

Arity-generic datatype-generic programming

Stephanie Weirich Chris Casinghino

University of Pennsylvania
{sweirich,ccasin}@cis.upenn.edu

Abstract

Some programs are doubly-generic. For example, `map` is datatype-generic in that many different data structures support a mapping operation. A generic programming language like Generic Haskell can use a single definition to generate `map` for each type. However, `map` is also arity-generic because it belongs to a family of related operations that differ in the number of arguments. For lists, this family includes `repeat`, `map`, `zipWith`, `zipWith3`, `zipWith4`, etc. With dependent types or clever programming, one can unify all of these functions together in a single definition.

However, no one has explored the combination of these two forms of genericity. These two axes are not orthogonal because the idea of arity appears in Generic Haskell: datatype-generic versions of `repeat`, `map` and `zipWith` have different arities of kind-indexed type. In this paper, we define arity-generic datatype-generic programs by building a framework for Generic Haskell-style generic programming in the dependently-typed programming language Agda 2.

1. Introduction

This is a story about doubly-generic programming. *Datatype-generic* programming defines operations that may be instantiated at many different types, so these operations need not be redefined for each one. For example, Generic Haskell [Hinze 2002, Clarke et al. 2001] includes a generic `map` operation that has instances for types such as lists, optional values, and products (even though these types have different kinds).

```
gmap ⟨ [] ⟩      :: (a → b) → ([a] → [b])
gmap ⟨ Maybe ⟩ :: (a → b) → (Maybe a → Maybe b)
gmap ⟨ (,) ⟩    :: (a1 → b1) → (a2 → b2)
                → (a1, a2) → (b1, b2)
```

Because all the instances of `gmap` are generated from the same definition, reasoning about that generic function tells us about `map` at each type.

However, there is another way to generalize `map`. Consider the following sequence of functions from the Haskell Prelude [Peyton Jones et al. 2003], all of which operate on lists.

```
repeat :: a → [a]
map    :: (a → b) → [a] → [b]
```

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLPV 2010 Madrid, Spain

Copyright © 2010 ACM [to be supplied]...\$5.00

```
zipWith  :: (a → b → c) → [a] → [b] → [c]
zipWith3 :: (a → b → c → d) → [a] → [b] → [c] → [d]
```

The `repeat` function creates an infinite list from its argument. The `zipWith` function is a generalization of `zip`—it combines the two lists together with its argument instead of with the tupling function. Likewise, `zipWith3` combines three lists.

As Fridlender and Indrika [2000] have pointed out, all of these functions are instances of the same generic operation, they just have different *arities*. They demonstrate how to encode the arity as a Church numeral in Haskell and uniformly produce all of these list operations from the same definition.

More, arity-genericity is not unique to the list instance of `map`. It is not difficult to imagine arity-generic versions of the maps for `Maybes` and for products. Using ideas from applicative functors, Fridlender and Indrika’s technique can be generalized to types of kind $\star \Rightarrow \star$, such as `Maybe`, but it does not apply to types of higher-order kinds.

Other datatype-generic functions are arity-generic as well. In this paper, we examine arity-generic `map`, `split`, and `equality`. Other examples include some fold-like operations, an equality function which returns a proof of its result, and a monadic `map` which uses a `Maybe` to indicate when its arguments have different shapes.

Arity-genericity is not merely an intellectual curiosity, as the notion of arity is an important part of mechanism of Generic Haskell. Each datatype-generic function defined in Generic Haskell must be defined at a particular arity, and that arity corresponds to the arities in `map` above. For example, the Generic Haskell version of `repeat` has arity one, the version of `map` requires arity two, and the version of `zipWith` has arity three.

Generic Haskell does not permit generalizing over arities, so a single definition cannot produce `repeat`, `map` and `zipWith`. Likewise, Generic-Haskell-style libraries encoded in Haskell, such as `RepLib` [Weirich 2006b] or `EMGM` (Extensible and Modular Generics for the Masses) [d. S. Oliveira et al. 2007] specialize their infrastructure to specific arities, so they too cannot write arity-generic code.

However, Altenkirch and McBride [2003] and Verbruggen et al. [2008] have shown how to encode Generic-Haskell-style generic programming in dependently typed programming languages, and their encodings are more general than the original. Although they do not consider arity-genericity in their work, their methodology is the building block for doubly-generic programming.

In this paper, we develop a similar generic programming framework in the dependently-typed language Agda 2 [Norell 2007]. As a running example, we show how such an encoding may be used to define the arity-generic datatype-generic operation `ngmap`, which generalizes all of the definitions shown above. We choose Agda because it is particularly tailored to dependently-typed programming, but we could have also used a number of different languages, such as `Coq` [The Coq Development Team 2006], `Epigram` [McBride

and McKinna 2004], Ω mega [Sheard 2005], or Haskell with recent extensions [Peyton Jones et al. 2006, Chakravarty et al. 2005].

Our contributions are as follows:

1. We develop an arity-generic version of map that reveals commonality between `gmap`, `gzipWith` and `gzipWith3`. This correspondence has not previously been expressed, but we find that it leads to insight into the nature of these operations. Since the definitions are all instances of the same dependently-typed function, we have shown formally that they are related.
2. This example is developed on top of a reusable framework for generic programming in Agda. Although our framework has the same structure as previous work, our treatment of datatype isomorphisms is novel and requires less boilerplate.
3. We use our framework to develop other doubly-generic operations, such as equality and `unzipWith`. All of these examples shed light on arity support in a generic programming framework. In particular, there are not many operations that require arity of two or more: this work suggests what such operations must look like.
4. Finally, because we develop a reusable framework, this work demonstrates how a tool like Generic Haskell could be extended to arity-genericity.

We explain doubly-generic map and our generic programming infrastructure in stages. In Section 2 we give a (brief) introduction to Agda, describe how to use applicative functors to define an arity-generic version of map, and show why this methodology does not extend to datatype-genericity. Next, in Section 3, we describe a general framework for generic programming that works for all types (of any kind) formed from unit, pairs, sums and natural numbers. We use this framework to define doubly-generic map. In Section 4 we show how datatype isomorphisms may be incorporated, so that we can specialize doubly-generic operations to inductive datatypes, such as options and vectors. We discuss other doubly-generic examples of in Section 5. Finally, Sections 6 and 7 discuss related work and conclude.

All code described in this paper is available from <http://www.cis.upenn.edu/~ccasin/papers/aritygen.tar.gz>.

2. Arity-generic map

We begin this section by introducing Agda and using it to define applicative functors. We show (in Agda) how to use applicative functors to define arity-generic map for vectors, following Fridlender and Indrika. Finally, we demonstrate why this approach does not scale to implementing datatype-generic arity-generic map.

2.1 Programming with Dependent Types in Agda

Agda is a dependently typed programming language where terms may appear in types. For example, the Agda standard library defines a type of polymorphic length-indexed vectors:

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

This datatype `Vec` is parameterized by an argument `A` of type `Set`, the analogue of Haskell's kind `*`, and indexed by an argument of type `ℕ`¹, the type of natural numbers. The parameter `A` specifies the type of the objects stored in the vector and the index specifies its length. For example, the type `Vec Bool 2` is a list of boolean values of length two. Note that indices can vary in the types of the constructors: for example, empty vectors `[]` use index 0.

¹Note that Unicode symbols are valid in Agda identifiers.

The underscores in `_::__` create an infix operator. Arguments to Agda functions may be made implicit by placing them in curly braces, so Agda will attempt to infer the length index by unification when applying `_::__`. For example, Agda can automatically determine that the term `true :: false :: []` has type `Vec Bool 2`.

Vectors are applicative functors, familiar to Haskell programmers from the `Applicative` type class [McBride and Paterson 2008]. Applicative functors have two operations. The first is `repeat` (called `pure` in Haskell). Given an initial value, it constructs a vector with `n` copies of that value.

```
repeat : {n : ℕ} → {A : Set} → A → Vec A n
repeat {zero} x = []
repeat {suc n} x = x :: repeat {n} x
```

Observe that, using curly braces, implicit arguments can be explicitly provided in a function call or matched against in a definition.

The second, `_@*_`, is an infix zipping application, pronounced "zap" and defined by:

```
_@*_ : {A B : Set} {n : ℕ}
      → Vec (A → B) n → Vec A n → Vec B n
[] @* [] = []
(a :: As) @* (b :: Bs) = (a b :: As @* Bs)
```

In this definition, we do not need to consider the cases where one vector is empty while the other is not because the type specifies that both arguments have the same length.

These two operations are the key to arity-generic map. The following sequence shows that the types and definitions of the different arities of map follow a specific pattern.

```
map0      : {m : ℕ} {A : Set} → A → Vec A m
map0      = repeat

map1      : {m : ℕ} {A B : Set}
           → (A → B) → Vec A m → Vec B m
map1 f x  = repeat f @* x

map2      : {m : ℕ} {A B C : Set}
           → (A → B → C)
           → Vec A m → Vec B m → Vec C m
map2 f x1 x2 = repeat f @* x1 @* x2
```

Indeed, all of these maps are defined by a simple application of `repeat` and `n` copies of `_@*_`. Agda can express the arity-generic operation that unifies all of these maps via dependent types, as we present in the next subsection.

2.2 Arity-generic vector map

The difficulty in the definition of arity-generic map is that all of the instances have different types. Given some arity `n`, we must generate the corresponding type in this sequence. Fridlender and Indrika, not working in a dependently typed language, do so by encoding `n` as a Church numeral that generates the appropriate type for map.

We prefer to be more orthogonal and use natural numbers to express the arity of the mapping operation. Our approach is then to program with Agda types. For example, we can construct a vector of Agda types, `Bool :: ℕ :: []`, which has type `Vec Set 2`², and use the `repeat` and `_@*_` operations with such vectors.

The first step is to define `arrTy`, which folds the arrow type constructor `→` over a non-empty vector of types. Given such a

²This type requires `Set` to have type `Set`, enabled by Agda's `--type-in-type` flag. The standard type system of Agda has an infinite hierarchy of `Sets`, and users must resolve their code to be at the appropriate level. Although we have done so, in the interest of clarity we have hidden this hierarchy and its associated complexities. We discuss this choice further in Section 7.

vector, this operation constructs the type of the function that will be mapped over the n data structures.

```
arrTy : {n : ℕ} → Vec Set (suc n) → Set
arrTy {0} (A :: []) = A
arrTy {suc n} (A :: As) = A → arrTy As
```

The function `arrTyVec` constructs the result type of arity-generic map for vectors. We define this operation with the help of `arrTy`, `repeat` and `_⊗_` by mapping the `Vec` constructor onto the vector of types, then placing arrows between them. Notice that there are two integer indices here - n determines the number of types we are dealing with (the arity), while m is the length of the vectors we map over.

```
arrTyVec : {m n : ℕ} → Vec Set (suc n) → Set
arrTyVec {m} As =
  arrTy (repeat (λ A → Vec A m) ⊗ As)
```

For example, we can define the three types above using these functions applied to lists of type variables.

```
map0 : {m : ℕ} {A : Set}
      → arrTy (A :: [])
      → arrTyVec m (A :: [])
map1 : {m : ℕ} {A B : Set}
      → arrTy (A :: B :: [])
      → arrTyVec m (A :: B :: [])
map2 : {m : ℕ} {A B C : Set}
      → arrTy (A :: B :: C :: [])
      → arrTyVec m (A :: B :: C :: [])
```

Now, to define arity-generic map, we start by defining a function `nvec-map`. The type of this function mirrors the examples above, except that it takes in the type arguments (A , B , etc) as a vector (As). After we define `nvec-map` we will curry it to get the desired operation.

```
nvec-map : {m : ℕ} → (n : ℕ)
          → (As : Vec Set (suc n))
          → arrTy As → arrTyVec {m} As
```

Intuitively, the definition of `nvec-map` is a simple application of `repeat` and n copies of `_⊗_`:

```
nvec-map As f v1 v2 ... vn =
  repeat f ⊗ v1 ⊗ v2 ⊗ ... ⊗ vn
```

We define this function by recursion on n , in accumulator style. After duplicating f we have a vector of functions to `zap`, so we define a helper function, g , for that more general case.

```
nvec-map n As f = g {n} As (repeat f) where
g : {n m : ℕ}
  → (As : Vec Set (suc n))
  → Vec (arrTy As) m → arrTyVec {m} As
g {0} (A :: []) a = a
g {suc n} (A1 :: As) f =
  (λ a → g As (f ⊗ a))
```

Finally, we define two operations for currying. The first, $\forall\Rightarrow$, creates a curried version of a type which depends on a vector. The second, $\lambda\Rightarrow$, curries a corresponding function term.

```
 $\forall\Rightarrow$  : {n : ℕ} {A : Set} → (Vec A n → Set) → Set
 $\forall\Rightarrow$  {zero} B = B []
 $\forall\Rightarrow$  {suc n} {A} B =
  {a : A} →  $\forall\Rightarrow$  {n} (λ as → B (a :: as))
 $\lambda\Rightarrow$  : {n : ℕ} {A : Set} {B : Vec A n → Set}
      → ((X : Vec A n) → B X) → ( $\forall\Rightarrow$  B)
```

```
 $\lambda\Rightarrow$  {zero} f = f []
 $\lambda\Rightarrow$  {suc n} {A} f =
  λ {a : A} →  $\lambda\Rightarrow$  {n} (λ as → f (a :: as))
```

With these operations, we can finish the definition of arity-generic map. Note that in the definition below, the implicit argument m is the length of the vector, and the explicit argument n is the specific arity of map that is desired.

```
nmap : (n : ℕ) → {m : ℕ}
      →  $\forall\Rightarrow$  (λ (As : Vec Set (suc n))
            → arrTy As → arrTyVec {m} As)
nmap n {m} =  $\lambda\Rightarrow$  (nvec-map {m} n)
```

We can use this arity-generic map just by providing the arity as an additional argument. For example, the term `nmap 1` has type

```
{m : ℕ} → {A B : Set} → (A → B)
      → (Vec A m) → (Vec B m)
```

and the expression

```
nmap 1 (λ x → x + 1) (1 :: 2 :: 3 :: [])
```

evaluates to `2 :: 3 :: 4 :: []`. Likewise, the term `nmap 2` has type

```
{m : ℕ} → {A B C : Set} → (A → B → C)
      → (Vec A m) → (Vec B m) → (Vec C m)
```

and the expression

```
nmap 2 (λ _ _ → 1) (1 :: 2 :: 3 :: []) (4 :: 5 :: 6 :: [])
```

returns `(1, 4) :: (2, 5) :: (3, 6) :: []`.

2.3 Towards type genericity

We have shown how to define arity-generic map for vectors, but what about for other types of data, such as products of vectors or vectors of products? This should be possible, as map is a *type-generic* operation, one that is defined by type structure.

Type-generic programming in Agda is done via a technique called *universes* [Martin-Löf 1984, Nordström et al. 1990]. The idea is to define an inductive datatype `Tyc`, called a universe, which represents types, along with an interpretation function `[_]` that maps elements of this universe to actual Agda types. A generic program is then an operation which manipulates this data structure.

However, there is one difficulty—what kind of types should we represent? The answer to that question determines the type of the interpretation function. For example, if the datatype `Tyc` represents types of kind `Set` then the interpretation function should have type `Tyc → Set`. If the universe represents type constructors, that is, functions from types to types, then the interpretation function should have type `Tyc → (Set → Set)`.

Suppose we have such a universe for type constructors and a mapping of that universe into `Set → Set`. For example, to define generic operations over types composed of natural numbers, unit, products, sums, and vectors, we might use the following universe.

```
data Tyc : Set where
  Nat : Tyc
  Unit : Tyc
  Prod : Tyc → Tyc → Tyc
  Sum : Tyc → Tyc → Tyc
  Arr : ℕ → Tyc → Tyc
  Var : Tyc
```

Each of these codes can be decoded as an Agda type constructor of “kind” `Set → Set`. For example, `⊤` is the unit type in Agda, `×` constructs the (non-dependent) type of products, and `⊔` forms the type of disjoint unions.

```

[ ]      : Tyc → (Set → Set)
[ Nat ]  a = ℕ
[ Unit ] a = ⊤
[ Prod t1 t2 ] a = [ t1 ] a × [ t2 ] a
[ Sum t1 t2 ] a = [ t1 ] a ⊕ [ t2 ] a
[ Arr n t1 ] a = Vec ([ t1 ] a) n
[ Var ]  a = a

```

With these two definitions, we can implement type-generic versions of the `repeat` and `_*` functions.³ They are implemented by recursion on the structure of the `Tyc`, but we elide the definitions for brevity.

```

grepeat : (t : Tyc) → {a : Set} → a → [ t ] a
gzap    : (t : Tyc) → {a b : Set}
        → [ t ] (a → b) → [ t ] a → [ t ] b

```

With these type-generic functions, we can generalize the definition for vectors into one that works for all type constructors in the universe. This definition is a straightforward extension of `nvec-map` and `nmap` for vectors.

```

gvec-map : (t : Tyc) → (n : ℕ)
        → (As : Vec Set (suc n))
        → arrTy As → arrTy (repeat [ t ] * As)
gvec-map t n As f = g As (grepeat t f) where
g : {n : ℕ} → (As : Vec Set (suc n))
  → [ t ] (arrTy As)
  → arrTy (repeat [ t ] * As)
g {0} (A :: []) a = a
g {suc n} (A1 :: As) f =
  (λ a → g As (gzap t f a))
gmap : (t : Tyc) → (n : ℕ)
      → ∀⇒ (λ (As : Vec Set (suc n))
            → arrTy As → arrTy (repeat [ t ] * As))
gmap t n = λ⇒ (gvec-map t n)

```

For example:

```

example-map : {m : ℕ} → {A B : Set} → (A → B)
            → Vec (A × (A ⊕ ⊤)) m
            → Vec (B × (B ⊕ ⊤)) m
example-map =
  gmap (Arr _ (Prod Var (Sum Var Unit))) 1

```

We have orthogonally combined arity genericity and type genericity. However, there is a problem with this definition: it only works for type constructors of kind `Set → Set`. Maps for other kinds are not available. Furthermore, this definition tells us nothing about how to define other arity-generic functions. We have not really gotten to the essence of arity genericity.

To extend arity-generic map to types of arbitrary kinds, we will redo our framework for type-generic programming using a kind-indexed universe. With this kind-indexed universe, based on Generic Haskell, the concept of arity naturally shows up: a generic function has a kind-indexed type of a particular arity. For example, generic `repeat` requires an arity one kind-indexed type, generic map requires arity two, and generic `zipWith` requires arity three. Remarkably, but perhaps unsurprisingly, this notion of arity mirrors the arity found in arity-generic map!

What is new in this paper is that we generalize over the arities in the kind-indexed types to give a completely new definition of arity-generic type-generic map. This definition incorporates arity-genericity right from the start. In the current section we layered

³These generic functions show that all type constructors in this universe are applicative functors.

arity-genericity on top of type-genericity; in the next, our type-generic functions will be inherently arity-generic.

3. Arity-generic type-generic map

Next, we show how to generalize arity-generic map to arbitrary type constructors by implementing a framework for Generic Haskell style kind-indexed types.

3.1 Universe definition

To write more general generic programs, we need a more expressive universe. The universe that we care about is based on the type language of F-omega [Girard 1972]. It is the simply-typed lambda calculus augmented with a number of constants that form types. Therefore, to represent this language, we need datatypes for kinds, constants, and for the lambda calculus itself.

Kinds include the base kind `*` and function kinds. The function kind arrow associates to the right.

```

data Kind : Set where
  *      : Kind
  _⇒_    : Kind → Kind → Kind

```

A simple recursive function takes a member of this datatype into an Agda "kind".

```

[ ]      : Kind → Set
[ * ]    = Set
[ a ⇒ b ] = [ a ] → [ b ]

```

Constants are indexed by their kinds. For now, we will concentrate on types formed from natural numbers, unit, binary sums, and binary products. Because vectors are representable in terms of the other constructors, we do not include a code for them in this universe. This keeps the definitions of arity-generic functions simple. In Section 4, we discuss how our generic programming framework can interface directly with Agda datatypes like `Vec`.

```

data Const : Kind → Set where
  Nat  : Const *
  Unit : Const *
  Sum  : Const (* ⇒ * ⇒ *)
  Prod : Const (* ⇒ * ⇒ *)

```

Again, each of these constants can be decoded as an Agda type constructor.

```

interp-c : ∀ {k} → Const k → [ k ]
interp-c Unit = ⊤
interp-c Nat  = ℕ
interp-c Sum  = _⊕_
interp-c Prod = _×_

```

To represent other types (of arbitrary kinds), we now define an indexed datatype called `Typ`. A `Typ` may be a variable, a lambda, an application, or a constant. The datatype is indexed by the kind of the type and a typing context which indicates the kinds of variables. We use de Bruijn indices for variables, so we represent the typing context as a list of `Kinds`. The `n`th `Kind` in the list is the kind of variable `n`.

```

Ctx : Set
Ctx = List Kind
data TyVar : Ctx → Kind → Set where
  VZ : ∀ {G k} → TyVar (k :: G) k
  VS : ∀ {G k' k} → TyVar G k → TyVar (k' :: G) k
data Typ : Ctx → Kind → Set where
  Var : ∀ {G k} → TyVar G k → Typ G k
  Lam : ∀ {G k1 k2} → Typ (k1 :: G) k2

```

```

→ Typ G (k1 ⇒ k2)
App : ∀ {G k1 k2} → Typ G (k1 ⇒ k2) → Typ G k1
    → Typ G k2
Con  : ∀ {G k} → Const k → Typ G k

```

We use the notation `Ty` for closed types—those that can be checked in the empty typing context.

```

Ty : Kind → Set
Ty = Typ []

```

Now that we can represent type constructors, we need a mechanism to decode them as Agda types. To do so, we must have an environment that contains an Agda type for each of the variables. We index the datatype for the environment with the typing context to make sure that each variable is mapped to a type of the right kind. Note that this definition overloads the `[]` and `_::_` constructors, but Agda can infer which we mean.

```

data Env : List Kind → Set where
  []      : Env []
  _::_    : ∀ {k G} → [[ k ]] → Env G → Env (k :: G)
sLookup  : ∀ {k G} → TyVar G k → Env G → [[ k ]]
sLookup VZ (v :: G) = v
sLookup (VS x) (v :: G) = sLookup x G

```

Finally, with the help of the environment, we can decode a `Typ` as an Agda type of the appropriate kind. We use the `[_]` notation for decoding closed types in the empty environment.

```

interp : ∀ {k G} → Typ G k → Env G → [[ k ]]
interp (Var x) e = sLookup x e
interp (Lam t) e = λ y → interp t (y :: e)
interp (App t1 t2) e = (interp t1 e) (interp t2 e)
interp (Con c) e = interp-c c
[ ] : ∀ {k} → Ty k → [[ k ]]
[ t ] = interp t []

```

For example, if we call the following type constructor `Option` (it is isomorphic to the standard `Maybe` datatype)

```

Option : Set → Set
Option = λ A → T ⊕ A

```

we can represent it with the following code:

```

option : Ty (★ ⇒ ★)
option =
  Lam (App (App (Con Sum) (Con Unit)) (Var VZ))

```

The Agda type checker can see that `[option]` normalizes to `Option`, so it considers these two expressions equal.

3.2 Framework for doubly-generic programming

Next, we give the signature of a framework for defining arity-generic type-generic programs. For space reasons, we do not give the implementation of this framework here. The interested reader may consult Altenkirch and McBride [2003], Verbruggen et al. [2008], or our source code for more details.

As with Generic Haskell, the behavior of a generic program defined using this framework is fixed for applications, lambdas and variables. Therefore, to define an arity-generic type-generic operation, we need only supply the behavior of the generic program for the type constants.

Datatype-generic operations have different types when instantiated at different kinds, so they are described by *kind-indexed types* [Hinze 2002]. For example, consider the type of the standard map function for the `Option` type constructor, of kind $★ ⇒ ★$:

```

option-map1 : ∀ {A B} → (A → B)
             → (Option A → Option B)

```

And map for the type constructor `_ × _`, of kind $★ ⇒ ★ ⇒ ★$

```

pair-map1 : ∀ {A1 A2 B1 B2}
           → (A1 → B1) → (A2 → B2)
           → (A1 × A2) → (B1 × B2)

```

Though different, `option-map1` and `pair-map1` are instances of the same kind-indexed type. In Generic Haskell, kind-indexed types are defined by recursion on the kind of the type arguments. For example, here is the Generic Haskell definition of `map`'s type [Hinze and Jeurig 2003]:

```

type Map ⟨ ★ ⟩ t1 t2 = t1 → t2
type Map ⟨ k1 ⇒ k2 ⟩ t1 t2 =
  ∀ a1 a2, Map ⟨ k1 ⟩ a1 a2 → Map ⟨ k2 ⟩ (t1 a1) (t2 a2)

```

Readers new to Generic Haskell-style generic programming may find it instructive to verify that `Map ⟨ ★ ⇒ ★ ⟩ Option Option` and `Map ⟨ ★ ⇒ ★ ⇒ ★ ⟩ _ × _ × _` simplify to the types given above for `option-map` and `pair-map` (modulo notational differences).

For arity-genericity, we must generalize kind-indexed types in another way. We want not only `pair-map1`, but also `pair-map` at other arities to be instances as well:

```

pair-map0 : ∀ {A B : Set} → A → B → A × B
pair-map2 : ∀ {A1 B1 C1 A2 B2 C2}
           → (A1 → B1 → C1) → (A2 → B2 → C2)
           → A1 × A2 → B1 × B2 → C1 × C2

```

We compute the type of a generic function instance from four pieces of information: the arity of the operation (given with an implicit argument `n`), a function `b` to construct the type in the base case, the kind `k` itself and a vector `v` of `n` arguments, each of kind `k`. The kind-indexed type is written `b ⟨ k ⟩ v`:

```

_⟨_⟩_ : {n : ℕ}
      → (b : Vec Set (suc n) → Set)
      → (k : Kind)
      → Vec [[ k ]] (suc n)
      → Set

```

```

b ⟨ ★ ⟩ Vs = b Vs
b ⟨ k1 ⇒ k2 ⟩ Vs = ∀ ⇒ λ (As : Vec [[ k1 ]] _) →
  b ⟨ k1 ⟩ As → b ⟨ k2 ⟩ (Vs ⊗ As)

```

The definition of a kind-indexed type is based on that of Generic Haskell. The primary difference is that because the arity is a parameter, we deal with the type arguments as a vector rather than as individuals. For higher kinds the polymorphic type produced takes `n` arguments of kind `[[k1]]` (the vector `As`) and a kind-indexed type for those arguments and produces a result where each higher kinded type in the vector `Vs` has been applied to each argument in vector `As`.

We do not allow the vectors to be empty because few generic functions make sense at arity 0. As a result, the number provided here as an arity (`n`) is one less than the corresponding Generic Haskell arity. We refer to this reduced number as the arity for convenience.

We use the `∀ ⇒` function from section 2 to curry the type so that the user may provide `n` individual `[[k1]]`'s rather than a vector. The `_ in Vec [[k1]] _` instructs Agda to infer the length of the vector (convenient since we did not give a name to the arity).

We define generic functions with `ngen`, whose type is shown below. This operation produces a value of a kind-indexed type given `ce`, a mapping from constants to appropriate definitions.

```

ngen : { n : ℕ } { b : Vec Set (suc n) → Set } { k : Kind }
  → ( t : Ty k )
  → ( ce : TyConstEnv n b )
  → b ⟨ k ⟩ (repeat [ t ])

```

The type of `ce` is a function which maps each constant to a value of the kind-indexed type associated with that constant.

```

TyConstEnv : { n : ℕ } → ( b : Vec Set n → Set ) → Set
TyConstEnv b =
  { k : Kind } ( c : Const k ) → b ⟨ k ⟩ repeat [ Con c ]

```

We can already use this framework for non-arity-generic programming. For example, suppose we wished to define the standard generic map. In this case, we would provide the following definition for `b`.

```

Map : Vec Set 2 → Set
Map ( A :: B :: [] ) = A → B

```

Next, we define the type-constant environment for this particular `b`. The mapping function for natural numbers and unit is an identity function. For products and sums, the mapping function takes those arguments apart, maps the subcomponents and then puts them back together.

```

gmap-const : TyConstEnv GMap
gmap-const Nat   = λ x → x
gmap-const Unit  = λ x → x
gmap-const Prod  = λ fa fb x →
  ( fa (proj1 x), fb (proj2 x) )
gmap-const Sum   = g
  where
    g : { A1 B1 A2 B2 : Set }
      → ( A1 → B1 ) → ( A2 → B2 )
      → A1 ⊔ A2 → B1 ⊔ B2
    g fa fb (inj1 xa) = inj1 ( fa xa )
    g fa fb (inj2 xb) = inj2 ( fb xb )

```

Generic map then calls `ngen` with this argument.

```

gmap : { k : Kind } → ( t : Ty k )
  → Map ⟨ k ⟩ ([ t ] :: [ t ] :: [])
gmap t = ngen t gmap-const

```

Providing the type code instantiates generic map at particular types. For example, using the code for the `Option` type of the previous section, we can define:

```

option-map1 : { A B : Set } → ( A → B )
  → Option A → Option B
option-map1 = gmap option

```

3.3 Doubly-generic Map

To use `ngen` to implement a doubly-generic function, we must first describe the result by defining its type at kind `*` (`ngen`'s argument `b`) and then implement that function for the constants (`ce`). For doubly-generic map, we call these pieces `NGmap` and `ngmap-const`. `NGmap` is simply the `arrTy` function from Section 2.2.

```

NGmap : ( n : ℕ ) → Vec Set (suc n) → Set
NGmap n = arrTy { n }

```

To mirror the behavior of Generic Haskell, generic zips (which are generic maps at arities greater than one) should be partial functions. For example, Generic Haskell's library version of `zipWith` returns a `Maybe`, so that it may fail if instantiated at a sum type and passed mismatched injections. A version of arity-generic map which returns a `Maybe` is included with our sources, but to keep

the presentation here simple we use an error term in those cases instead. Since Agda lacks Haskell's error function, we use a postulate.

```

postulate error : ( A : Set ) → A

```

Next, we define the behavior of arity-generic type-generic map at the constant types. We do this by writing a term that dispatches to cases for the various constants (defined below).

```

ngmap-const : { n : ℕ } → TyConst (NGmap n)
ngmap-const { n } Nat   = defNat n
ngmap-const { n } Unit  = defUnit n
ngmap-const { n } Prod  = defPair n
ngmap-const { n } Sum   = defSum n

```

Recalling the definition of `NGmap` for kind `*`, we must pick arity-`n` functions with the types $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \dots \rightarrow \mathbb{N}$ and $\top \rightarrow \top \rightarrow \dots \rightarrow \top$. We pick a default element of each type to return arbitrarily. For `Nat`, we use three clauses to check that the inputs are identical for arities larger than one.

```

defNat : ( n : ℕ ) → NGmap n ⟨ * ⟩ (repeat ℕ)
defNat zero      = zero      -- arbitrary ℕ
defNat (suc zero) = λ x → x  -- return what was given
defNat (suc (suc n)) =
  λ x → λ y → if eqNat x y then defNat (suc n) y
                else error _
defUnit : ( n : ℕ ) → NGmap n ⟨ * ⟩ (repeat ⊤)
defUnit zero     = tt
defUnit (suc n)  = λ x → (defUnit n)

```

The `Prod` and `Sum` cases remain. Because these constants have higher kinds, the type of `ngmap-const` changes. Consider `Prod` first:

```

NGmap n ⟨ * ⇒ * ⇒ * ⟩ (repeat _ × _ ) =
  ∀ ⇒ λ ( As : Vec Set n ) → arrTy As →
  ∀ ⇒ λ ( Bs : Vec Set n ) → arrTy Bs →
  arrTy ((repeat ×) ⊗ As ⊗ Bs)

```

If we imagine writing out `As` as `A1 :: A2 :: ... :: An :: []` and `Bs` as `B1 :: B2 :: ... :: Bn :: []` the type simplifies to:

```

NGmap n ⟨ * ⇒ * ⇒ * ⟩ (repeat _ × _ ) =
  { A1 A2 ... An : Set } → ( A1 → A2 → ... → An )
  → { B1 B2 ... Bn : Set } → ( B1 → B2 → ... → Bn )
  → ( A1 × B1 ) → ( A2 × B2 ) → ... → ( An × Bn )

```

However, it is easier to define the case where the `A1 ... An` arguments are uncurried, and then curry the resulting function.

```

defPairAux : ( n : ℕ )
  → ( As : Vec Set (suc n) ) → arrTy As
  → ( Bs : Vec Set (suc n) ) → arrTy Bs
  → arrTy (repeat _ × _ ⊗ As ⊗ Bs)
defPairAux zero ( A :: [] ) a ( B :: [] ) b = ( a, b )
defPairAux (suc n) ( A1 :: As ) a ( B1 :: Bs ) b =
  λ p →
    ( defPairAux n As ( a (proj1 p) ) Bs ( b (proj2 p) ) )

```

In the zero case of `defPairAux`, `a` and `b` are arguments of type `A` and `B` respectively—the function must merely pair them up. In the successor case, `a` and `b` are functions with types `A1 → arrTy As` and `B1 → arrTy Bs`. We want to produce a result of type `A1 × B1 → arrTy (repeat _ × _ ⊗ As ⊗ Bs)`. Therefore, this case takes an argument `p` and makes a recursive call, passing in `a` applied to the first component of `p` and `b` applied to the second component of `p`. We use a kind-directed currying function `k-curry`, whose definition has been elided, to define the final version.

```

defPair : (n : ℕ)
  → NGmap n ⟨ * ⇒ * ⇒ * ⟩ (repeat _ × _)
defPair n = k-curry (* ⇒ * ⇒ *) (defPairAux n)

```

Sum also has kind $* \Rightarrow * \Rightarrow *$, so the type of its `ngmap-const` case is similar. However, for sums, we must check that the terms provided have the same structure (are either all `inj1` or all `inj2`). Otherwise, we signal an error.

Below, `defSumAux` checks if the first argument is an `inj1` or an `inj2`, then calls `defSumLeft` or `defSumRight` which require that all subsequent arguments match. In the degenerate case of an arity one map over sums, we arbitrarily choose the right injection (just as we chose the right injection in the Generic Haskell definition of empty in the introduction).

```

defSumLeft : (n : ℕ)
  → (As : Vec Set (suc n)) → arrTy As
  → (Bs : Vec Set (suc n))
  → arrTy (repeat _ ⊔ _ ⊗ As ⊗ Bs)
defSumLeft zero (A :: []) a (B :: []) = inj1 a
defSumLeft (suc n) (A1 :: As) a (B1 :: Bs) = f
  where
    f : A1 ⊔ B1 → arrTy (repeat _ ⊔ _ ⊗ As ⊗ Bs)
    f (inj1 a1) = defSumLeft n As (a a1) Bs
    f (inj2 b1) = defSumLeft n As (a (error A1)) Bs
defSumRight : (n : ℕ)
  → (As : Vec Set (suc n))
  → (Bs : Vec Set (suc n)) → arrTy Bs
  → arrTy (repeat _ ⊔ _ ⊗ As ⊗ Bs)
defSumRight zero (A :: []) (B :: []) b = inj2 b
defSumRight (suc n) (A1 :: As) (B1 :: Bs) b = f
  where
    f : A1 ⊔ B1 → arrTy (repeat _ ⊔ _ ⊗ As ⊗ Bs)
    f (inj1 a1) = defSumRight n As Bs (b (error B1))
    f (inj2 b1) = defSumRight n As Bs (b b1)
defSumAux : (n : ℕ)
  → (As : Vec Set (suc n)) → arrTy As
  → (Bs : Vec Set (suc n)) → arrTy Bs
  → arrTy (repeat _ ⊔ _ ⊗ As ⊗ Bs)
defSumAux zero (A :: []) a (B :: []) b =
  (inj2 b) -a rbitrary
defSumAux (suc n) (A1 :: As) a (B1 :: Bs) b = f
  where
    f : A1 ⊔ B1 → arrTy (repeat _ ⊔ _ ⊗ As ⊗ Bs)
    f (inj1 a1) = defSumLeft n As (a a1) Bs
    f (inj2 b1) = defSumRight n As Bs (b b1)

```

Finally, we also curry `defSumAux` to get the desired branch.

```

defSum : (n : ℕ)
  → NGmap n ⟨ * ⇒ * ⇒ * ⟩ (repeat _ ⊔ _)
defSum n = k-curry (* ⇒ * ⇒ *) (defSumAux n)

```

We can then define `ngmap` by instantiating `ngen`.

```

ngmap : (n : ℕ) → {k : Kind} → (e : Ty k)
  → NGmap n ⟨ k ⟩ (repeat [ e ])
ngmap n e = ngen e ngmap-const

```

If we had included vectors in our universe, we could simply use `nmap` from Section 2.2 for that case.

Just as datatype-generic functions are instantiated at a type, doubly generic functions are instantiated at an arity and a type. For example, given the definitions `Option` and `option` from the last section, we can define various maps for this type constructor:

```

option-map0 : {A : Set} → A → Option A
option-map0 = ngmap 0 option
option-map1 : {A B : Set} → (A → B)
  → Option A → Option B
option-map1 = ngmap 1 option
option-map2 : {A B C : Set} → (A → B → C)
  → Option A → Option B → Option C
option-map2 = ngmap 2 option

```

Of course, pair map functions are also instances of `ngmap`. We only show the definition of `pair-map2` below.

```

pair-map2 : {A1 B1 C1 A2 B2 C2 : Set}
  → (A1 → B1 → C1) → (A2 → B2 → C2)
  → A1 × A2 → B1 × B2 → C1 × C2
pair-map2 x y = ngmap 2 (Con Prod) x y

```

4. Datatype Isomorphisms

The infrastructure described so far permits us to instantiate arity-generic functions at different types based on their structure. However, to complete the story and generate versions of n-ary map for datatypes like `Vec`, we must make a connection between arbitrary datatypes and their structure. In this Section, we describe modifications to the implementation necessary to support generic functions on arbitrary datatypes through datatype isomorphisms.

4.1 Representing datatypes

There are at least two ways to support datatypes. The current system already can encode datatypes on an ad hoc basis, in a manner described by Verbruggen et al. [2008]. However, this encoding requires some tedious applications of coercions between the datatype and its isomorphism for each datatype instance of the generic operation. Instead, we can move that boilerplate to the generic function itself by adding a new constructor to the `Typ` universe. This new constructor, `Data`, contains information about a particular datatype definition.

```

data Typ : Ctx → Kind → Set where
  ...
  Data : ∀ {G} → DT G → Typ G *

```

The `DT` data structure contains four pieces of information about a datatype: its `Typ` representation `t`, the actual Agda datatype that this code represents `s`, and two functions for coercing between values of type `t` and values of type `s`.

```

data DT (G : Ctx) : Set where
  mkDT : (t      : Typ G *)
  → (s      : Env G → Set)
  → (to     : ({e : Env G} → interp t e → s e))
  → (from   : ({e : Env G} → s e → interp t e))
  → DT G

```

Note that we can only represent datatypes of kind `Set`. Other kinds do not support the coercion functions `to` and `from` as their interpretations have the wrong type. To create isomorphisms of type constructors like `Vec`, the `DT` datatype is parameterized by a context `G`, and `s` may depend on an environment for that context. We describe this mechanism in more detail below.

We define a number of accessor functions for retrieving the four parts of a `DT`, called `DT-s`, `DT-t`, `DT-from` and `DT-to` (here elided).⁴ Because `interp` is mentioned by the components of `DT`, it must be defined mutually with `Env`, `sLookup`, `Typ`, `DT`, and its accessors.

⁴Unfortunately, Agda does not support record definitions in a mutual block.

Finally, we extend the interpretation function for codes by looking up the Agda type and giving it the current environment.

```
interp : ∀ {k G} → Typ G k → Env G → [[ k ]]
...
interp (Data dt) e = DT-s dt e
```

For example, suppose we have a simple datatype definition that distinguishes some natural numbers as Oranges.

```
data Orange : Set where
  toOrange : ℕ → Orange
```

We can form the code for this datatype as below.

```
fromOrange : Orange → ℕ
fromOrange (toOrange x) = x
orange : {G : Ctx} → Typ G ★
orange = Data (mkDT
  (Con Nat)          -- t
  (λ e → Orange)    -- s
  (λ {e} → toOrange) -- to
  (λ {e} → fromOrange)) -- from
```

Even though the kind of a datatype isomorphism must be \star , we can still create isomorphisms for datatypes with higher kinds, such as `Maybe` and `Vec`. This works by creating an isomorphism with a "hole" (exploiting the fact that the environment need not be empty), then wrapping it in a lambda.

Instead of defining the structure of the `Maybe` type as a code with higher kind (i.e. something of type $\text{Ty } (\star \Rightarrow \star)$, such as option from Section 3.1), we instead define its structure as a function from codes to codes.

```
maybeDef : {G : Ctx} → Typ G ★ → Typ G ★
maybeDef t = (App (App (Con Sum) (Con Unit)) t)
```

The conversions to and from the `Maybe` type are also parameterized by the code of the argument to `Maybe`

```
toMaybe : {G : Ctx} {e : Env G}
  → (t : Typ G ★)
  → (interp (maybeDef t) e) → Maybe (interp t e)
toMaybe t (inj1 _) = nothing
toMaybe t (inj2 x) = just x
fromMaybe : {G : Ctx} {e : Env G} → (t : Typ G ★)
  → Maybe (interp t e)
  → (interp (maybeDef t) e)
fromMaybe t (just x) = inj2 x
fromMaybe t nothing = inj1 tt
```

Finally, we form the code of the datatype itself by wrapping the `Data` constructor in a `Lam` and using variable zero for the parameter. The environments supplied to `s`, `to` and `from` allow us to specify the type this variable corresponds to.

```
maybe : {G : Ctx} → Typ G (★ ⇒ ★)
maybe {G} = Lam (Data
  (mkDT (maybeDef (Var VZ))
    (λ e → Maybe (interp (Var VZ) e))
    (λ {e} → toMaybe {★ :: G} {e} (Var VZ))
    (λ {e} → fromMaybe {★ :: G} {e} (Var VZ))))
```

We can use this same idea to encode vectors. Because we know the length of given vector, it is isomorphic to an n -tuple—a sequence of products terminated by unit. The code for the vector type then abstracts both the code for the type of the elements of the vector and a natural number for its length.

```
vecDef : {G : Ctx} → Typ G ★ → (n : ℕ) → Typ G ★
vecDef _ 0 = Con Unit
vecDef t (suc n) =
  (App (App (Con Prod) t) (vecDef t n))
fromVec : {n : ℕ} {G : Ctx}
  {t : Typ G ★} {e : Env G}
  → Vec (interp t e) n → (interp (vecDef t n) e)
fromVec {0} [] = tt
fromVec {suc n} (x :: xs) = (x, fromVec xs)
toVec : {n : ℕ} {G : Ctx} {t : Typ G ★} {e : Env G}
  → interp (vecDef t n) e → Vec (interp t e) n
toVec {0} _ = []
toVec {suc n} (x, xs) = (x :: toVec xs)
vec : {G : Ctx} → {n : ℕ} → Typ G (★ ⇒ ★)
vec {G} {n} = Lam (Data
  (mkDT (vecDef (Var VZ) n)
    (λ e → Vec (interp (Var VZ) e) n)
    toVec
    fromVec))
```

4.2 Adding Data support to ngen

Although `Data` provides a mechanism for coding datatypes, we cannot use it to define generic functions until we extend `ngen` to handle `Data`. However, there is a complication—it is not clear how to do so. Without getting too much into the technicalities, the issue is that in this definition we need to produce a result of type⁵

```
b (repeat (DT-s dt) ⊗ envs)
```

but we only have a value of type

```
b (repeat (interp (DT-t dt)) ⊗ envs)
```

We would like to coerce the latter to the former using `to` and `from`, but we know nothing about `b`. Therefore, we require an additional argument to `ngen`, to be supplied when the generic operation is defined (i.e. when `b` is supplied.)

```
ngen : {n : ℕ} {b : Vec Set (suc n) → Set} {k : Kind}
  → (t : Ty k)
  → TyConstEnv n b
  → DataGen b
  → b ⟨ k ⟩ (repeat [ t ])
```

This argument, of type `DataGen`, shown below, is exactly the coercion function necessary.

```
DataGen : {n : ℕ} → (b : Vec Set (suc n) → Set) → Set
DataGen {n} b =
  {G : Ctx}
  → (dt : DT G)
  → (envs : Vec (Env G) (suc n))
  → b (repeat (interp (DT-t dt)) ⊗ envs)
  → b (repeat (DT-s dt) ⊗ envs)
```

As an example of an instance of `DataGen`, recall the definition of `ngmap` and its base type `NGmap` from Section 3.3.

```
arrTy : {n : ℕ} → Vec Set (suc n) → Set
arrTy {0} (A :: []) = A
arrTy {suc n} (A1 :: As) = A1 → arrTy As
NGmap : {n : ℕ} → Vec Set (suc n) → Set
NGmap = arrTy
```

⁵Here $b : \text{Vec Set } (\text{suc } n) \rightarrow \text{Set}_1$ describes the type of the generic operation and $\text{envs} : \text{Vec } (\text{Env } G) (\text{suc } n)$ is a vector of environments for the free variables.

The definition of the DataGen coercion for the case where b is NGmap , called `ngmap-data` below, proceeds by induction on the arity. In the base case of $n=0$, `ngmap-data` must coerce a result from the representation type to the Agda type using the `to` component.

For higher n , `ngmap-data` is provided with a vector of environments $e1 :: es$ and a function of type:

$$\text{interp } (\text{DT-t dt}) e1 \rightarrow \text{arrTy } (\text{repeat } (\text{DT-t dt}) \otimes es)$$

Its result type is:

$$(\text{DT-s dt}) e1 \rightarrow \text{arrTy } (\text{repeat } (\text{DT-s dt}) \otimes es)$$

This case takes in a $(\text{DT-s dt}) e1$, uses the `from` function to convert it to an `interp (DT-t dt) e1`, then coerces the result of the provided function by calling `ngmap-data` recursively.

```
ngmap-data : {n : ℕ} → DataGen (NGmap {n})
ngmap-data {0} dt (e :: []) bt =
  DT-to dt bt
ngmap-data {suc n} dt (e1 :: es) bt =
  λ x → ngmap-data {n} dt es
        (bt (DT-from dt x))
```

4.3 Using `ngen` at datatypes

With the `ngmap-data` function from the previous section, we may instantiate the updated `ngen` for NGmap .

```
ngmap : (n : ℕ) → {k : Kind} → (e : Typ [] k) →
  NGmap {n} ⟨ k ⟩ (repeat (interp e []))
ngmap n e = ngen e ngmap-const ngmap-data
```

This new `ngmap` adds support for datatypes. For example, we may use it with the `maybe` and `vec` representations of Section 4.1. Note that `vec-map0` is precisely the `repeat` function we have used throughout this paper.

```
maybe-map1 : {A B : Set} → (A → B)
  → Maybe A → Maybe B
maybe-map1 = ngmap 1 maybe
vec-map0 : {A : Set} {n : ℕ} → A → Vec A n
vec-map0 = ngmap 0 vec
vec-map1 : {A B : Set} {n : ℕ}
  → (A → B) → Vec A n → Vec B n
vec-map1 = ngmap 1 vec
```

Observe that instantiating `ngmap` at a datatype is no different than any other type we have seen. The codes for `Maybe` and `Vec` work for any generic operation. Although the definition of `ngmap` needed the `DataGen` argument, this argument is must be implemented once per generic operation, just like `TyConst`. In contrast, previous work [Verbruggen et al. 2008], could not define a general code for datatypes like `Maybe` and `Vec`, and required significant boilerplate at every instantiation of a generic function with a specific datatype.

5. Other arity-generic operations

Mapping is not the only arity-generic function. In this section, we examine two others.

5.1 Equality

We saw in Section 3.3 that arity-generic `map` must check that its arguments have the same structure. We can define n -ary equality itself in a similar manner. This function takes n arguments, returning `true` if they are all equal, and `false` otherwise.

In the specific case of vectors, this arity-generic function has a similar structure to n -ary `map`. Given an n -ary equality function for the type argument, we can define n -ary equality for vectors as:

```
nvec-eq : {m : ℕ} {A : Set}
  → (A → ... → A → Bool)
  → Vec A m → ... → Vec A m → Bool
nvec-eq f v1 ... vn = all (repeat f ⊗ v1 ⊗ ... ⊗ vn)
```

However, for other types, the definition is not the same as that of `map`. For example, the definition of equality for natural numbers returns `true` only if all three match:

```
nat-eq2 : ℕ → ℕ → ℕ → Bool
nat-eq2 = ngeq 2 (Con Nat)
```

Furthermore, we can generalize n -ary equality for type constructors. There, the type arguments need not be the same. We can pass any sort of comparison function in to examine the values carried by the data constructors. For example, the arity one instance for pairs is parameterized by equality functions on the pair components, since their types are left general:

```
pair-eq1 : {A1 B1 A2 B2 : Set}
  → (A1 → B1 → Bool) → (A2 → B2 → Bool)
  → (A1 × A2) → (B1 × B2) → Bool
pair-eq1 eq1 eq2 = ngeq 1 (Con Prod) eq1 eq2
```

If we instead instantiated `ngeq` at a specific pair type, it would recursively call itself to get equality functions for these arguments.

The definition of `ngeq` is similar to that of `ngmap`, so we will only highlight the differences.⁶ One occurs in the definition of the arity-indexed type, NGeq . This function returns a boolean value rather than one of the provided types, which means that `ngeq` makes sense even for $n = 0$. In that case its type is simply `Bool`.

```
toBool : {n : ℕ} → (v : Vec Set n) → Set
toBool {zero} [] = Bool
toBool {suc n} (A1 :: As) = A1 → toBool As
NGeq : {n : ℕ} → (v : Vec Set n) → Set
NGeq v = toBool v
```

Next we must define a `TyConstEnv` for NGeq . For simplicity, we only show the cases for `Unit` and `Nat`. The cases for `Prod` and `Sum` are straightforward variations of `ngmap`. As there is only a single member of the \top type, the case for `unit` is just a function that takes n arguments and returns `true`.

```
defUnit : (n : ℕ) → NGeq (repeat ⊤)
defUnit zero = λ x → true
defUnit (suc n) = λ x → defUnit n
```

For natural numbers, `ngeq` should compare each number and return `true` only when they all match (or when n is less than 2). We implement this by checking each argument for equality with the next. If a mismatch is found, `ngeq` uses `constFalse`, which consumes a given number of arguments and returns `false`.

```
constFalse : {n : ℕ} → (v : Vec Set n) → toBool v
constFalse {zero} [] = false
constFalse {suc m} (A1 :: As) = λ a → constFalse As
defNat : (n : ℕ) → NGeq (repeat {n} ℕ)
defNat zero = true
defNat (suc zero) = λ x → true
defNat (suc (suc n)) =
  λ x → λ y → if eqNat x y then defNat (suc n) y
                else constFalse (repeat ℕ)
```

⁶The complete definition may be found in `ngeq.agda` with our sources.

Finally, because we wish to use `ngeq` at various Agda datatypes, we must define an instance of `DataGen` from Section 4. As before, we go by recursion on the arity. Since `NGeq` is an n -ary function of representable types, we simply take in each argument, use the provided `DT` isomorphism to coerce it to the appropriate type, and recurse:

```
ngeq-data : {n : ℕ} → DataGen (NGeq {suc n})
ngeq-data {0} dt (e :: []) bt =
  λ s → bt (DT-from dt s)
ngeq-data {suc n} dt (e :: es) bt =
  λ s → ngeq-data dt es (bt (DT-from dt s))
```

With these pieces defined, the definition of `ngeq` is a straightforward application of `ngen`:

```
ngeq : (n : ℕ) → {k : Kind} → (e : Ty k) →
  NGeq ⟨ k ⟩ (repeat {suc n} (λ e []))
ngeq n e = ngen e ngeq-const ngeq-data
```

5.2 Splitting

The Haskell prelude and standard library include the functions

```
unzip  :: [(a, b)]      → ([a], [b])
unzip3 :: [(a, b, c)]  → ([a], [b], [c])
unzip4 :: [(a, b, c, d)] → ([a], [b], [c], [d])
unzip5 :: [(a, b, c, d, e)] → ([a], [b], [c], [d], [e])
unzip6 :: [(a, b, c, d, e, f)] → ([a], [b], [c], [d], [e], [f])
```

suggesting that there should be an arity-generic version of `unzip`. Furthermore, it makes sense that we should be able to `unzip` data structures other than lists.

Indeed, the Generic Haskell library includes `unzipWith` (here specialized to lists), which generates the standard `unzip` for arbitrary datatypes when passed the identity function.

```
unzipWith :: (a → (b, c, d)) → [a] → ([b], [c], [d])
```

Here, we describe the definition of `ngsplit`, which generates `unzip` for arbitrary data structures at arbitrary arities. In some sense, `ngsplit` is the dual to `ngmap`. Instead of taking in n arguments (with the same structure) and combining them together to a single result, `split` takes a single argument and distributes it to n results, all with the same structure.

For example, here is an instance of `ngsplit`, specialized to the `Option` type:

```
unzipWith2 : {A B C : Set} → (A → B × C)
  → (Option A → Option B × Option C)
unzipWith2 = ngsplit 2 option
```

The definition of `unzipWith` gives us `unzip` when applied to the identity function.

```
unzip2 : {A B : Set} → Option (A × B)
  → (Option A × Option B)
unzip2 = unzipWith2 (λ x → x)
```

The function `NGsplit` gives the type of `ngsplit` at base kinds. The first type in the vector passed to `NGsplit` is the type to split. For this reason, like `ngmap`, splitting does not make sense for an empty vector of types. The subsequent types are those the first type will be split into. If there is only one type, the function returns unit. The helper function `prodTy` folds the `_×_` constructor across a vector of types.

```
prodTy : {n : ℕ} → (As : Vec Set n) → Set
prodTy {0} _ = ⊤
prodTy {1} (A :: []) = A
prodTy {suc (suc _)} (A :: As) = (A × prodTy As)
```

```
NGsplit : {n : ℕ} → (v : Vec Set (suc n)) → Set
NGsplit (A1 :: As) = A1 → prodTy As
```

The cases for `Nat` and `Unit` are straightforward, so we do not show them. They simply make n copies of the argument.

To split a product (x, y) , we first split x and y , then combine together the results. For this combination, `prodn` takes arguments of types $(A1 × A2 × … × An)$ and $(B1 × B2 × … × Bn)$ and forms a result of type $(A1 × B1) × (A2 × B2) × … × (An × Bn)$.

```
prodn : {n : ℕ} → (As Bs : Vec Set n)
  → prodTy As → prodTy Bs
  → prodTy (repeat _×_ ⊗ As ⊗ Bs)
prodn {0} _ _ a b = tt
prodn {1} (A :: []) (B :: []) a b = (a, b)
prodn {suc (suc n)} (A :: As) (B :: Bs) (a, as) (b, bs) =
  ((a, b), prodn {suc n} _ _ as bs)
defPair : (n : ℕ)
  → (As : Vec Set (suc n)) → (NGsplit As)
  → (Bs : Vec Set (suc n)) → (NGsplit Bs)
  → NGsplit (repeat _×_ ⊗ As ⊗ Bs)
defPair n (A :: As) a (B :: Bs) b =
  λ p → prodn {n} _ _ (a (proj1 p)) (b (proj2 p))
```

The case for sums scrutinizes the argument to see if it is a left or right injection, and uses the appropriate provided function to split the inner expression. Then we use either `injLeft` or `injRight` (elided), which simply map `inj1` or `inj2` onto the members of the resulting tuple.

```
defSum : (n : ℕ)
  → (As : Vec Set (suc n)) → (NGsplit As)
  → (Bs : Vec Set (suc n)) → (NGsplit Bs)
  → NGsplit (repeat ⊕_ ⊗ As ⊗ Bs)
defSum 0 (A :: []) af (B :: []) bf = λ _ → tt
defSum (suc n) (A :: As) af (B :: Bs) bf = f
  where f : A ⊕ B → prodTy (repeat ⊕_ ⊗ As ⊗ Bs)
        f (inj1 x1) = injLeft {n} (af x1)
        f (inj2 x1) = injRight {n} (bf x1)
```

The definition of `split-const` (elided) dispatches to the branches above in the standard way, delegating to a trivial case when n is 0. Finally, we must define an instance of `DataGen` so that we may use `ngsplit` at representable Agda datatypes. Since `NGsplit` is defined in terms of `prodTy`, we must also convert instances of that type. These (elided) functions are similar to previous examples, except that we are converting a pair instead of an arrow. With `split-const` and `split-data`, we can define `ngsplit` as usual.

Splitting is a good example of datatype-generic programming's potential to save time and eliminate errors. Defining a separate instance of `split` for Vectors would be involved. For example, we would need functions to transpose vectors of products, transforming `Vec m (A1 × A2 × … × An)` into `(Vec A1 m × Vec A2 m × … × Vec An m)`. This code is slightly tricky and potentially error-prone, but we get the `Vec` instances of `split` for free with generic programming, and do not need to reason separately about their correctness.

5.3 Other doubly-generic operations

Mapping, equality and splitting provide three worked out examples of doubly generic functions. We know of a few others, such as a monadic map, a map that returns a `Maybe` instead of an error when the `Sum` injections do not match, a comparison function, and an equality function that returns a proof that the arguments are all equal. Furthermore, there are arity-generic versions of standard Generic Haskell functions like `crushes` or `enumerations`. For example, an arity-generic `gsum` adds together all of the numbers found

in n data structures. Such examples seem less generally useful than arity-generic map or unzip, but are not difficult to define.

Compared to the space of datatype-generic functions, the space of doubly generic operations is limited. This is unsurprising, as there already were not many examples of Generic Haskell functions with arities greater than one. However, this work has given us new insight into what other arity-generic functions might look like. Furthermore, though the collection of arity-generic functions is small, this is no reason not to study it: indeed, it includes some of the most fundamental operations of functional programming, and it makes sense that we should learn as much as we can about these operations.

6. Related work

Only a few sources discuss arity-generic programming. Fridlender and Indrika [2000] show how to encode n -ary list map in Haskell, using a Church encoding of numerals to reflect the necessary type dependencies. They remark that a generic programming language could provide a version of `zipWith` that works for arbitrary datatypes, but that no existing language provides such functionality. They also mention a few other arity-generic programs: `taut` which determines whether a boolean expression of n variables is a tautology, and variations on `liftM`, `curry` and `uncurry` from the Haskell prelude. It is not clear whether any of these functions could be made datatype-generic. McBride [2002] shows an alternate encoding of arity-generic list map in Haskell using type classes to achieve better safety properties. He examines several other families of operations, like `crush` and `sum`, but does not address type genericity.

Sheard [2006] translates Fridlender and Indrika's example to the Ω mega programming language, using that language's native indexed datatypes instead of the Church encoding. He also demonstrates one other arity-generic program, an n -ary version of addition. Although the same work also includes an implementation of datatype-generic programming in Ω mega, the two ideas are not combined.

Several researchers have used dependent types (or their encodings) to implement Generic-Haskell-style datatype-genericity. In previous work, we encoded representations of types using Church encodings [Weirich 2006a] and GADTs [Weirich 2006b] and showed how to implement a number of datatype-generic operations such as `map`. Hinze [2006], inspired by this approach, gave a similar encoding based on type classes. In those encodings, doubly-generic programming is not possible because datatype-generic programs of different arities require different representations or type classes.

The most closely related encoding of Generic Haskell to this one is by Verbruggen et al. [2008]. They use the Coq programming language to define a framework for generic programming, but do not consider arity-genericity. Altenkirch and McBride [2003] show a similar development in Oleg. This work extends those developments by considering examples not possible in Generic Haskell and showing a technique for writing generic programs which work on source-language datatypes.

The idea of generic programming in dependent type theory via universes has seen much attention since it was originally proposed [Martin-Löf 1984, Nordström et al. 1990]. While demonstrating a new form of double genericity, this paper covers only one part of what is possible in a dependently typed language. In particular, our codes do not extend to all inductive families and so we cannot represent all types that are available (see Benke et al. [2003] and Morris et al. [2007] for more expressive universes.) A dependently-typed language also permits the definition of generic proofs about generic programs. Chlipala [2007] uses this technique in the Coq proof assistant to generically define and prove substitution properties of programming languages. Verbruggen et al. [2009]

use Coq's dependent types to develop a framework for proving properties about generic programs.

7. Discussion

Generic programming in a dependently-typed language As we mentioned in the introduction, there are several dependently-typed languages that we could have used for this development. We selected Agda because the focus of its design has been this sort of programming. Like Coq, Agda is a full-spectrum dependently typed language. That has allowed us the flexibility to use universes to directly implement generic programming. We had the full power of the computational language available to express the relationships between values and types. A phase-sensitive language, such as Ω mega or Haskell, would have required singletons to reflect computation to the type level, and would have permitted type-level computation only in a restricted language.

Compared to Coq, Agda has more vigorous type inference, especially combined with pattern matching. Although Coq can also infer implicit arguments, if we had written the functions in Coq we would have had to add many more type annotations. Additionally, developing in Agda allowed us to deal with non-termination more conveniently—while Coq must be able to see that a definition terminates before moving on, Agda shows the user where it can not prove termination and allows other work to continue.

On the other hand, using Coq would have led to two advantages. Coq's tactic language can be used to automate some of the reasoning. Tactics would have been particularly useful in proving some of the equalities needed to typecheck the (elided) implementation of `ngen`. However, we did not see any need for tactics in any of the *uses* of `ngen` to define doubly-generic operations. More importantly, as discussed below, differences in the way Coq and Agda handle type levels forced us to use Agda's `--type-in-type` flag to clarify the presentation.

Type levels in Agda Although we have hidden it, Agda actually has an infinite hierarchy of type levels. `Set`, also known as `Set0`, is the lowest level in the type hierarchy. Terms like `Set0` and `Set0 → Set0` have type `Set1`, which itself has type `Set2`, and so on.

To simplify our exposition, we collapsed all of these levels to the type `Set`, with the help of the `--type-in-type` flag. This flag makes Agda's logic inconsistent, so to demonstrate that we are not using it in an unsound way, we have also implemented a version of the code that may be compiled without the flag. That version can be found in the `noTypeInType` subdirectory of our source tarball.

Three differences between Coq and Agda make this explicit version more complicated than the one presented here. First, Agda currently lacks *universe polymorphism*, a feature which allows definitions to work on multiple type levels. As a result, many of the data structures in this paper must actually be duplicated at the level of `Set1`, creating significant clutter. Second, since `Set` is not impredicative in Agda, many definitions that could live at the level of `Set` in Coq must be at the level of `Set1` instead. Finally, because `Set0` is not a subtype of `Set1` in Agda, we found it necessary to explicitly lift types from `Set0` to `Set1`.

Future work and Conclusions Because we are working in the flexible context of a dependently-typed programming language, our work here will allow us to adapt and extend orthogonal results in generic programming to this framework. For example, we would like to use Agda as a proof assistant to reason about the properties of the generic programs that we write. We would also like to extend our universe so that it may encode more of Agda's type system, such as arbitrary indexed datatypes. Finally, we would like to gain more experience with arity-generic programming by creating and analyzing additional examples.

In this paper, we have combined arity-generic and datatype-generic programming into a single framework. Crucially, this combination takes advantage of the natural role that arities play in the definition of kind-indexed types. This framework has provided us with new understanding of the definition and scope of doubly-generic programs.

Acknowledgments Thanks to Andres Löf and Tim Sheard for discussion. This paper was generated from literate Agda sources using `lhs2TeX`.

References

- Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Generic Programming*, 2003. Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl, July 2002.
- Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic J. of Computing*, 10(4):265–289, 2003. ISSN 1236-6064.
- Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 241–253, New York, NY, USA, 2005. ACM. ISBN 1-59593-064-7. doi: <http://doi.acm.org/10.1145/1086365.1086397>.
- Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. *SIGPLAN Not.*, 42(6):54–65, 2007. ISSN 0362-1340. doi: <http://doi.acm.org/10.1145/1273442.1250742>.
- Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löf, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Utrecht University, 2001.
- Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Loeh. Extensible and modular generics for the masses. In Henrik Nilsson, editor, *Trends in Functional Programming (TFP 2006)*, April 2007. URL <http://www.comlab.ox.ac.uk/people/Bruno.Oliveira/extensibleGM.pdf>.
- Daniel Fridlender and Mia Indrika. Do we need dependent types? *Journal of Functional Programming*, 10(4):409–415, July 2000.
- Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.
- Ralf Hinze. Generics for the masses. *J. Funct. Program.*, 16(4-5): 451–483, 2006. ISSN 0956-7968. doi: <http://dx.doi.org/10.1017/S0956796806006022>.
- Ralf Hinze. Polymorphic values possess polykinded types. *Science of Computer Programming*, 43(2-3):129–159, 2002. MPC Special Issue.
- Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and theory. In Roland Backhouse and Jeremy Gibbons, editors, *Generic Programming, Advanced Lectures*, volume 2793 of *Lecture Notes in Computer Science*, pages 1–56. Springer-Verlag, 2003.
- Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis-Napoli, 1984.
- Conor McBride. Faking it simulating dependent types in haskell. *Journal of Functional Programming*, 12(5):375–392, 2002. ISSN 0956-7968. doi: <http://dx.doi.org/10.1017/S0956796802004355>.
- Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004. ISSN 0956-7968. doi: <http://dx.doi.org/10.1017/S0956796803004829>.
- Conor McBride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008. ISSN 0956-7968. doi: <http://dx.doi.org/10.1017/S0956796807006326>.
- Peter Morris, Thorsten Altenkirch, and Neil Ghani. Constructing strictly positive families. In *CATS '07: Proceedings of the thirteenth Australasian symposium on Theory of computing*, pages 111–121, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc. ISBN 1-920-68246-5.
- B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: an introduction*. Oxford University Press, 1990.
- Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.
- Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. <http://www.haskell.org/definition/>.
- Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming (ICFP)*, pages 50–61, Portland, OR, USA, September 2006.
- Tim Sheard. Generic programming programming in omega. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 258–284. Springer, 2006. ISBN 978-3-540-76785-5.
- Tim Sheard. Putting Curry-Howard to work. In *Proceedings of the ACM SIGPLAN 2005 Haskell Workshop*. ACM Press, September 2005.
- The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.1*. LogiCal Project, 2006. Available from <http://coq.inria.fr/V8.1beta/refman/>.
- Wendy Verbruggen, Edsko de Vries, and Arthur Hughes. Polymorphic programming in Coq. In *WGP '08: Proceedings of the ACM SIGPLAN workshop on Generic programming*, pages 49–60, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-060-9. doi: <http://doi.acm.org/10.1145/1411318.1411326>.
- Wendy Verbruggen, Edsko de Vries, and Arthur Hughes. Polymorphic properties and proofs in Coq. In *WGP '09: Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming*, pages 1–12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-510-9. doi: <http://doi.acm.org/10.1145/1596614.1596616>.
- Stephanie Weirich. Type-safe run-time polymorphic programming. *Journal of Functional Programming*, 16(10):681–710, November 2006a.
- Stephanie Weirich. RepLib: A library for derivable type classes. In *Haskell Workshop*, pages 1–12, Portland, OR, USA, September 2006b.