of foldr are in a different order from that specified by foldListP. However, we can still connect foldr with foldListP by manually writing a wrapper to specify the argument order,

```
foldrT : FoldT foldListP
foldrT (ℓ′ , ℓ , _) ((A , _) , B , e , f , _) = foldr f e
```

and then generating a fold connection (using a variant of genFoldC that takes a wrapper)

```
foldrC = genFoldC′ foldListP foldrT
```

Now we can obtain foldr-fusion in Section 1 by

```
unquoteDecl foldr-fusion = defineInd (fold-fusion ListC foldrC) foldr-fusion
```

and use foldr-fusion to prove, for example, the handy law that a map followed by a foldr (two list traversals) can be optimised as a foldr (a single list traversal),

```
foldr-map-fusion : {A : Set ℓ} {B : Set ℓ′} {C : Set ℓ″}
                   (f : B → C → C) (e : C) (g : A → B) →
                   (as : List A) → foldr f e (map g as) ≡ foldr (f ∘ g) e as
foldr-map-fusion f e g = foldr-fusion (foldr f e) refl (λ {_ _ _ refl → refl})
```

where map *g* = foldr [] (_::_ ∘ *g*). (The law could also be proved generically, but first we would need to give map a generic definition, which requires extra machinery not developed in this paper.)

In general, if the library user is not satisfied with the form of a manufactured function or datatype (argument order, visibility, etc), they can print the definition, change it to a form they want, and connect the customised version back to the library in the same way as how 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 for functions where argument name suggestions (for definition printing) and visibility specifications are included in FoldP (and IndP) and processed by relevant components such as Curried$_\mathrm{T}$, 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.

Let us switch to the perspective of the library programmer and look at fold-operator in a bit more detail. The ordinary parameters of fold-operator C are mainly a $[\![D]\!]_\mathrm{D}$-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 3.2): 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.

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 ℕ using genDataD, the following ornament states that List —having an additional element field— is a more informative version of 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

```
forget : DataC D M → DataC E N → DataO D E → FoldP
```

which instantiates to a forgetful function from M to N — if we instantiate forget to lenP = forget ListC NatC ListO where ListC and NatC are connections for List and ℕ respectively, the function manufactured from lenP will be list length (Section 1), which discards the list elements.

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 manufactured from 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 (Section 1), which is a standard example of algebraic ornamentation. To manufacture Vec (modulo constructor naming and argument visibility), we use the **unquoteDecl data** syntax (Section 4.1) to introduce the names of the datatype and constructors into the scope, and invoke the metaprogram defineByDataD with the description VecD = AlgD lenP, the datatype name, and the list of the constructor names:

```
unquoteDecl data Vec constructor []V _::V_ = defineByDataD VecD Vec ([]V :: _::V_ :: [])
```

(This form of declaration is somewhat verbose but works as a temporary solution that avoids drastic changes to the internals of Agda.) Then we generate a datatype connection between VecD and (a wrapper around) Vec so that we can instantiate generic programs for Vec later:

```
VecC = genDataC VecD (genDataT VecD Vec)
```

From the algebraic ornament VecO = AlgO lenP between Vec and List, in addition to a forgetful function fromV instantiated from forget VecC ListC VecO, 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 Σ ℕ (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              data _⊢_ : List Ty → Ty → Set where
  var : ℕ → Λ                     var : Γ ∋ τ → Γ ⊢ τ
  app : Λ → Λ → Λ                 app : Γ ⊢ σ ⇒ τ → Γ ⊢ σ → Γ ⊢ τ
  lam : Λ → Λ                     lam : σ :: Γ ⊢ τ → Γ ⊢ σ ⇒ τ
data Ty : Set where             data _⊢_:_ : List Ty → Λ → Ty → Set where
  base : Ty                       var : (i : Γ ∋ τ) → Γ ⊢ var (toℕ i) : τ
  _⇒_ : Ty → Ty → Ty              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

```
SC : DataD → Setω                              record SCP (D : PDataD) : Setω where field
SC D = ∀ {ℓs} → SCP (D .applyL ℓs)                {elevel} : Level
                                                  El   : ⟦D .Param⟧T → Set elevel
SCB : ConB → Set                                  pos : ListAll SCB (D .struct)
SCB = ListAll (λ { (inl _) → Bool                 coe : (ps : ⟦D .Param⟧T) → SCCs (D .applyP ps) pos (El ps)
                 ; (inr _) → ⊤ })

SCCs : {I : Set ℓI} → ConDs I cbs → ListAll SCB cbs → Set ℓ → Setω
SCCs []       _        X = ⊤
SCCs (D :: Ds) (s :: ss) X = SCC D s E × SCCs Ds ss X

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

Fig. 11. The simple-container predicate on datatype descriptions

'_⊢_:_' 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 omitted the types of the generic programs related to algebraic ornamentation because they are somewhat verbose, making generic program invocation less cost-effective — for example, the generic programs proving the inverse properties need the 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 tested 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 — for example, to-fromV can be derived by supplying just the names Vec and List. However, instance searching currently brings serious performance overhead, and the solution still requires us to instantiate one definition at a time. 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. In this case the datatype computation is not difficult, but it still makes sense to reason that the computed levels will always pass the universe check, and Agda conveniently allows us to perform the reasoning internally.

6.3 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 3) is an instance of All.

The definition of simple containers (in several layers) is shown in Figure 11. The top layer SC on DataD only quantifies over the level parameters, and the main definition is at the next layer SCᴘ 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, 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 SCᴘ makes sure that the types of the fields marked with **true** are equal to El; subsequently, when a generic program encounters 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 a promotion isomorphism (Section 6.2) that allows us to convert between ListWP P (a list of pairs of an element and a proof) and Σ (List A) (ListAll P) (a pair of a list of elements and a list of proofs).

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 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 ℕ, allowing us to derive a forgetful function toℕ 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

7.1 Efficiency

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 metaprogram define(P)Fold (Section 4.3) is a partial evaluator that specialises a generic program (general program) to a given description (known input). Indeed, it has been observed that we can perform partial evaluation in functional languages by normalisation [Filinski 1999], which define(P)Fold 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 not only puts burdens on the programmer but also 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.

While programs instantiated using our framework are optimised, there is still the overhead of type-checking the additional definitions required by our framework and running the metaprograms to generate code. Minimising this overhead is particularly important for an interactive development environment (which Agda is famous for). We performed a quick experiment comparing two versions of all our examples: the first version used our metaprograms to instantiate the generic programs, whereas the second version contained printed (hand-written) definitions only, without anything from our framework. The result was disappointing: on a typical laptop, it took about 66 seconds to check the first version, and about 5 seconds to check the second version. We think that the experiment is more an indication of how much the current implementation of Agda's type checker and elaborator reflection (and, to a lesser extent, our metaprograms) can still be optimised, since the overhead inherently associated with our core idea —instantiating generic programs with normalisation and unquotation— should not be so high (and our metaprograms are just straightforward implementations of the idea). But however high the overhead is, it can always be alleviated by separate compilation —collecting generic entities in separate files, which are checked only once and do not interfere with subsequent interactive development sessions— or copying and pasting printed definitions into the source files. These solutions require the programmer's effort, but as generic libraries become more powerful and the user interface gets improved, the cost will be outweighed by the benefit of not having to write the definitions that can generated.

7.2 Dependently Typed Datatype-Generic Libraries

Our framework makes it possible and worthwhile to develop dependently typed datatype-generic libraries for wider and practical use in Agda (and other languages when the framework is ported there). There are still many opportunities to explore for such 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. Derived datatypes, such as the lifted predicates All and Any in Section 6.3, are also common and should be treated generically, as opposed to duplicating an instance for each datatype as in the standard library (version 1.7.1 at the time of writing). 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; 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].

The change to traditional generic programs required by our framework is a mild generalisation from the operators μ and fold (Section 2) to datatype and fold connections capturing the behaviour of the operators, so it is easy to adapt existing generic libraries (as well as develop new ones). To interface with our metaprograms, the generic-library developer can reimplement their libraries on our descriptions, or translate their descriptions to ours — in Haskell, Magalhães and Löh [2014] provide automatic conversions between the datatype representations of several generic libraries and a representative representation, through which native datatypes are connected to all the libraries at once; our descriptions can serve as a representative representation.

Our framework also helps with backward compatibility when existing non-generic library components can be derived from some new generic constructions. Rather than replacing those components with **unquoteDecl** definitions, we can simply supply connections for the components (like how we treated foldr in Section 6.1); in this way, existing code is not broken, and the library user only needs to understand the new generic constructions when starting to use them.

7.3 Foundations

7.3.1 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 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, which our framework does.

7.3.2 Typed Metaprogramming. One important issue that we have not been able to address fully is the correctness of our metaprograms. Agda's syntax and semantics are somewhat complicated and already make it hard for the metaprogrammer to consider all possible scenarios. Even worse, since elaborator reflection has no formal specification and the reflected representations are only uni-typed, the metaprogrammer gets little help from the documentation or the type checker. To get metaprograms right, the metaprogrammer needs to tweak the metaprograms based on their understandings of the elaborator's inner workings, and perform extensive testing. Correctness arguments can only be given informally (like in Section 4.3.1).

In contrast to our experience with Agda's elaborator reflection, which was painful, our datatype-generic programming experience is more pleasant overall. Datatype-generic programming can be seen as a form of typed metaprogramming because datatype-generic programs manipulate datatype representations just like metaprograms, with a notable difference being that the representations are precisely typed. Christiansen and Brady [2016] argued that their metaprograms were shorter and simpler because correctness proofs were not mandatory, but we find that correctness proofs —especially those implicitly embedded in precisely typed data— lead to a smoother development process, notably without having to rely on testing for correctness guarantees.

One way for elaborator reflection to offer better correctness guarantees about meta-level constructions is to introduce more precise types, and (dependently typed) datatype-generic programming already provides a working solution for typing a good range of the constructions. Our work can thus be regarded as bringing in a more precisely typed alternative to some of the uni-typed reflected representations, allowing the metaprogrammer to gain better correctness guarantees for some of their constructions. A possible next step is to give more precise types to the elaborator reflection API itself: Say, suppose that we have a type TTerm A of A-typed reflected expressions. Normalisation (by evaluation) always works on well-typed expressions, so the type of normalise could be TTerm $A \rightarrow$ TTerm A. Type checking transforms a possibly ill-formed expression to a typed expression if successful, so the type of checkType could be Term $\rightarrow (A : \mathsf{Set}\ \ell) \rightarrow$ TC (TTerm A). Typed reflected expressions also benefit efficiency, since they need not be elaborated again.

7.3.3 Universe Polymorphism. While universe polymorphism is a convenient feature in practice, it is ignored by most of the existing datatype encodings [Dybjer and Setzer 2006; Chapman et al. 2010; Nordvall Forsberg 2013; Kaposi and Kovács 2020]. Our universe-polymorphic datatype descriptions are made possible by Agda's first-class universe levels, which allow us to express non-trivial level computation and guarantee level-correctness just like we can guarantee the correctness of typed metaprograms without testing or checking the generated entities. On the other hand, theoretically there does not seem to be any type theory backing Agda's design, and some problems have already surfaced — for example, currently subject reduction does not hold for expressions in Set_ω without universe cumulativity [Agda Issue 2022]. Kovács [2022] initiated a model-theoretic study of first-class universe levels, including features such as bounded universe polymorphism, but the metatheory is bare-bones and lacks, for example, an elaboration algorithm needed for implementation, so there is still a significant gap between theory and practice.

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

ACKNOWLEDGMENTS

We thank the anonymous reviewers and shepherd for their great help in improving the presentation of this paper. 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. 2022. Loss of Subject Reduction with Setω. https://github.com/agda/agda/issues/5810 Accessed: 2022-03-02.

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

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 Conference on Principles and Practice of Declarative Programming (PPDP) (Lecture Notes in Computer Science, Vol. 1702)*. 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

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) (Lecture Notes in Computer Science, Vol. 8324)*. 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

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 für 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

Matthieu Sozeau and Nicolas Tabareau. 2014. Universe Polymorphism in Coq. In *International Conference on Interactive Theorem Proving (ITP) (Lecture Notes in Computer Science, Vol. 8558)*. Springer, 499–514. https://doi.org/10.1007/978-3-319-08970-6_32

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. To appear at the *International Conference on Mathematics of Program Construction (MPC)*. https://arxiv.org/abs/2202.13633