A Formal Comparison of Approaches to Datatype-Generic Programming

José Pedro Magalhães*
Utrecht University
jpm@cs.uu.nl

Andres Löh
Well-Typed LLP
andres@well-typed.com

Datatype-generic programming increases program abstraction and reuse by making functions operate uniformly across different types. Many approaches to generic programming have been proposed over the years, most of them for Haskell, but recently also for dependently typed languages such as Agda. Different approaches vary in expressiveness, ease of use, and implementation techniques.

Some work has been done in comparing the different approaches informally. However, to our knowledge there have been no attempts to formally prove relations between different approaches. We thus present a formal comparison of generic programming libraries. We show how to formalise different approaches in Agda, including a coinductive representation, and then establish theorems that relate the approaches to each other. We provide constructive proofs of inclusion of one approach in another that can be used to convert between approaches, helping to reduce code duplication across different libraries. Our formalisation also helps in providing a clear picture of the potential of each approach, especially in relating different generic views and their expressiveness.

1 Introduction

There are many forms of genericity [8]. Of particular interest to us is *datatype-genericity*: behavior that is generic over the structure of types. Functional programming languages typically support the definition of algebraic datatypes. Due to their algebraic nature, these datatypes can be conveniently encoded in a sum-of-products structure. Many functions can be defined to operate on this structure alone; typical examples are (de)serialisation, traversals, equality, and enumeration.

Datatype-generic programming (from here on referred to simply as generic programming) approaches have been especially prolific in Haskell [24]. `PolyP` [14], now 15 years old, was the first approach to generic programming in Haskell, implemented as a pre-processor. Since then, and especially with the advent of advanced type features such as generalized algebraic datatypes (GADTs) and type families [28], numerous other approaches have appeared, most of them implemented directly as a library. This abundance is caused by the lack of a clearly superior approach; each approach has its strengths and weaknesses, uses different implementation mechanisms, a different generic view [13] (i.e. a different structural representation of datatypes), or specialises on a particular task. Their sheer number and variety makes comparisons difficult, and can make prospective generic programming users struggle even before actually writing a generic program, since first they have to choose a library that is adequate to their needs.

Some effort has been made in comparing different approaches to generic programming from a practical point of view [10,26], or to classify approaches [11]. While Generic Haskell [15] has been formalised in different attempts [27, 29], no formal comparison between modern approaches has been attempted, leaving a gap in the knowledge of the relationships between each approach. We argue that this gap

*Funded by the Portuguese Foundation for Science and Technology (FCT) via the SFRH/BD/35999/2007 grant. We thank Johan Jeuring and the anonymous reviewers for the helpful feedback.

J. Chapman and P. B. Levy (Eds.): Fourth Workshop on Mathematically Structured Functional Programming (MSFP 2012).
EPTCS 76, 2012, pp. 50–67, doi:10.4204/EPTCS.76.6

should be filled; for starters, a formal comparison provides a theoretical foundation for understanding different generic programming approaches and their differences and similarities. However, the contribution is not merely of theoretical interest, since a comparison can also provide means for converting between approaches. Ironically, code duplication across generic programming libraries is evident: the same function can be nearly identical in different approaches, yet impossible to reuse, due to the underlying differences in representation. With a formal proof of inclusion between two approaches a conversion function can be derived, removing the duplication of generic code.

In this paper we take the initial steps towards a formal comparison of generic programming approaches:

- We encode five distinct, yet related generic programming libraries in the dependently-typed programming language Agda [23].
- We encode one specific Haskell library using recursive codes, by means of a coinductive formulation. We pay special attention to this approach, as it also gives rise to more challenging proofs.
- We show the relations between the approaches, and reason about them. While the inclusion relations are the expected, the way to convert between approaches is often far from straightforward, and reveals subtle differences between the approaches. Each inclusion is evidenced by a conversion function that brings codes from one universe into another, enabling generic function reuse across different approaches.
- Although fully machine-checked (modulo non-termination), our proofs are in equational reasoning style and resemble handwritten proofs, remaining clear and elegant.

The rest of this paper proceeds as follows: we first introduce the five approaches we compare by showing their encoding in Agda in Section 2. We show the inclusion relations between these approaches in Section 3, and focus on the details of some particular proofs. Finally we conclude in Section 4, discussing shortcomings of our work and providing directions for future research.

2 Generic programming libraries

In this section we introduce each of the five libraries that we model. We model all the libraries in Agda, and reduce them to their essence, namely to the generic view they encode. We leave out details such as implementation mechanisms (e.g. through type classes or GADTs), encoding of meta-information such as constructor names (it is generally an issue orthogonal to the library), or modularity.

All five libraries we choose use a sum-of-products representation of data. One does not use fixed points, while the other four use each a different form of fixed-point operator. For the latter we show how to define a mapping function, which can be used to define all the standard recursion morphisms [19], and are also necessary for the conversion proofs. We leave the comparison of libraries using a view other than sum-of-products for future work (Section 4). Nonetheless, the libraries we choose are rather different in their expressiveness, especially regarding support for parametrised datatypes and families of mutually recursive types, as we will show.

2.1 Regular

`Regular` is a simple generic programming library, originally written to support a generic rewriting system [22]. It has a fixed-point view on data: the generic representation is a pattern-functor, and a fixed-point operator ties the recursion explicitly. In the original formulation, this is used to ensure that rewriting meta-variables can only occur at recursive positions of the datatype.

We model every library by defining a Code type that represents the generic universe, and its interpretation function $[\![\_]\!]$ that maps the codes into Agda types. The universe, interpretation, and fixed-point operator for `Regular` follow:

```
data Code : Set where                    [[_]] : Code → (Set → Set)
  U   :                  Code            [[ U     ]] A = ⊤
  I   :                  Code            [[ I     ]] A = A
  _⊕_ : (F G : Code) → Code              [[ F ⊕ G ]] A = [[ F ]] A ⊎ [[ G ]] A
  _⊗_ : (F G : Code) → Code              [[ F ⊗ G ]] A = [[ F ]] A × [[ G ]] A

data μ (F : Code) : Set where ⟨_⟩ : [[ F ]] (μ F) → μ F
```

We have codes for units, sums, products, and recursive positions, denoted by I. The interpretation of unit, sum, and product relies on the Agda types for unit ($\top$), disjoint sum ($\_\uplus\_$), and non-dependent product ($\_\times\_$), respectively. The interpretation is parametrised over a Set that is returned in the I case.

Libraries with a fixed-point view on data allow defining a map function. In `Regular`, this function lifts a transformation between sets A and B to a transformation between interpretations parametrised over A and B, simply by applying the function in the I case:

```
map : (F : Code) → {A B : Set} → (A → B) → [[ F ]] A → [[ F ]] B
map U       f _       = tt
map I       f x       = f x
map (F ⊕ G) f (inj₁ x) = inj₁ (map F f x)
map (F ⊕ G) f (inj₂ x) = inj₂ (map G f x)
map (F ⊗ G) f (x , y)  = map F f x , map G f y
```

The map function can be used to define many other useful generic functions, most notably recursion morphisms [19]. For example, catamorphisms are defined as follows:

```
cata : {A : Set} → (F : Code) → ([[ F ]] A → A) → μ F → A
cata C f ⟨ x ⟩ = f (map C (cata C f) x)
```

Datatypes can be encoded by giving their code, such as NatC for the natural numbers, and then taking the fixed point. Hence, a natural number is a value of type μ NatC; in the example below, aNat encodes the number 2:

```
NatC : Code
NatC = U ⊕ I

aNat : μ NatC
aNat = ⟨ inj₂ ⟨ inj₂ ⟨ inj₁ tt ⟩ ⟩ ⟩
```

2.2 PolyP

`PolyP` [14] is an early pre-processor approach to generic programming. However, this aspect is not essential to `PolyP`, as it still works with an underlying view of Haskell datatypes. This view is very similar to that of `Regular`, only that it also abstracts over one datatype parameter, in addition to one recursive

occurrence. `PolyP` therefore represents types as bifunctors, whereas `Regular` uses plain functors. The encoding of `PolyP`'s universe in Agda follows:

```
data Code : Set where
  U    :                    Code
  P    :                    Code
  I    :                    Code
  _⊕_  : (F G : Code) → Code
  _⊗_  : (F G : Code) → Code
  _◎_  : (F G : Code) → Code

⟦_⟧ : Code → (Set → Set → Set)
⟦ U     ⟧ A R = ⊤
⟦ P     ⟧ A R = A
⟦ I     ⟧ A R = R
⟦ F ⊕ G ⟧ A R = ⟦ F ⟧ A R ⊎ ⟦ G ⟧ A R
⟦ F ⊗ G ⟧ A R = ⟦ F ⟧ A R × ⟦ G ⟧ A R
⟦ F ◎ G ⟧ A R = μ F (⟦ G ⟧ A R)

data μ (F : Code) (A : Set) : Set where ⟨_⟩ : ⟦ F ⟧ A (μ F A) → μ F A
```

In the codes, the only differences from `Regular` are the addition of a `P` code, for the parameter, and a code `_◎_` for composition. The interpretation is parametrised over two `Set`s, one for the parameter and the other for the recursive position. Composition is interpreted by taking the fixed-point of the left bifunctor, thereby closing its recursion, and replacing its parameters by the interpretation of the right bifunctor. There is at least one other plausible interpretation for composition, namely interpreting the left bifunctor with closed right bifunctors as parameter (`⟦ F ⟧ (μ G A) R`), but we give the interpretation taken by the original implementation.

This asymmetric treatment of the parameters to composition is worth a detailed discussion. In `PolyP`, the left functor `F` is first closed under recursion, and its parameter is set to be the interpretation of the right functor `G`. The parameter `A` is used in the interpretation of `G`, as is the recursive position `R`. Care must be taken when using composition to keep in mind the way it is interpreted. For instance, if we have a code for binary trees with elements at the leaves `TreeC`, and a code for lists `ListC`, one might naively think that the code for trees with lists at the leaves is `TreeC ◎ ListC`, but that is not the case. Instead, the code we are after is `(ListC ◎ P) ⊕ (I ⊗ I)`. Apart from requiring careful usage, this composition does not allow us to reuse the code for trees when defining trees with lists (although the resulting code quite resembles that of trees). The `Indexed` approach (described in Section 2.4) has a more convenient interpretation of composition; this subtle difference is revealed explicitly in our conversion from `PolyP` to `Indexed` (Section 3.2).

The fixed-point operator takes a bifunctor and produces a functor, by closing the recursive positions and leaving the parameter open. The `map` operation for bifunctors takes two argument functions, one to apply to parameters, and the other to apply to recursive positions:

```
map : {A B R S : Set} (F : Code) → (A → B) → (R → S) → ⟦ F ⟧ A R → ⟦ F ⟧ B S
map U       f g _       = tt
map P       f g x       = f x
map I       f g x       = g x
map (F ⊕ G) f g (inj₁ x) = inj₁ (map F f g x)
map (F ⊕ G) f g (inj₂ x) = inj₂ (map G f g x)
map (F ⊗ G) f g (x , y)  = map F f g x , map G f g y
map (F ◎ G) f g ⟨ x ⟩    = ⟨ map F (map G f g) (map (F ◎ G) f g) x ⟩
```

A map over the parameters, `pmap`, operating on fixed points of bifunctors, can be built from `map` trivially:

```
pmap : {A B : Set} (F : Code) → (A → B) → μ F A → μ F B
pmap F f ⟨ x ⟩ = ⟨ map F f (pmap F f) x ⟩
```

As an example encoding in `PolyP` we show the type of non-empty rose trees:

```
ListC : Code                    sRose : μ RoseC ⊤
ListC = U ⊕ (P ⊗ I)             sRose = ⟨ tt, ⟨ inj₁ tt ⟩ ⟩
RoseC : Code                    lRose : μ RoseC ⊤
RoseC = P ⊗ (ListC ◎ I)         lRose = ⟨ tt, ⟨ inj₂ (sRose, ⟨ inj₂ (sRose, ⟨ inj₁ tt ⟩) ⟩) ⟩ ⟩
```

We first encode lists in ListC; rose trees are a parameter and a list containing more rose trees (RoseC). The smallest possible rose tree is sRose, containing a single element and an empty list. A larger tree lRose contains a parameter and a list with two small rose trees.

2.3 Multirec

The `MultiRec` library [25] is also a generalisation of `Regular`, allowing for multiple recursive positions instead of only one. This means that families of mutually recursive datatypes can be encoded in `MultiRec`. For this, types are represented as higher-order (or indexed) functors:

```
Indexed : Set → Set₁
Indexed I = I → Set
```

Codes themselves are parametrised over an index Set, that is used in the I case. Furthermore, we have a new code ! for tagging a code with a particular index. The interpretation is parametrised by a function r that maps indices (recursive positions) to Sets, and a specific index i that defines which particular position we are interested in; since a code can define several types, the interpretation is a function from the index of a particular type to its interpretation. For an occurrence of an index I, we retrieve the associated set using the function r. Tagging constrains the interpretation to a particular index j, so we check if j is the same as i:

```
data Code (I : Set) : Set where
  U   :                  Code I
  I   : I                → Code I
  !   : I                → Code I
  _⊕_ : (F G : Code I) → Code I
  _⊗_ : (F G : Code I) → Code I

⟦_⟧ : {I : Set} → Code I → Indexed I → Indexed I
⟦ U     ⟧ r i = ⊤
⟦ I j   ⟧ r i = r j
⟦ ! j   ⟧ r i = i ≡ j
⟦ F ⊕ G ⟧ r i = ⟦ F ⟧ r i ⊎ ⟦ G ⟧ r i
⟦ F ⊗ G ⟧ r i = ⟦ F ⟧ r i × ⟦ G ⟧ r i
```

Mapping is entirely similar to the `Regular` map, only that the function being mapped is now an index-preserving map. Similarly, the fixed-point operator is also indexed:

```
data μ {I : Set} (F : Code I) (i : I) : Set where ⟨_⟩ : ⟦ F ⟧ (μ F) i → μ F i

map : {I : Set} {R S : Indexed I} (F : Code I)
    → (∀ i → R i → S i) → (∀ i → ⟦ F ⟧ R i → ⟦ F ⟧ S i)
map U     f i _   = tt
map (I j) f i x   = f j x
```

```
map (! j)       f i x          = x
map (F ⊕ G) f i (inj₁ x) = inj₁ (map F f i x)
map (F ⊕ G) f i (inj₂ x) = inj₂ (map G f i x)
map (F ⊗ G) f i (x , y)  = map F f i x , map G f i y
```

To show an example involving mutually recursive types we encode a zig-zag sequence of even length. Consider first the family we wish to encode, inside a **mutual** block as the datatypes are mutually recursive:

```
mutual
  data Zig : Set where
    zig : Zag → Zig
    end : Zig
  data Zag : Set where
    zag : Zig → Zag
```

We can encode this family in `MultiRec` as follows:

```
ZigC : Code (⊤ ⊎ ⊤)
ZigC = I (inj₂ tt) ⊕ U
ZagC : Code (⊤ ⊎ ⊤)
ZagC = I (inj₁ tt)
ZigZagC : Code (⊤ ⊎ ⊤)
ZigZagC = (! (inj₁ tt) ⊗ ZigC) ⊕ (! (inj₂ tt) ⊗ ZagC)
zigZagEnd : μ ZigZagC (inj₁ tt)
zigZagEnd = ⟨ inj₁ (refl, inj₁ ⟨ inj₂ (refl, ⟨ inj₁ (refl, inj₂ tt) ⟩) ⟩) ⟩
```

zigZagEnd encodes the value zig (zag end), as its name suggests. Note how we define the code for each type in the family separately (ZigC and ZagC), and then a code ZigZagC for the family, encoding a tagged choice between the two types. As a consequence, proofs of index equality (witnessed by refl) are present throughout the encoded values.

2.4 Indexed Functors

Just like `MultiRec` can be seen as a generalisation of `Regular` to multiple recursive positions, the `Indexed` approach [16] can be seen as a generalisation of `PolyP` both to multiple recursive positions and multiple parameters. In `Indexed`, datatypes are represented by functors which are not only indexed on their input (like `MultiRec`) but also on their output. It is related to other approaches to dependently-typed generic programming, such as the levitating universe of Epigram 2 [5], or the theory of indexed containers [2].

The added complexity makes `Indexed` cumbersome to encode in Haskell, so its original description was in Agda (although recent developments in GHC's kind system [30] might now allow us to write a Haskell version of this library). Below we show a subset of its universe; we elide the reindexing, sigma, and isomorphism operators from the original presentation:

```
Indexed : Set → Set₁
Indexed I = I → Set

_|_ : ∀ {I J} → Indexed I → Indexed J
    → Indexed (I ⊎ J)
(r | s) (inj₁ i) = r i
(r | s) (inj₂ i) = s i

data Code (I O : Set) : Set₁ where
  U    :          Code I O
  I    : I  →     Code I O
  !    : O →      Code I O
  _⊕_  : (F G : Code I O) → Code I O
  _⊗_  : (F G : Code I O) → Code I O
  _◎_  : {M : Set} → (F : Code M O)
       → (G : Code I M) → Code I O
  Fix  : (F : Code (I ⊎ O) O) → Code I O

⟦_⟧ : {I O : Set} → Code I O
    → Indexed I → Indexed O
⟦ U     ⟧ r i = ⊤
⟦ I j   ⟧ r i = r j
⟦ ! j   ⟧ r i = i ≡ j
⟦ F ⊕ G ⟧ r i = ⟦ F ⟧ r i ⊎ ⟦ G ⟧ r i
⟦ F ⊗ G ⟧ r i = ⟦ F ⟧ r i × ⟦ G ⟧ r i
⟦ F ◎ G ⟧ r i = ⟦ F ⟧ (⟦ G ⟧ r) i
⟦ Fix F ⟧ r i = μ F r i

data μ {I O : Set} (F : Code (I ⊎ O) O) (r : Indexed I) (o : O) : Set where
  ⟨_⟩ : ⟦ F ⟧ (r | μ F r) o → μ F r o
```

A major difference from the previous approaches is that the fixed-point operator is contained within the universe. Composition is also allowed, and, unlike in `PolyP`, it does not require taking a fixed point. Codes are parametrised over two Sets, which can be thought of as the input set (parameters) and output set (types defined). Composition is only possible for codes with composable indices; for instance, a family of three types with two parameters can be composed with a family of two types with one parameter, resulting in the original family of three types, but now taking only one parameter. The fixed-point operator takes a code with tagged input indices (parameters on the left, recursive occurrences on the right) and closes the recursive occurrences, producing a code with only parameters as input.

The map function lifts a transformation $r \rightrightarrows s$ between indexed functors r and s to a transformation $⟦ F ⟧ r \rightrightarrows ⟦ F ⟧ s$ between interpretations. In the Fix case, care has to be taken to ensure that the mapping function f is applied to the parameters on the left, and map to the recursive positions on the right:

```
_⇉_ : ∀ {I} → (R S : Indexed I) → Set
r ⇉ s = (i : _) → r i → s i

_‖_ : ∀ {I J} {A C : Indexed I} {B D : Indexed J} → A ⇉ C → B ⇉ D → (A | B) ⇉ (C | D)
(f ‖ g) (inj₁ x) z = f x z
(f ‖ g) (inj₂ x) z = g x z

map : {I O : Set} {r s : Indexed I} (F : Code I O) → (r ⇉ s) → ⟦ F ⟧ r ⇉ ⟦ F ⟧ s
map U       f i _       = tt
map (I j)   f i x       = f j x
map (! j)   f i x       = x
map (F ⊕ G) f i (inj₁ x) = inj₁ (map F f i x)
map (F ⊕ G) f i (inj₂ x) = inj₂ (map G f i x)
map (F ⊗ G) f i (x , y) = map F f i x , map G f i y
map (F ◎ G) f i x       = map F (map G f) i x
map (Fix F) f i ⟨ x ⟩   = ⟨ map F (f ‖ map (Fix F) f) i x ⟩
```

For more information and examples of how to encode datatypes in `Indexed`, we refer the reader to the paper that introduced this approach [16].

2.5 Instant Generics

`InstantGenerics` [4] is another approach to generic programming in Haskell with type families. It distinguishes itself from all the other approaches we have discussed so far in that it does not represent recursion via a fixed-point combinator. Like `Regular`, `InstantGenerics` also supports a generic rewriting library [21]. To allow meta-variables to occur at any position (i.e. not only in recursive positions), type-safe runtime casts are performed to determine if the type of the meta-variable matches that of the expression. `InstantGenerics` is also rather similar to the generic programming support recently built into the Glasgow and Utrecht Haskell compilers [17].

In the original encoding of `InstantGenerics` in Haskell, recursive datatypes are handled through indirect recursion between the conversion functions (between a datatype and its generic representation) and the generic functions. We find that the most natural way to model this in Agda is to use coinduction [7]. This allows us to define infinite codes, and generic functions operating on these codes, while still passing the termination check. This encoding would also be appropriate for other Haskell approaches without a fixed-point operator, such as "Generics for the Masses" [9] and `LIGD` [6]. Although approaches without a fixed-point operator have trouble expressing recursive morphisms, they have been popular in Haskell because they easily allow encoding datatypes with irregular forms of recursion (such as mutually recursive or nested [3] datatypes).

Compared to the previous approaches, the novelties in the universe of `InstantGenerics` are a code K for arbitrary Sets, and a code R for tagging recursive codes. We give the interpretation as a datatype to ensure that it is inductive[1]. The judicious use of the coinduction primitive $\flat$ in the R case makes the Agda encoding pass the termination checker, as the definitions remain productive:

```
data Code : Set₁ where
  U   :                          Code
  K   : Set                 → Code
  R   : (C : ∞ Code)        → Code
  _⊕_ : (C D : Code)        → Code
  _⊗_ : (C D : Code)        → Code
```

```
data ⟦_⟧ : Code → Set₁ where
  tt   :                                          ⟦ U     ⟧
  k    : {A : Set}      → A                 → ⟦ K A   ⟧
  rec  : {C : ∞ Code} → ⟦ ♭ C ⟧           → ⟦ R C   ⟧
  inj₁ : {C D : Code} → ⟦ C ⟧             → ⟦ C ⊕ D ⟧
  inj₂ : {C D : Code} → ⟦ D ⟧             → ⟦ C ⊕ D ⟧
  _,_  : {C D : Code} → ⟦ C ⟧ → ⟦ D ⟧ → ⟦ C ⊗ D ⟧
```

Note the encoding of lists in `InstantGenerics`:

```
ListC : Set → Code
ListC A = U ⊕ (K A ⊗ R (♯ ListC A))
```

The definition for ListC is directly recursive; since it remains productive, it is accepted by the termination checker. Due to the lack of fixed points, we cannot write a map function. But we can easily write other generic functions, also recursive, such as a traversal that crushes a term into a result:

```
crush : {R : Set} (A : Code) → (R → R → R) → (R → R) → R → ⟦ A ⟧ → R
crush U       _⊞_ ⇑ 𝟙 _       = 𝟙
```

[1] Alternatively, we could use the experimental Agda flag `--guardedness-preserving-type-constructors` to treat type constructors as inductive constructors when checking productivity.

```
crush (K y)     _⊞_ ⇑ 𝟙 _          = 𝟙
crush (R C)     _⊞_ ⇑ 𝟙 (rec x)    = ⇑ (crush (♭ C) _⊞_ ⇑ 𝟙 x)
crush (C ⊕ D) _⊞_ ⇑ 𝟙 (inj₁ x)     = crush C _⊞_ ⇑ 𝟙 x
crush (C ⊕ D) _⊞_ ⇑ 𝟙 (inj₂ x)     = crush D _⊞_ ⇑ 𝟙 x
crush (C ⊗ D) _⊞_ ⇑ 𝟙 (x , y)     = (crush C _⊞_ ⇑ 𝟙 x) ⊞ (crush D _⊞_ ⇑ 𝟙 y)
```

Function crush is similar to map in the sense that it can be used to define many generic functions. It takes three arguments that specify how to combine the results of each constructor argument (_⊞_), how to adapt the result of a recursive call (⇑), and what to return for constants and constructors with no arguments (𝟙).[2] However, crush is unable to change the type of datatype parameters, since `InstantGenerics` has no knowledge of parameters.

We can compute the size of a structure as a crush, for instance:

```
size : (A : Code) → ⟦ A ⟧ → ℕ
size C = crush C _+_ suc 0
```

Here we combine multiple results by adding them, increment the total at every recursive call, and ignore constants and units for size purposes. We can test that this function behaves as expected on lists:

```
aList : ⟦ ListC ⊤ ⟧
aList = inj₂ (k tt , rec (inj₂ (k tt , rec (inj₁ tt))))
testSize : size _ aList ≡ 2
testSize = refl
```

While a map function cannot be defined like in the previous approaches, traversal and transformation functions can still be expressed in `InstantGenerics`. In particular, if one is willing to exchange static by dynamic type checking, type-safe runtime casts can be performed to compare the types of elements being mapped against the type expected by the mapping function, resulting in convenient to use generic functions [21]. However, runtime casting is known to result in poor runtime performance, as it prevents the compiler from performing type-directed optimisations [18].

2.6 Summary

We have shown an Agda encoding for the five libraries we compare. In order to simplify the proofs for the remainder of the paper, we have omitted a few details:

- In their original formulation, all libraries supported embedding Sets into the universe (like the K code in `InstantGenerics`). We omit these for simplicity, except in `InstantGenerics` where they are essential for embedding datatype parameters.
- We paid no attention to ease of use of the encodings. Adding isomorphisms to the universe [16], for instance, would make the encodings easier to use in practice. However, for our purposes of formal modelling, isomorphisms would serve only to enlarge the proofs, with no added benefit. Therefore we decided to omit them from the model.
- The variant of `Indexed` we consider here is significantly simpler than its original presentation [16]. In particular, we omit the sigma code, which is used for encoding indexed types. If we were to consider this code then `Indexed` would no longer embed fully into `InstantGenerics` (Section 3.3), since the latter does not support indexed types.

[2] It is worth noting that crush is itself a simplified catamorphism for the Code type.

3 Comparing the approaches

We now proceed to describe how the approaches relate to each other. We show which approaches can be *embedded* in other approaches; when we say that approach `A` embeds into approach `B`, we mean that the interpretation of any code defined in approach `A` has an equivalent interpretation in approach `B`. The starting point of an embedding is a code-conversion function that maps codes from approach `A` into approach `B`. Figure 1 presents a graphical view of the embedding relation between the five approaches; the arrows mean "embeds into". Note that the embedding relation is naturally transitive. As expected, `MultiRec` and `PolyP` both subsume `Regular`, but they don't subsume each other, since one supports families of recursive types and the other supports one parameter. They are however both subsumed by `Indexed`. Finally, the liberal encoding of `InstantGenerics` allows encoding at least all the types supported by the other approaches (even if it doesn't support the same operations on those types, such as catamorphisms).

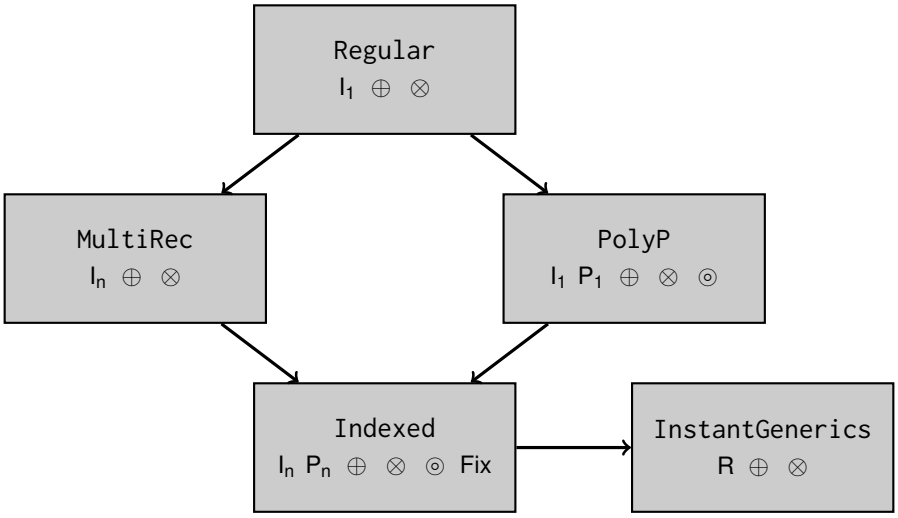

Figure 1: Embedding relation between the approaches

We will now focus on some of the conversions and their proofs, choosing those that are particularly interesting.

3.1 Regular to PolyP

We start with the simplest relation: embedding `Regular` into `PolyP`. The first step is to convert `Regular` codes into `PolyP` codes:

$$
\begin{aligned}
&{}_{\mathsf{r}}\Uparrow^{\mathsf{p}} : \mathsf{Code}_{\mathsf{r}} \to \mathsf{Code}_{\mathsf{p}} \\
&{}_{\mathsf{r}}\Uparrow^{\mathsf{p}}\ \mathsf{U}_{\mathsf{r}} &&= \mathsf{U}_{\mathsf{p}} \\
&{}_{\mathsf{r}}\Uparrow^{\mathsf{p}}\ \mathsf{I}_{\mathsf{r}} &&= \mathsf{I}_{\mathsf{p}} \\
&{}_{\mathsf{r}}\Uparrow^{\mathsf{p}}\ (\mathsf{F} \oplus_{\mathsf{r}} \mathsf{G}) &&= ({}_{\mathsf{r}}\Uparrow^{\mathsf{p}}\ \mathsf{F}) \oplus_{\mathsf{p}} ({}_{\mathsf{r}}\Uparrow^{\mathsf{p}}\ \mathsf{G}) \\
&{}_{\mathsf{r}}\Uparrow^{\mathsf{p}}\ (\mathsf{F} \otimes_{\mathsf{r}} \mathsf{G}) &&= ({}_{\mathsf{r}}\Uparrow^{\mathsf{p}}\ \mathsf{F}) \otimes_{\mathsf{p}} ({}_{\mathsf{r}}\Uparrow^{\mathsf{p}}\ \mathsf{G})
\end{aligned}
$$

Since all libraries share similar names, we use subscripts to denote to what library we are referring; r for `Regular` and p for `PolyP`. All `Regular` codes embed trivially into `PolyP`, so ${}_{\mathsf{r}}\Uparrow^{\mathsf{p}}$ is unsurprising. After

having defined code conversion, we can show that the interpretation of a code in `Regular` is equivalent to the converted code in `PolyP`. We do this by defining an isomorphism between the two interpretations (and their fixed points). We show one direction of the conversion, from `Regular` to `PolyP`:

```
from_r : {R : Set} (C : Code_r) → ⟦C⟧_r R → ⟦_r⇑^p C⟧_p ⊥ R
from_r U_r       tt       = tt
from_r I_r       x        = x
from_r (F ⊕_r G) (inj_1 x) = inj_1 (from_r F x)
from_r (F ⊕_r G) (inj_2 x) = inj_2 (from_r G x)
from_r (F ⊗_r G) (x , y)   = from_r F x , from_r G y
fromμ_r : (C : Code_r) → μ_r C → μ_p (_r⇑^p C) ⊥
fromμ_r C ⟨x⟩_r = ⟨from_r C (map_r C (fromμ_r C) x)⟩_p
```

Since `Regular` does not support parameters, we set the `PolyP` parameter to $\bot$ for `Regular` codes. Function from_r does the conversion of one layer, while $\mathsf{from}\mu_\mathsf{r}$ ties the recursive knot by expanding fixed points, converting one layer, and mapping itself recursively. Unfortunately $\mathsf{from}\mu_\mathsf{r}$ (and indeed all conversions in this paper involving fixed points) does not pass Agda's termination checker; we provide some insights on how to address this problem in Section 4.

The conversion in the other direction (to_r and $\mathsf{to}\mu_\mathsf{r}$) is entirely symmetrical.

Having defined the conversion functions, we now have to prove that they indeed form an isomorphism. We first consider the case without fixed points:

```
iso_1 : {R : Set} → (C : Code_r) → (x : ⟦C⟧_r R) → to_r C (from_r C x) ≡ x
iso_1 U_r       _        = refl
iso_1 I_r       _        = refl
iso_1 (F ⊕_r G) (inj_1 x) = cong inj_1 (iso_1 F x)
iso_1 (F ⊕_r G) (inj_2 x) = cong inj_2 (iso_1 G x)
iso_1 (F ⊗_r G) (x , y)   = cong_2 _,_ (iso_1 F x) (iso_1 G y)
```

This proof is trivial, and so is its counterpart for the other direction ($\mathsf{from}_\mathsf{r}\ \mathsf{C}\ (\mathsf{to}_\mathsf{r}\ \mathsf{C}\ \mathsf{x}) \equiv \mathsf{x}$). We use the Agda terms refl, sym, and cong for expressing reflexivity, symmetry, and congruence of equivalences, respectively. In particular cong is used very often as it allows us to set the focus of the proof deeper inside an expression.

When considering fixed points the proofs become more involved, since recursion has to be taken into account. However, using the equational reasoning module from the standard library (see the work of Shin-Cheng Mu et al. [20] for a detailed account on this style of proofs in Agda) we can keep the proofs readable:

```
open ≡-Reasoning
isoμ_1 : (C : Code_r) (x : μ_r C) → toμ_r C (fromμ_r C x) ≡ x
isoμ_1 C ⟨x⟩_r = cong ⟨_⟩_r $
  begin
    to_r C (map_p (_r⇑^p C) id (toμ_r C) (from_r C (map_r C (fromμ_r C) x)))
  ≡⟨ mapCommute^p_r C _ ⟩
    map_r C (toμ_r C) (to_r C (from_r C (map_r C (fromμ_r C) x)))
```

$$
\begin{aligned}
&\equiv\langle\ \mathsf{cong}\ (\mathsf{map_r}\ \mathsf{C}\ (\mathsf{to}\mu_\mathsf{r}\ \mathsf{C}))\ (\mathsf{iso}_1\ \mathsf{C}\ \_)\ \rangle \\
&\quad \underline{\mathsf{map_r}\ \mathsf{C}}\ (\mathsf{to}\mu_\mathsf{r}\ \mathsf{C})\ (\underline{\mathsf{map_r}\ \mathsf{C}}\ (\mathsf{from}\mu_\mathsf{r}\ \mathsf{C})\ \mathsf{x}) \\
&\equiv\langle\ \mathsf{map}^{\circ}_\mathsf{r}\ \mathsf{C}\ \rangle \\
&\quad \mathsf{map_r}\ \mathsf{C}\ \underline{(\mathsf{to}\mu_\mathsf{r}\ \mathsf{C} \circ \mathsf{from}\mu_\mathsf{r}\ \mathsf{C})}\ \mathsf{x} \\
&\equiv\langle\ \mathsf{map}^{\forall}_\mathsf{r}\ \mathsf{C}\ (\mathsf{iso}\mu_1\ \mathsf{C})\ \mathsf{x}\ \rangle \\
&\quad \underline{\mathsf{map_r}\ \mathsf{C}\ \mathsf{id}}\ \mathsf{x} \\
&\equiv\langle\ \mathsf{map}^{\mathsf{id}}_\mathsf{r}\ \mathsf{C}\ \rangle \\
&\quad \mathsf{x} \qquad \blacksquare
\end{aligned}
$$

To ease the reading of the equational style proofs, we highlight the term(s) that we are focusing on at each step. In this proof we start with an argument relating the maps of `Regular` and `PolyP`:

$$
\begin{aligned}
\mathsf{mapCommute}^{\mathsf{p}}_\mathsf{r}\ :\ & \{\mathsf{R}_1\ \mathsf{R}_2 : \mathsf{Set}\}\ \{\mathsf{f} : \mathsf{R}_1 \to \mathsf{R}_2\}\ (\mathsf{C} : \mathsf{Code_r})\ (\mathsf{x} : [\![\mathsf{r}\Uparrow^{\mathsf{p}}\ \mathsf{C}]\!]_\mathsf{p}\ \bot\ \mathsf{R}_1) \\
& \to \mathsf{to_r}\ \mathsf{C}\ (\mathsf{map_p}\ (\mathsf{r}\Uparrow^{\mathsf{p}}\ \mathsf{C})\ \mathsf{id}\ \mathsf{f}\ \mathsf{x}) \equiv \mathsf{map_r}\ \mathsf{C}\ \mathsf{f}\ (\mathsf{to_r}\ \mathsf{C}\ \mathsf{x})
\end{aligned}
$$

In words, this theorem states that the following two operations over a `Regular` term x that has been lifted into `PolyP` are equivalent:

- To map a function f over the recursive positions of x in `PolyP` with $\mathsf{map_p}$ and then convert to `Regular`;
- To first convert x back into `Regular`, and then map the function f with $\mathsf{map_r}$.

After this step, we either proceed by a recursive argument (referring to iso_1 or $\mathsf{iso}\mu_1$) or by a lemma. For conciseness, we show only the types of the lemmas:

$$
\begin{aligned}
&\mathsf{map}^{\circ}_\mathsf{r}\ :\ \{\mathsf{A}\ \mathsf{B}\ \mathsf{C} : \mathsf{Set}\}\ \{\mathsf{f} : \mathsf{B} \to \mathsf{C}\}\ \{\mathsf{g} : \mathsf{A} \to \mathsf{B}\}\ (\mathsf{D} : \mathsf{Code_r})\ \{\mathsf{x} : [\![\mathsf{D}]\!]_\mathsf{r}\ \mathsf{A}\} \\
&\quad \to \mathsf{map_r}\ \mathsf{D}\ \mathsf{f}\ (\mathsf{map_r}\ \mathsf{D}\ \mathsf{g}\ \mathsf{x}) \equiv \mathsf{map_r}\ \mathsf{D}\ (\mathsf{f} \circ \mathsf{g})\ \mathsf{x} \\
&\mathsf{map}^{\forall}_\mathsf{r}\ :\ (\mathsf{C} : \mathsf{Code_r})\ \{\mathsf{A}\ \mathsf{B} : \mathsf{Set}\}\ \{\mathsf{f}\ \mathsf{g} : \mathsf{A} \to \mathsf{B}\} \\
&\quad \to (\forall\ \mathsf{x} \to \mathsf{f}\ \mathsf{x} \equiv \mathsf{g}\ \mathsf{x}) \to (\forall\ \mathsf{x} \to \mathsf{map_r}\ \mathsf{C}\ \mathsf{f}\ \mathsf{x} \equiv \mathsf{map_r}\ \mathsf{C}\ \mathsf{g}\ \mathsf{x}) \\
&\mathsf{map}^{\mathsf{id}}_\mathsf{r}\ :\ \forall\ \{\mathsf{A}\}\ (\mathsf{C} : \mathsf{Code_r})\ \{\mathsf{x} : [\![\mathsf{C}]\!]_\mathsf{r}\ \mathsf{A}\} \to \mathsf{map_r}\ \mathsf{C}\ \mathsf{id}\ \mathsf{x} \equiv \mathsf{x}
\end{aligned}
$$

These lemmas are standard properties of $\mathsf{map_r}$, namely the functor laws and the fact that $\mathsf{map_r}$ preserves extensional equality (a form of congruence on $\mathsf{map_r}$). All of them are easily proved by induction on the codes.

Put together, $\mathsf{from}\mu_\mathsf{r}$, $\mathsf{to}\mu_\mathsf{r}$, $\mathsf{iso}\mu_1$, and $\mathsf{iso}\mu_2$ (the dual of $\mathsf{iso}\mu_1$) form an isomorphism that shows how to embed `Regular` codes into `PolyP` codes.

3.2 PolyP to Indexed

We proceed to the conversion between `PolyP` and `Indexed` codes. As we mentioned before, particular care has to be taken with composition; the remaining codes are trivially converted, so we only show composition:

$$
\begin{aligned}
&\mathsf{p}\Uparrow^{\mathsf{i}} : \mathsf{Code_p} \to \mathsf{Code_i}\ (\top \uplus \top)\ \top \\
&\mathsf{p}\Uparrow^{\mathsf{i}}\ (\mathsf{F} \circledcirc_\mathsf{p} \mathsf{G}) \ =\ (\mathsf{Fix_i}\ (\mathsf{p}\Uparrow^{\mathsf{i}}\ \mathsf{F})) \circledcirc_\mathsf{i} (\mathsf{p}\Uparrow^{\mathsf{i}}\ \mathsf{G})
\end{aligned}
$$

We cannot simply take the `Indexed` composition of the two converted codes because their types do not allow composition. A `PolyP` code results in an open `Indexed` code with one parameter and one recursive position, therefore of type $\mathsf{Code_i}\ (\top \uplus \top)\ \top$. Taking the fixed point of such a code gives a code of type $\mathsf{Code_i}\ \top\ \top$, so we can mimic `PolyP`'s interpretation of composition in our conversion, using the $\mathsf{Fix_i}$ of `Indexed`. In fact, converting `PolyP` to `Indexed` helps us understand the interpretation of composition in `PolyP`, because the types now show us that there's no way to define a composition other than by combining it with the fixed-point operator.

Converting composed values from `PolyP` is then a recursive task, due to the presence of fixed points. We first convert the outer functor F, and then map the conversion onto the arguments, recalling that on the left we have parameter codes G, while on the right we have recursive occurrences of the original composition:

$$\mathsf{from_p} : \{\mathsf{A\ R} : \mathsf{Set}\}\ (\mathsf{C} : \mathsf{Code_p}) \to [\![\mathsf{C}]\!]_\mathsf{p}\ \mathsf{A\ R} \to [\![{}_\mathsf{p}\Uparrow^\mathsf{i}\ \mathsf{C}]\!]_\mathsf{i}\ ((\mathsf{const\ A})\ |_\mathsf{i}\ (\mathsf{const\ R}))\ \mathsf{tt}$$
$$\mathsf{from_p}\ (\mathsf{F} \circledcirc_\mathsf{p} \mathsf{G})\ \langle \mathsf{x} \rangle_\mathsf{p} = \langle \mathsf{map_i}\ ({}_\mathsf{p}\Uparrow^\mathsf{i}\ \mathsf{F})\ ((\lambda\ \_ \to \mathsf{from_p}\ \mathsf{G})\ \|_\mathsf{i}\ (\lambda\ \_ \to \mathsf{from_p}\ (\mathsf{F} \circledcirc_\mathsf{p} \mathsf{G})))\ \mathsf{tt}\ (\mathsf{from_p\ F\ x}) \rangle_\mathsf{i}$$

We also show the conversion in the opposite direction, which is entirely symmetrical:

$$\mathsf{to_p} : \{\mathsf{A\ R} : \mathsf{Set}\}\ (\mathsf{C} : \mathsf{Code_p}) \to [\![{}_\mathsf{p}\Uparrow^\mathsf{i}\ \mathsf{C}]\!]_\mathsf{i}\ ((\mathsf{const\ A})\ |_\mathsf{i}\ (\mathsf{const\ R}))\ \mathsf{tt} \to [\![\mathsf{C}]\!]_\mathsf{p}\ \mathsf{A\ R}$$
$$\mathsf{to_p}\ (\mathsf{F} \circledcirc_\mathsf{p} \mathsf{G})\ \langle \mathsf{x} \rangle_\mathsf{i} = \langle \mathsf{to_p\ F}\ (\mathsf{map_i}\ ({}_\mathsf{p}\Uparrow^\mathsf{i}\ \mathsf{F})\ ((\lambda\ \_ \to \mathsf{to_p\ G})\ \|_\mathsf{i}\ (\lambda\ \_ \to \mathsf{to_p}\ (\mathsf{F} \circledcirc_\mathsf{p} \mathsf{G})))\ \mathsf{tt\ x}) \rangle_\mathsf{p}$$

The conversion of `PolyP` fixed points is very similar to the conversion of composition. The main difference lies in the functions that we map to the arguments and recursive positions:

$$\mathsf{from}\mu_\mathsf{p} : \{\mathsf{A} : \mathsf{Set}\}\ (\mathsf{C} : \mathsf{Code_p}) \to \mu_\mathsf{p}\ \mathsf{C\ A} \to [\![\mathsf{Fix_i}\ ({}_\mathsf{p}\Uparrow^\mathsf{i}\ \mathsf{C})]\!]_\mathsf{i}\ (\mathsf{const\ A})\ \mathsf{tt}$$
$$\mathsf{from}\mu_\mathsf{p}\ \mathsf{C}\ \langle \mathsf{x} \rangle_\mathsf{p} = \langle \mathsf{map_i}\ ({}_\mathsf{p}\Uparrow^\mathsf{i}\ \mathsf{C})\ ((\mathsf{const\ id})\ \|_\mathsf{i}\ (\lambda\ \_ \to \mathsf{from}\mu_\mathsf{p}\ \mathsf{C}))\ \mathsf{tt}\ (\mathsf{from_p\ C\ x}) \rangle_\mathsf{i}$$

We omit the conversion in the opposite direction for brevity, and also the isomorphism proof. Both the case for composition and for `PolyP` fixed points require lengthy proofs using properties of $\mathsf{map_i}$, which are presented in more detail in the following section.[3]

3.3 Indexed to InstantGenerics

As a final example we show how to convert from a fixed-point view to the coinductive representation of `InstantGenerics`. Since all fixed-point views embed into `Indexed`, we need to define only the embedding of `Indexed` into `InstantGenerics`. Since the two universes are less similar, the code transformation requires more care:

$${}_\mathsf{i}\Uparrow^\mathsf{ig} : \{\mathsf{I\ O} : \mathsf{Set}\} \to \mathsf{Code_i\ I\ O} \to (\mathsf{I} \to \mathsf{Set}) \to (\mathsf{O} \to \mathsf{Code_{ig}})$$
$${}_\mathsf{i}\Uparrow^\mathsf{ig}\ \mathsf{U_i}\qquad \mathsf{r\ o} = \mathsf{U_{ig}}$$
$${}_\mathsf{i}\Uparrow^\mathsf{ig}\ (\mathsf{I_i\ i})\qquad \mathsf{r\ o} = \mathsf{K_{ig}}\ (\mathsf{r\ i})$$
$${}_\mathsf{i}\Uparrow^\mathsf{ig}\ (!_\mathsf{i}\ \mathsf{i})\qquad \mathsf{r\ o} = \mathsf{K_{ig}}\ (\mathsf{o} \equiv \mathsf{i})$$
$${}_\mathsf{i}\Uparrow^\mathsf{ig}\ (\mathsf{F} \oplus_\mathsf{i} \mathsf{G})\ \mathsf{r\ o} = ({}_\mathsf{i}\Uparrow^\mathsf{ig}\ \mathsf{F\ r\ o}) \oplus_\mathsf{ig} ({}_\mathsf{i}\Uparrow^\mathsf{ig}\ \mathsf{G\ r\ o})$$
$${}_\mathsf{i}\Uparrow^\mathsf{ig}\ (\mathsf{F} \otimes_\mathsf{i} \mathsf{G})\ \mathsf{r\ o} = ({}_\mathsf{i}\Uparrow^\mathsf{ig}\ \mathsf{F\ r\ o}) \otimes_\mathsf{ig} ({}_\mathsf{i}\Uparrow^\mathsf{ig}\ \mathsf{G\ r\ o})$$
$${}_\mathsf{i}\Uparrow^\mathsf{ig}\ (\mathsf{F} \circledcirc_\mathsf{i} \mathsf{G})\ \mathsf{r\ o} = \mathsf{R_{ig}}\ (\sharp\ {}_\mathsf{i}\Uparrow^\mathsf{ig}\ \mathsf{F}\ (\lambda\ \mathsf{i} \to [\![{}_\mathsf{i}\Uparrow^\mathsf{ig}\ \mathsf{G\ r\ i}]\!]_\mathsf{ig})\ \mathsf{o})$$

[3] This proof and other omitted details can be found in the code available at the first author's webpage (`http://dreixel.net`).

$$_{i}\Uparrow^{ig}\ (\mathsf{Fix}_i\ F)\ \ r\ o\ =\ \mathsf{R}_{ig}\ (\sharp\ _{i}\Uparrow^{ig}\ F\ (r\,|_i\,(\lambda\ i \to [\![_{i}\Uparrow^{ig}\ (\mathsf{Fix}_i\ F)\ r\ i]\!]_{ig}))\ o)$$

Unit, sum, and product exist in both universes, so their conversion is trivial. Recursive invocations with I_i are replaced by simple constants; we lose the ability to abstract over recursive positions, which is in line with the behavior of `InstantGenerics`. Tagging is also converted to a constant, trivially inhabited if we are in the expected output index o, and empty otherwise. Note that since an `Indexed` code can define multiple types, but an `InstantGenerics` code can only represent one type, $_{i}\Uparrow^{ig}$ effectively produces multiple `InstantGenerics` codes, one for each output index of the original `Indexed` family.

A composition $F \circledcirc_i G$ is encoded through recursion; the resulting code is the conversion of F, whose parameters are `Indexed` G functors. We convert these functors to sets using $_{i}\Uparrow^{ig}$ to get `InstantGenerics` codes, which we then interpret with $[\![\_]\!]_{ig}$.

A fixed-point Fix_i F is naturally encoded through recursion, in a similar way to composition. The recursive positions of the fixed point are either: parameters on the left, converted with r as before; or recursive occurrences on the right, handled by recursively converting the codes with $_{i}\Uparrow^{ig}$ and interpreting.

Note that both for composition and fixed points we instantiate the function which interprets indices (the r argument) with an `InstantGenerics` interpretation. However, r has type $I \to \mathsf{Set}$, whereas $[\![\_]\!]_{ig}$ has return type Set_1. If we were to raise `Indexed` to Set_1, the interpretation function would then have type $I \to \mathsf{Set}_1$, but then we could no longer use it in the I_i case. For now we rely on the Agda flag `--type-in-type`, and leave a formally correct solution for future work (see Section 4).

Having the code conversion in place, we can proceed to convert values:

$$\begin{aligned}
&\mathsf{from} : \{I\ O : \mathsf{Set}\}\ \{r : I \to \mathsf{Set}\}\ (C : \mathsf{Code}_i\ I\ O)\ (o : O) \to [\![C]\!]_i\ r\ o \to [\![_{i}\Uparrow^{ig}\ C\ r\ o]\!]_{ig}\\
&\mathsf{from}\ \mathsf{U}_i\ \ \ \ \ o\ \mathsf{tt}\ \ \ \ \ \ = \mathsf{tt}_{ig}\\
&\mathsf{from}\ (\mathsf{I}_i\ i)\ \ \ \ o\ x\ \ \ \ \ \ = \mathsf{k}_{ig}\ x\\
&\mathsf{from}\ (!_i\ i)\ \ \ \ o\ x\ \ \ \ \ \ = \mathsf{k}_{ig}\ x\\
&\mathsf{from}\ (F \oplus_i G)\ o\ (\mathsf{inj}_1\ x) = \mathsf{inj}_{1\,ig}\ (\mathsf{from}\ F\ o\ x)\\
&\mathsf{from}\ (F \oplus_i G)\ o\ (\mathsf{inj}_2\ x) = \mathsf{inj}_{2\,ig}\ (\mathsf{from}\ G\ o\ x)\\
&\mathsf{from}\ (F \otimes_i G)\ o\ (x\ ,\ y) = (\mathsf{from}\ F\ o\ x),_{ig}\ (\mathsf{from}\ G\ o\ y)\\
&\mathsf{from}\ (F \circledcirc_i G)\ o\ x\ \ \ \ \ = \mathsf{rec}_{ig}\ (\mathsf{from}\ F\ o\ (\mathsf{map}_i\ F\ (\mathsf{from}\ G)\ o\ x))\\
&\mathsf{from}\ (\mathsf{Fix}_i\ F)\ \ o\ \langle x\rangle_i\ \ \ \ = \mathsf{rec}_{ig}\ (\mathsf{from}\ F\ o\ (\mathsf{map}_i\ F\ ((\lambda\ _ \to \mathsf{id})\ \|_i\ (\mathsf{from}\ (\mathsf{Fix}_i\ F)))\ o\ x))
\end{aligned}$$

The cases for composition and fixed point are more challenging because we have to map the conversion function inside the argument positions; we do this using the map_i function. As usual, the inverse function to is entirely symmetrical, so we omit it.

It remains to show that the conversion functions form an isomorphism. We show the only two interesting cases: composition and fixed points. Following previous work [16], we lift composition, equality, and identity to natural transformations in `Indexed` (respectively $\_ \circ_{\rightrightarrows_i} \_$, $\_ \cong_i \_$, and $\mathsf{id}_{\rightrightarrows_i}$). We use equational reasoning for the proofs, and highlight the part of the term that we focus on in each line of the proof:

$$\begin{aligned}
&\mathsf{iso}_1 : \{I\ O : \mathsf{Set}\}\ (C : \mathsf{Code}_i\ I\ O)\ (r : I \to \mathsf{Set}) \to (\mathsf{to}\ \{r = r\}\ C \circ_{\rightrightarrows_i} \mathsf{from}\ C) \cong_i \mathsf{id}_{\rightrightarrows_i}\\
&\mathsf{iso}_1\ (F \circledcirc_i G)\ r\ o\ x\ =\\
&\quad \mathsf{begin}\\
&\qquad \mathsf{map}_i\ F\ (\mathsf{to}\ G)\ o\ (\underline{\mathsf{to}\ F\ o\ (\mathsf{from}\ F}\ o\ (\mathsf{map}_i\ F\ (\mathsf{from}\ G)\ o\ x)))\\
&\quad \equiv\langle\ \mathsf{cong}\ (\mathsf{map}_i\ F\ (\mathsf{to}\ G)\ o)\ (\mathsf{iso}_1\ F\ _\ o\ _)\ \rangle
\end{aligned}$$

```
    map_i F (to G) o (map_i F (from G) o x)
≡⟨ sym (map°_i F (to G) (from G) o x) ⟩
    map_i F (to G ∘⇉_i from G) o x
≡⟨ map^∀_i F (iso₁ G r) o x ⟩
    map_i F id⇉_i o x
≡⟨ map^id_i F o x ⟩
    x                                                                    ∎
```

The proof for composition is relatively simple, relying on applying the proof recursively, fusing the two maps, reasoning by recursion on the resulting map, which results in an identity map. The proof for fixed points is slightly more involved:

```
iso₁ (Fix_i F) r o ⟨x⟩_i = cong ⟨_⟩_i $
  begin
    map_i F (id⇉_i ‖_i (to (Fix_i F))) o (to F o (from F o (map_i F (id⇉_i ‖_i (from (Fix_i F))) o x)))
  ≡⟨ cong (map_i F (id⇉_i ‖_i (to (Fix_i F))) o) (iso₁ F _ o _) ⟩
    map_i F (id⇉_i ‖_i (to (Fix_i F))) o (map_i F (id⇉_i ‖_i (from (Fix_i F))) o x)
  ≡⟨ sym (map°_i F (id⇉_i ‖_i (to (Fix_i F))) (id⇉_i ‖_i (from (Fix_i F))) o x) ⟩
    map_i F (id⇉_i ‖_i (to (Fix_i F))) ∘⇉_i (id⇉_i ‖_i (from (Fix_i F))) o x
  ≡⟨ sym (map^∀_i F ‖∘_i o x) ⟩
    map_i F ((id⇉_i ∘⇉_i id⇉_i) ‖_i (to (Fix_i F) ∘⇉_i from (Fix_i F))) o x
  ≡⟨ map^∀_i F (‖_cong_i (λ _ _ → refl) (iso₁ (Fix_i F) r)) o x ⟩
    map_i F (id⇉_i ‖_i id⇉_i) o x
  ≡⟨ map^∀_i F (‖_id_i (λ _ _ → refl) (λ _ _ → refl)) o x ⟩
    map_i F id⇉_i o x
  ≡⟨ map^id_i F o x ⟩
    x                                                                    ∎
```

We start in the same way as with composition, but once we have fused the maps we have to deal with the fact that we are mapping distinct functions to the left (arguments) and right (recursive positions). We proceed with a lemma on disjunctive maps that states that a composition of disjunctions is the disjunction of the compositions ($\|\circ_i$). Then we are left with a composition of identities on the left, which we solve with reflexivity, and a composition of to and from on the right, which we solve by induction. Finally, we show that a disjunction of identities is the identity (with the $\|_{id_i}$ lemma), and that the identity map is the identity. The lemmas regarding map_i that we require are the following:

```
map°_i : {I O : Set} {r s t : Indexed I}
         (C : Code_i I O) (f : s ⇉_i t) (g : r ⇉_i s) → map_i C (f ∘⇉_i g) ≅_i (map_i C f ∘⇉_i map_i C g)
map^∀_i : {I O : Set} {r s : Indexed I} {f g : r ⇉_i s}
          (C : Code_i I O) → f ≅_i g → map_i C f ≅_i map_i C g
map^id_i : {I O : Set} {r : Indexed I} (C : Code_i I O) → map_i {r = r} C id⇉_i ≅_i id⇉_i
```

Like the lemmas for map_r of Section 3.1, these are trivially proven by induction on the structure of `Indexed` codes.

4 Discussion and future work

We have compared different generic programming universes by showing the inclusion relation between them. This is useful to determine that one approach can encode at least as many datatypes as another approach, and also allows for lifting representations between compatible approaches. This also means that generic functions from approach `B` are all applicable in approach `A`, if `A` embeds into `B`, because we can bring generic values from approach `A` into `B` and apply the function there. However, we cannot make statements about the variety of generic functions that can be encoded in each approach. The generic map function, for instance, cannot be defined in `InstantGenerics`, while it is standard in `Indexed`. One possible direction for future research is to devise a formal framework for evaluating what generic functions are possible in each universe, adding another dimension to the comparison of the approaches.

Notably absent from our comparison are libraries with a generic view not based on a sum of products. In particular, the spine view [12] is radically different from the approaches we model; yet, it is the basis for a number of popular generic programming libraries. It would be interesting to see how these approaches relate to those we have seen, but, at the moment, converting between entirely different universes remains a challenge.

An issue that remains with our modelling is to properly address termination. While our conversion functions can be used operationally to enable portability across different approaches, to serve as a formal proof they have to be terminating. Since Agda's algorithm for checking termination is highly syntax-driven, attempts to convince Agda of termination are likely to clutter the model, making it less easy to understand. We have thus decided to postpone such efforts for future work, perhaps relying on sized types for guaranteeing termination of our proofs [1].

A related issue that remains to be addressed is our use of `--type-in-type` in Section 3.3, for the code conversion function $_{\mathsf{i}}\Uparrow^{\mathsf{ig}}$. It was not immediately clear to us how to solve this issue even with the recently added support for universe polymorphism in Agda.

Nonetheless, we believe that our work is an important first step towards a formal categorisation of generic programming libraries. Future approaches can now rely on our formalisation to describe precisely the new aspects they introduce, and how the new approach relates to existing ones. In this way we can hope for a future of *modular generic programming*, where a specific library might be constructed using components from different approaches, tailored to a particular need while still reusing code from existing libraries.

References

[1] Andreas Abel (2010): *MiniAgda: Integrating Sized and Dependent Types*. In Ana Bove, Ekaterina Komendantskaya & Milad Niqui, editors: *Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers*, EPTCS 43, doi:10.4204/EPTCS.43.2.

[2] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride & Peter Morris: *Indexed Containers*. Unpublished manuscript, available at `http://strictlypositive.org/indexed-containers.pdf`.

[3] Richard Bird & Lambert Meertens (1998): *Nested datatypes*. In Johan Jeuring, editor: *Mathematics of Program Construction*, *LNCS* 1422, Springer, pp. 52–67, doi:10.1007/BFb0054285.

[4] Manuel M. T. Chakravarty, Gabriel C. Ditu & Roman Leshchinskiy (2009): *Instant Generics: Fast and Easy*. Available at `http://www.cse.unsw.edu.au/~chak/papers/CDL09.html`.

[5] James Chapman, Pierre-Évariste Dagand, Conor McBride & Peter Morris (2010): *The gentle art of levitation*. In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, ACM, New York, NY, USA, pp. 3–14, doi:10.1145/1863543.1863547.

[6] James Cheney & Ralf Hinze (2002): *A lightweight implementation of generics and dynamics*. In: *Proceedings of the ACM SIGPLAN Workshop on Haskell*, Haskell '02, ACM, New York, NY, USA, pp. 90–104, doi:10.1145/581690.581698.

[7] Nils Anders Danielsson & Thorsten Altenkirch (2010): *Subtyping, Declaratively*. In Claude Bolduc, Jules Desharnais & Béchir Ktari, editors: *Mathematics of Program Construction*, *LNCS* 6120, Springer Berlin / Heidelberg, pp. 100–118, doi:10.1007/978-3-642-13321-3_8.

[8] Jeremy Gibbons (2007): *Datatype-Generic Programming*. In: *Spring School on Datatype-Generic Programming*, *LNCS* 4719, Springer, pp. 1–71, doi:10.1007/978-3-540-76786-2_1.

[9] Ralf Hinze (2006): *Generics for the Masses*. *Journal of Functional Programming* 16(4-5), pp. 451–483, doi:10.1017/S0956796806006022.

[10] Ralf Hinze, Johan Jeuring & Andres Löh (2007): *Comparing Approches to Generic Programming in Haskell*. In: *Datatype-Generic Programming*, *LNCS* 4719, Springer, pp. 72–149, doi:10.1007/978-3-540-76786-2_2.

[11] Ralf Hinze & Andres Löh (2009): *Generic programming in 3D*. *Science of Computer Programming* 74, pp. 590–628, doi:10.1016/j.scico.2007.10.006.

[12] Ralf Hinze, Andres Löh & Bruno C. d. S. Oliveira (2006): *"Scrap Your Boilerplate" Reloaded*. In: *Functional and Logic Programming*, *LNCS* 3945, Springer, pp. 13–29, doi:10.1007/11737414_3.

[13] Stefan Holdermans, Johan Jeuring, Andres Löh & Alexey Rodriguez Yakushev (2006): *Generic Views on Data Types*. In: *Mathematics of Program Construction*, *LNCS* 4014, pp. 209–234, doi:10.1007/11783596_14.

[14] Patrik Jansson & Johan Jeuring (1997): *PolyP—a polytypic programming language extension*. In: *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, ACM, pp. 470–482, doi:10.1145/263699.263763.

[15] Andres Löh (2004): *Exploring Generic Haskell*. Ph.D. thesis, Utrecht University.

[16] Andres Löh & José Pedro Magalhães (2011): *Generic programming with indexed functors*. In: *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, WGP '11, ACM, New York, NY, USA, pp. 1–12, doi:10.1145/2036918.2036920.

[17] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring & Andres Löh (2010): *A generic deriving mechanism for Haskell*. In: *Proceedings of the ACM Symposium on Haskell*, Haskell '10, ACM, New York, NY, USA, pp. 37–48, doi:10.1145/1863523.1863529.

[18] José Pedro Magalhães, Stefan Holdermans, Johan Jeuring & Andres Löh (2010): *Optimizing Generics Is Easy!* In: *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM'10, ACM, New York, NY, USA, pp. 33–42, doi:10.1145/1706356.1706366.

[19] Erik Meijer, Maarten Fokkinga & Ross Paterson (1991): *Functional programming with bananas, lenses, envelopes and barbed wire*. In J. Hughes, editor: *Functional Programming Languages and Computer Architecture*, *LNCS* 523, Springer-Verlag, pp. 124–144, doi:10.1007/3540543961_7.

[20] Shin-Cheng Mu, Hsiang-Shang Ko & Patrik Jansson (2009): *Algebra of programming in Agda: Dependent types for relational program derivation*. *Journal of Functional Programming* 19, pp. 545–579, doi:10.1017/S0956796809007345.

[21] Thomas van Noort, Alexey Rodriguez Yakushev, Stefan Holdermans, Johan Jeuring, Bastiaan Heeren & José Pedro Magalhães (2010): *A lightweight approach to datatype-generic rewriting*. *Journal of Functional Programming* 20(Special Issue 3-4), pp. 375–413, doi:10.1017/S0956796810000183.

[22] Thomas van Noort, Alexey Rodriguez Yakushev, Stefan Holdermans, Johan Jeuring & Bastiaan Heeren (2008): *A lightweight approach to datatype-generic rewriting*. In: *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, WGP '08, ACM, pp. 13–24, doi:10.1145/1411318.1411321.

[23] Ulf Norell (2009): *Dependently typed programming in Agda*. In Pieter Koopman, Rinus Plasmeijer & Doaitse Swierstra, editors: *Advanced Functional Programming, 6th International School, AFP 2008, Revised Lectures*, *LNCS* 5832, Springer, pp. 230–266, doi:10.1007/978-3-642-04652-0_5.

[24] Simon Peyton Jones, editor (2003): *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press, doi:10.1017/S0956796803000315. Journal of Functional Programming Special Issue 13(1).

[25] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh & Johan Jeuring (2009): *Generic programming with fixed points for mutually recursive datatypes*. In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, ACM, pp. 233–244, doi:10.1145/1596550.1596585.

[26] Alexey Rodriguez Yakushev, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov & Bruno C.d.S. Oliveira (2008): *Comparing libraries for generic programming in Haskell*. In: *Proceedings of the ACM SIGPLAN Symposium on Haskell*, Haskell'08, ACM, pp. 111–122, doi:10.1145/1411286.1411301.

[27] Wendy Verbruggen, Edsko De Vries & Arthur Hughes (2010): *Formal polytypic programs and proofs*. *Journal of Functional Programming* 20(Special Issue 3-4), pp. 213–270, doi:10.1017/S0956796810000158.

[28] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers & Martin Sulzmann (2011): *OutsideIn(X)—Modular type inference with local assumptions*. *Journal of Functional Programming* 21(Special Issue 4-5), pp. 333–412, doi:10.1017/S0956796811000098.

[29] Stephanie Weirich & Chris Casinghino (2010): *Arity-generic datatype-generic programming*. In: *Proceedings of the ACM SIGPLAN Workshop on Programming Languages meets Program Verification*, PLPV '10, ACM, New York, NY, USA, pp. 15–26, doi:10.1145/1707790.1707799.

[30] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis & José Pedro Magalhães (2012): *Giving Haskell a Promotion*. In: *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, ACM, New York, NY, USA, pp. 53–66, doi:10.1145/2103786.2103795.