Generic Programming with Dependent Types

Stephanie Weirich and 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 familiar functions from the Haskell standard library (such as repeat, map, and zipWith).
This tutorial explores these forms of genericity individually and together. These two axes are not orthogonal: datatype-generic versions of repeat, map and zipWith have different arities of kind-indexed types. We explore these forms of genericity in the context of the Agda programming language, using the expressiveness of dependent types to capture both forms of genericity in a common framework. Therefore, this tutorial serves as an introduction to dependently typed languages as well as generic programming.
The target audience of this work is someone who is familiar with functional programming languages, such as Haskell or ML, but would like to learn about dependently typed languages. We do not assume prior experience with Agda, type- or arity-generic programming.

1 Introduction

Generic programming is about doing more with less. It is about saving time, so that the same piece of code can be used over and over again. It is about making the similarities between programs formal, so that the relationships between common functions are apparent. And it is about capturing the essence of an algorithm, no matter how complicated, so that a programmer needs only to fill in the details to use it.

Functional programmers use genericity. Every time that they use map or fold to capture the recursive behavior of a function and every time they use parametric polymorphism to abstract the type of an operation, they are doing generic programming.

However, there are ways to generalize code beyond higher-order functions and parametric polymorphism. For example, datatype-generic functions operate based on the type structure of data, so they need not be redefined for each new datatype definition. Generic Haskell [12, 8] includes a generic mapping operation, called gmap, that has instances for types such as lists, optional values, and products (even though these type constructors 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 these instances are generated from the same definition, reasoning about gmap tells us about mapping at each type. Other examples of datatype-generic operations include serialization, structural equality, and folds.

Likewise, arity genericity allows functions to be applied to a variable number of arguments. For example, we can also generalize map in this way. Consider the following sequence of functions from the Haskell Prelude [23], all of which operate on lists.

```
repeat   :: a                         → [a]
map      :: (a → b)                → [a] → [b]
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. Similarly, zipWith3 combines three lists. As Fridlender and Indrika [9] 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.

Generic programming is a natural example of *dependently-typed programming*. The features of dependently-typed languages, such as type-level computation and type-refining pattern matching, directly support the definition generic operations such as above.

In this tutorial, we show how to implement datatype genericity and arity genericity in the Agda programming language [22]. Embedding both of these ideas in the same context has an added benefit—it demonstrates the relationship between them. Map is an example of a function that is both datatype-generic and arity-generic; we call it *doubly generic*. Other functions also have both datatype-generic and arity-generic versions; map has an inverse operation called unzipWith that is doubly generic, and equality can be applied to any number of arguments of the same type. Other examples include folds, enumerations and monadic maps.

In fact, arity genericity is not independent of datatype genericity. Generic Haskell has its own notion of arity and each datatype-generic function must be defined at a particular arity. Importantly, that arity corresponds exactly to the arities in map above—the Generic Haskell version of repeat has arity one, its map has arity two, and zipWith arity three. What is missing is that Generic Haskell does not permit generalizing over arities, so a single definition cannot produce repeat, map and zipWith.

This tutorial demonstrates that it is possible to implement these doubly generic functions with a single definition in a dependently typed programming language. In particular, we describe a reusable generic programming framework similar to those discussed, and we show how it supports doubly generic definitions. We have chosen the language Agda, but we could have also used a number of different languages, such as Coq [29], Epigram [19], Ωmega [25], Cayenne [2] or Haskell with recent extensions [24, 5].

However, the goals of this tutorial are broader than doubly generic programming. The target audience of this work is someone who is familiar with functional programming languages, such as Haskell or ML, but would like to learn about dependently typed languages. We do not assume prior experience with Agda[1] or with generic programming. In that context, this tutorial demonstrates the expressive power of dependent types. Although many existing examples of the uses of dependent types are for verification—using precise types to rule out programs that contain bugs—we want to emphasize that dependent types can be used for much more than capturing program invariants. The message of this tutorial is that dependent type systems naturally support generic programming. This leads to more flexible interfaces, eliminates boilerplate and draws connections between common patterns within software.

This tutorial is based on the paper "Arity-generic type-generic programming" which appeared at the workshop Programming Languages meets Program Verification (PLPV 2010) [36] and lectures titled "Generic programming with dependent types" presented at the Spring School on Generic and Indexed Programming, held in Oxford, March 2010 [35]. All code described in this paper is available from `http://www.seas.upenn.edu/~sweirich/papers/aritygen-lncs.tar.gz` and has been tested with Agda version 2.2.10.

2 Simple Type-Generic Programming in Agda

Agda has a dual identity. It is both a functional programming language with dependent types, based on Martin-Löf intuitionistic type theory [17], and a proof assistant. Under the Curry-Howard Isomorphism, proofs are programs and propositions are types. Historically, Agda is derived from a series of proof assistants and languages implemented at Chalmers Institute of Technology in Gothenburg, Sweden. The current version, officially named "Agda 2", was implemented by Ulf Norell [22]. In this tutorial, we use the name Agda to refer to the current version.[2]

Here, we will focus exclusively on Agda's role as a dependently typed programming language. In fact, we will be using Agda in a nonstandard way, giving it three flags `-type-in-type`, `-no-termination-check`, and `-no-positivity-check`

[1] For more information on the Agda language, including installation instructions, manuals and other tutorials, see the Agda Wiki at `http://wiki.portal.chalmers.se/agda/`.

[2] The name Agda comes from a Swedish song about Agda the Hen, a pun on the Coq rooster.

that change its type checker. With these flags enabled, Agda cannot be used as a proof assistant. Instead, its semantics is similar to the programming languages Epigram [19], and Cayenne [2]. We discuss the implications of these flags in more detail in Section 8.

Because Agda is a *full-spectrum* dependently-typed language, terms may appear in types, and in fact, there is no syntactic distinction between the two. However, despite this significant difference, many basic Agda concepts and syntax should appear familiar to functional programmers.

For example, we may define a datatype for booleans with the following code. It creates the type Bool and its two data constructors true and false.

```
data Bool : Set where
  true  : Bool
  false : Bool
```

Note that Agda uses a single colon for type annotations. Furthermore, all Agda identifiers can be used as datatype constructors and by convention they are uncapitalized in Agda.

The type of Bool itself is Set, the analogue of Haskell's kind $\star$. Even though Agda does not syntactically distinguish between types and terms, we know that Bool is a "type" because it has type Set.

Like other functional programming languages, we can define functions over booleans by pattern matching. For example, we can define the standard negation operation as follows.

```
¬ : Bool → Bool
¬ true  = false
¬ false = true
```

Agda supports infix operators and unicode symbols in identifiers. Consider the following definition of infix boolean conjunction, where the underscores around the $\wedge$ in the type signature indicate that it is an infix operator.

```
_∧_  : Bool → Bool → Bool
true ∧ true  = true
_    ∧ _     = false
```

Agda also supports "mixfix" identifiers. Below, the underscores in if_then_else indicate that it takes two arguments that should appear between if and then and between then and else.

```
if_then_else : ∀ {A} → Bool → A → A → A
if true  then a1 else a2 = a1
if false then a1 else a2 = a2
```

Like Haskell, if_then_else is a polymorphic function, available for any result type A. However, unlike Haskell, the type of if_then_else must explicitly quantify over

A. The curly braces around A indicate that it is an *implicit argument* that does not participate in the pattern matching and that Agda should try to infer it when if_then_else is used. For example, one need not supply this type argument in an if expression such as below.

```
if true then 1 else 2
```

Like functional programming languages, Agda also includes recursive datatypes, such as natural numbers and lists. For convenience, Agda allows users to abbreviate values of the ℕ datatype, such as suc (suc zero), with their corresponding Arabic numbers, such as 2. Both of these definitions are from the Agda standard library. As usual, the list type below is *parameterized* by A, the type of the values stored in the list. (One trickiness of Agda is that the cons data constructor (::) is notated with a single unicode character.)

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

data List (A : Set) : Set where
  []    : List A
  _::_  : A → List A → List A
```

Functions over these datatypes can again be defined via pattern matching. For example, the following function constructs a list with n copies of its argument.

```
replicate : ∀ {A} → ℕ → A → List A
replicate zero    x = []
replicate (suc n) x = x :: replicate n x
```

One of the most powerful features of dependently typed languages is the ability to define indexed datatypes whose types depend on terms. For example, we may define a type of vectors, which are lists that know their own length. Unlike Haskell, Agda permits overloading of data constructors such as [] and _::_ and can tell from the context what sort of list should be constructed.

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

Like List, Vec is parameterized by the type A, indicating the type of data stored in the list. Vec is also *indexed* by an argument of type ℕ. This number records the length of the vector and varies in the types of the constructors. For example, empty vectors [] use index 0. Cons (written _::_) takes an implicit argument n that is the length of the tail of the list. Usually, Agda can infer this length. The term true :: false :: [] has type Vec Bool 2.

By indexing lists in this way, we can give informative types to functions. For example, compare the definitions of replicate above and repeat below. Both of these functions construct a value with n copies of its argument. However, the type of the repeat function makes explicit that the length of the output vector will be n.

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

Because n appears in the result type of repeat, it makes sense to declare that n is an implicit argument by putting it in curly braces. If the context of a call to repeat determines the length of the vector that is required, type inference will often be able to automatically supply that argument. If this process fails, the argument can be explicitly provided in curly braces.

2.1 Basic Type-Generic Programming

Using pattern matching, it is simple to define equality tests for the datatypes we have seen above. For example, equality functions for booleans and natural numbers can be defined as follows.

```
eq-bool : Bool → Bool → Bool
eq-bool true  true   = true
eq-bool false false  = true
eq-bool _     _      = false
eq-nat : ℕ → ℕ → Bool
eq-nat zero    zero    = true
eq-nat (suc n) (suc m) = eq-nat n m
eq-nat _       _       = false
```

In fact, to determine the equality of booleans or natural numbers we must define such functions. Agda does not include a built-in structural equality function (like Scheme or OCaml) nor does it include an equality type class (like Haskell). It is somewhat annoying to define and use equality functions for datatypes like these, because they follow a very regular pattern. Functions such as structural equality motivate *type-generic* programming, which allows programmers to define functions that observe and make use of the structure of types.

In a dependently typed language, type-generic programming is accomplished using *universes* [17, 21]. The idea is to define an inductive datatype Type, called a universe, along with an interpretation function ⌊__⌋ that maps elements of this universe to actual Agda types. Each element of Type can be thought of as a "code" for a particular type, and pattern matching gives us access to its structure. A generic program is then an operation that manipulates this structure to define an operation at different types.

For example, here is a very simple universe of types composed of natural number, boolean and product types.

```
data Type : Set where
  TNat  : Type
  TBool : Type
  TProd : Type → Type → Type
```

In Agda, types are first-class values (of type Set), so it is simple to define the interpretation function ⌊_⌋ for this universe. For example, we would like ⌊ TProd TNat TBool ⌋ to evaluate to ℕ × Bool. Here, _×_ is the type of non-dependent pairs in Agda.

```
⌊_⌋ : Type → Set
⌊ TNat ⌋         = ℕ
⌊ TBool ⌋        = Bool
⌊ TProd t1 t2 ⌋  = ⌊ t1 ⌋ × ⌊ t2 ⌋
```

Now we can define a basic type-generic equality function by dispatching on the universe. If the given code is for natural numbers or booleans, geq below uses the equality functions defined above. If it is the code for a product type, then geq calls itself recursively. Observe that the second and third arguments in the type of geq depend on the first argument. This function requires dependency to express its type.

```
geq : (t : Type) → ⌊ t ⌋ → ⌊ t ⌋ → Bool
geq TNat         n1      n2      = eq-nat n1 n2
geq TBool        b1      b2      = eq-bool b1 b2
geq (TProd a b) (a1,b1) (a2,b2) = geq a a1 a2 ∧ geq b b1 b2
```

We can use this function by supplying it the appropriate code. For example,

```
geq (TProd TNat TBool) (1,false) (1,false)
```

evaluates to true. Unfortunately, we can not make this function's universe argument t implicit because Agda can not derive the code for a type from the type.

In subsequent sections we will consider many more examples of type-generic functions (such as map, size, unzip) and define a larger universe representing many more types. Before we do that, however, we consider the other half of double genericity.

3 Arity-Generic Programming

Arity-generic functions generalize over the number of arguments that they take. For example, the sequence of map-like functions over lists shown in the introduction are all instances of Scheme's map function. Other examples of arity-generic functions include Scheme's +, foldl and foldr functions. A recent survey of the PLT Scheme code base found 1761 definitions for variable-arity functions [28].

What makes arity-generic functions so rare in statically typed languages like Haskell and ML is their type. It is difficult for the type systems of these languages to give them a type which allows them to be applied to any number of arguments. It can be done through clever encodings [9], or by extending the type systems [28]. Here we show how to use dependent types to describe the type of arity-generic functions.

The challenge for this section is to generalize the following sequence of functions into one definition. We will use different arities of maps for length-indexed vectors, defined above. The definitions of these 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
```

The function repeat is the same function that we defined in Section 2. The other operation, _⊛_, is an infix zipping application, pronounced "zap" for "zip with apply," defined below. These two functions are the components of the Applicative type class in Haskell. We employ these functions to define map1, which is the standard map for vectors, and map2, which is an analogue of Haskell's zipWith.

```
_⊛_ : {A B : Set} {n : ℕ} → Vec (A → B) n → Vec A n → Vec B n
[]        ⊛ []        = []
(a :: As) ⊛ (b :: Bs) = a b :: As ⊛ Bs
infixl 40 _⊛_
```

The last line of the Agda code declares the precedence value of _⊛_ operator and associates it to the left. In its definition, we do not need to consider the case where one vector is empty while the other is not because the type specifies that both arguments have the same length.

Intuitively, each map above is defined by a simple application of repeat and n copies of _⊛_. Let us call the arity-generic version nvec-map.

```
nvec-map f n v1 v2 ... vn = repeat f ⊛ v1 ⊛ v2 ⊛ ... ⊛ vn
```

We can define this function by recursion on n in accumulator style as sketched below. After repeating f we have a vector of functions, we then zap this vector across n argument vectors, using the helper function g.

```
nvec-map n f = g n (repeat f) where
  g 0       a = a
  g (suc n) f = (λ a → g n (f ⊛ a))
```

Although the definition is straightforward, the code above does not type check in Agda. It requires additional typing annotations that express how the type of nvec-map depends on the argument n. Agda can express the arity-generic operation that unifies all of these maps via dependent types, as we present below.

3.1 Typing 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 the sequence above. Part of the difficulty is that our generic function is curried in both its type and term arguments. In this subsection, will start with an initial definition that takes all of the type arguments together (in a vector), but curries the term arguments. In the next subsection, we then demonstrate how to uncurry the type arguments.

We use natural numbers to express the arity of the mapping operation. Therefore, we must *program* with Agda types, taking advantage of the fact that types are first-class data. For example, we store types in data structures, such as a vector of Agda types, Bool :: ℕ :: []. This vector itself has type Vec Set 2, so we can use standard vector operations (such as _⊛_) with it. Our development uses the Agda flag `--type-in-type`, which makes this typing possible by giving Set the type Set. This flag allows us to simplify our presentation by hiding Agda's infinite hierarchy of Set levels, at the cost of making Agda's logic inconsistent. We discuss this choice further in Section 8.

The first step towards defining the type of nvec-map is to define arrTy, which folds the arrow type constructor → over a non-empty vector of types. For example, arrTy (ℕ :: ℕ :: Bool :: []) should return ℕ → ℕ → Bool. This operation constructs the type of the first argument to nvec-map, the function to map over the n vectors.

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

Next, the function arrTyVec constructs the result type of arity-generic map for vectors. We define this operation 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. Recall that the curly braces in the types of arrTyVec and arrTy mark n as an implicit argument, so we need not always match against it in definitions nor provide it explicitly as an argument.

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

For example, we can define the sequence of types from Section 2 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 put these pieces together. The type of nvec-map 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 can curry it to get the desired type.

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

Now we can complete the definition of nvec-map. We make two small changes from the code presented above. First, we add a type annotation for the helper function g, using the arrTy and arrTyVec functions. Second, we add an explicit pattern match on the vector of types in g. This allows Agda to unfold the definitions of arrTy and arrTyVec when type checking g's branches.

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

This function can be used as is. For example, we can use an arity 1 map to add 10 to each natural number in a vector. The term

```
nvec-map 1 {ℕ :: ℕ :: []} (λ x → 10 + x) (1 :: 5 :: [])
```

evaluates to (11 :: 15 :: []). Note that we must explicitly supply the types because Agda cannot infer them from the other arguments to nvec-map. However, supplying these types explicitly as a vector to nvec-map is annoying. To help Agda infer them, we define some general currying functions in the next subsection.

3.2 A Curried Vector Map

To make nvec-map more convenient, we will curry the type arguments so that they are supplied individually rather than in a vector. Then, Agda will usually be able to infer them. For this, we need two functions. The first, ∀⇒, creates a curried version of a type which depends on a vector. The second, $\lambda\Rightarrow$, curries a corresponding function term.

```
∀⇒ : {n : ℕ} → ((_ : Vec Set n) → Set) → Set
∀⇒ {zero}  B = B []
∀⇒ {suc n} B = {a : Set} → ∀⇒ (λ as → B (a :: as))
λ⇒ : {n : ℕ} → {B : (_ : Vec Set n) → Set}
    → ({X : Vec Set n} → B X) → (∀⇒ B)
λ⇒ {zero}  f = f {[]}
λ⇒ {suc n} f = λ {a : Set} → λ⇒ {n} (λ {as} → f {a :: as})
```

With these operations, we can finish the definition of arity-generic map. Again, the (implicit) argument m is the length of the term vectors, and the (explicit) argument n is the specific arity of map that is desired.

```
nmap : {m : ℕ} → (n : ℕ)
      → ∀⇒ (λ (As : Vec Set (suc n)) → arrTy As → arrTyVec m As)
nmap {m} n = λ⇒ (λ {As} → nvec-map {m} n {As})
```

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 → 10 + x) (10 :: 5 :: [])
```

evaluates to 11 :: 15 :: []. 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 (_,_) (1 :: 2 :: 3 :: []) (4 :: 5 :: 6 :: [])
```

evaluates to (1,4) :: (2,5) :: (3,6) :: []. Notice that, unlike the previous version, we did not need to explicitly specify the type of the data in the vectors.

4 Generic Haskell in Agda

In the previous section, we have seen how to embed simple generic functions in Agda, using both type and arity genericity. In this section, we will work through a more sophisticated example of type-generic programming by embedding a portion of Generic Haskell [8] in Agda.

The purpose of this embedding is twofold. First, it explores the foundations of Generic Haskell in a framework where it is easy to explore its variations. As we do not assume prior knowledge of Generic Haskell, this section also serves as an introduction to its foundations. Second, this embedding gives an example of dependently typed programming used for metaprogramming. It employs the techniques of typeful representations and tagless interpreters, and so demonstrates a powerful use of dependent types.

The initial embedding that we define differs in several ways from Generic Haskell. While Generic Haskell treats recursive types implicitly, we will make recursion explicit. Additionally, the embedding here lacks the full power of Generic Haskell because we only consider operations of arity one. We will rectify that in Section 5, when we extend this framework with arity genericity. In fact, that extension takes the notion of arity *further* than Generic Haskell, as it allows the definition of doubly generic operations.

4.1 Challenge problem

To motivate the initial embedding of Generic Haskell, we start with the following challenge problem. In Section 2.1 we developed a version of generic equality that works for all types composed of natural numbers, booleans and products. However, consider the type Choice, which is either an A, a B, both, or neither.

```
Choice : Set → Set → Set
Choice = λ A B → (A × B) ⊎ A ⊎ B ⊎ ⊤
```

Section 2.1's equality does *not* generalize the following functions, which are defined for lists and Choice.

```
eq-list : ∀ {A} → (A → A → Bool) → List A → List A → Bool
eq-list f  []        []        = true
eq-list f  (a :: as) (b :: bs) = f a b ∧ eq-list f as bs
eq-list f  _ _                 = false
eq-choice : ∀ {A B} → (A → A → Bool) → (B → B → Bool)
          → Choice A B → Choice A B → Bool
eq-choice fa fb (inj₁ (a1,b1))          (inj₁ (a2,b2))  = fa a1 a2 ∧ fb b1 b2
eq-choice fa fb (inj₂ (inj₁ a1))        (inj₂ (inj₁ a2))        = fa a1 a2
eq-choice fa fb (inj₂ (inj₂ (inj₁ b1))) (inj₂ (inj₂ (inj₁ b2))) = fb b1 b2
eq-choice fa fb _ _ = true
```

Note that these two functions apply to *parameterized datatypes*. List and Choice are type constructors (of type Set → Set and Set → Set → Set respectively) and their equality functions are polymorphic, abstracting the element type of the List, or options of the Choice. Therefore, these equality functions must also abstract over an equality function for the type parameters.

Furthermore, there are other generic operations that apply to multiple kinds of types. For example, size determines the size of a data structure and arb produces an arbitrary element of a (nonempty) type. The types of the size function at various instances is shown below.

```
size-nat    : ℕ → ℕ
size-bool   : Bool → ℕ
size-list   : ∀ {A} → (A → ℕ) → List A → ℕ
size-choice : ∀ {A B} → (A → ℕ) → (B → ℕ) → Choice A B → ℕ

arb-nat    : ℕ
arb-bool   : Bool
arb-list   : ∀ {A} → A → List A
arb-choice : ∀ {A B} → A → B → Choice A B
```

Our strategy for generalizing these operations is the same as it was before. We start with a code for types (using a datatype like Typ from before), give an interpretation of that code as an Agda type (using a function like ⌊_⌋ from before) and then define the generic operation by interpreting that code as an Agda function (like geq). However, this time, our definition of codes will include codes for multiple *kinds* of types.

4.2 A sublanguage of types

The types and type constructors that we will encode in this section are a sublanguage of Agda. This sublanguage includes a basic lambda calculus (variables, abstraction and application), basic type constants ($\top$, $\uplus$, and $\times$), and recursive definitions. This sublanguage is based on the type language of F_ω [10].

For example, we can define a type much like the standard Bool data type in this sublanguage. So that we can differentiate this type from the one in the standard library, we call it MyBool.

```
MyBool  : Set
MyBool  = ⊤ ⊎ ⊤

mytrue  : MyBool
mytrue  = inj₁ tt

myfalse : MyBool
myfalse = inj₂ tt
```

Likewise, type functions allow us to define a type similar to the standard, parameterized Maybe type. We call this one Option.

```
Option : Set → Set
Option = λ A → ⊤ ⊎ A

none   : ∀ {A} → Option A
none   = inj₁ tt

some   : ∀ {A} → A → Option A
some a = inj₂ a
```

Note that the definition of Choice, given above, already fits in this sublanguage.

For recursive types, such as natural numbers or lists, we must make recursion explicit in our type definitions. While we could define the structure of recursive types using recursion in Agda, such recursive type definitions can make the Agda type checker diverge.

Therefore, for explicit type recursion, we use the following definition. The type operator μ creates an explicit fixed point of a type function. The operations roll and unroll witness the isomorphism between the recursive type and its unrolling.

```
data μ : (Set → Set) → Set where
  roll : ∀ {A} → A (μ A) → μ A
unroll : ∀ {A} → μ A → A (μ A)
unroll (roll x) = x
```

Natural numbers are the fixed point of the function that takes a type to either unit (for zero) or the type again (for successor).

```
Nat   : Set
Nat   = μ (λ A → ⊤ ⊎ A)
zilch : Nat
zilch = roll (inj₁ tt)
succ  : Nat → Nat
succ x = roll (inj₂ x)
```

Likewise, lists can be defined using the fixed point type constructor.

```
MyList     : Set → Set
MyList A   = μ (λ B → ⊤ ⊎ (A × B))
nil        : ∀ {A} → MyList A
nil        = roll (inj₁ tt)
cons       : ∀ {A} → A → MyList A → MyList A
cons x xs  = roll (inj₂ (x,xs))
```

Finally, we can also represent a family of indexed types. In particular, a length-indexed vector can be defined as a $n-$tuple. Note that we do not use the μ constructor here—vectors are not represented as a recursive type, but rather MyVec is a family of types (one per natural number) defined by recursion in Agda.

```
MyVec             : Set → ℕ → Set
MyVec A 0         = ⊤
MyVec A (suc n)   = A × MyVec A n
vnil              : ∀ {A} → MyVec A 0
vnil          = tt
vcons         : ∀ {n} {A} → A → MyVec A n → MyVec A (suc n)
vcons x xs    = (x,xs)
```

Note that we are working with a simple operator for recursive types. Because μ has type (Set → Set) → Set, we can only work with regular, singly-recursive datatypes. Nested types would require a higher-typed fixed point operator. Mutually recursive datatypes cannot be modelled as fixed points of tuples. They must instead be translated to use single recursion.

The next step is to describe how we represent the structure of types as a code. Because our sublanguage of types is the simply typed lambda calculus (STLC) plus recursion and a few constants, we define a representation of STLC in Agda.

4.3 Representing the simply typed lambda calculus

To encode the sublanguage described in the previous subsection, we need datatypes for kinds, constants, and for the lambda calculus itself.

Kinds include the base kind $\star$ and function kinds. The function kind arrow associates to the right.

```
data Kind : Set where
  ⋆     : Kind
  _⇒_   : Kind → Kind → Kind
```

Constants are represented by a datatype that is indexed by the kind of the constant. They include unit, sum and product types.

```
data Const : Kind → Set where
  Unit  : Const ⋆
  Sum   : Const (⋆ ⇒ ⋆ ⇒ ⋆)
  Prod  : Const (⋆ ⇒ ⋆ ⇒ ⋆)
```

To represent other types (of arbitrary kinds), we now define an indexed datatype called Typ. A Typ may be a variable, an abstraction, an application, or a constant. The datatype is indexed by the kind of the type and a 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 nth Kind in the list is the kind of variable n.

```
data Ctx : Set where
  []    : Ctx
  _::_  : Kind → Ctx → Ctx
```

```
data V : Ctx → Kind → Set where
  VZ : ∀ {Γ k} → V (k :: Γ) k
  VS : ∀ {Γ k' k} → V Γ k → V (k' :: Γ) k

data Typ : Ctx → Kind → Set where
  Var  : ∀ {Γ k} → V Γ k → Typ Γ k
  Lam  : ∀ {Γ k₁ k₂} → Typ (k₁ :: Γ) k₂
         → Typ Γ (k₁ ⇒ k₂)
  App  : ∀ {Γ k₁ k₂} → Typ Γ (k₁ ⇒ k₂) → Typ Γ k₁
         → Typ Γ k₂
  Con  : ∀ {Γ k} → Const k → Typ Γ k
  Mu   : ∀ {Γ} → Typ Γ (⋆ ⇒ ⋆) → Typ Γ ⋆
```

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 kinds, constants, and type constructors, we need a mechanism to decode them as Agda types. A simple recursive function takes our encoding of kinds into an Agda kind.

```
⟦_⟧         : Kind → Set
⟦ ⋆ ⟧       = Set
⟦ a ⇒ b ⟧   = ⟦ a ⟧ → ⟦ b ⟧
```

Likewise, a simple function decodes constants. However, note that we need to know the kind of a constant to define the type of its interpretation.

```
C⟦_⟧ : ∀ {k} → Const k → ⟦ k ⟧
C⟦ Unit ⟧  = ⊤      -- has kind Set
C⟦ Sum ⟧   = _⊎_    -- has kind Set → Set → Set
C⟦ Prod ⟧  = _×_
```

To interpret type constructors, we must have an environment to interpret the variables. We index the datatype for the environment with the typing context to make sure that each variable is mapped to an Agda type of the right kind. We also define sLookup, which finds the type in an environment corresponding to a particular variable.

Note that the definition of Env overloads the [] and _::_ constructors, but Agda can again infer which we mean.

```
data Env : Ctx → Set where
  []    : Env []
  _::_  : ∀ {k G} → ⟦ k ⟧ → Env G → Env (k :: G)
sLookup : ∀ {k G} → V 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. Note that the interpretation of codes is a 'tagless' lambda-calculus interpreter. [3]

```
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       = C⟦ c ⟧
interp (Mu t) e        = μ (interp t e)
```

We use the ⌊_⌋ notation for decoding closed types in the empty environment.

```
⌊_⌋ : ∀ {k} → Ty k → ⟦ k ⟧
⌊ t ⌋ = interp t []
```

For example, recall the recursive type MyList.

```
MyList : Set → Set
MyList = λ A → μ (λ B → ⊤ ⊎ (A × B))
```

We can represent this type constructor with the following code:

```
list : Ty (⋆ ⇒ ⋆)
list =
  Lam (Mu (Lam
    (App (App (Con Sum) (Con Unit))
         (App (App (Con Prod) (Var (VS VZ))) (Var VZ)))))
```

The Agda type checker can normalize the type ⌊ list ⌋ to MyList, so these two types are equal.

As another example, we can represent the MyVec family of vector types by using a recursive function to calculate the length of the tuple.

```
myvec : ℕ → Ty (⋆ ⇒ ⋆)
myvec n = Lam (f n) where
  f : ℕ → Typ (⋆ :: []) ⋆
  f 0       = Con Unit
  f (suc n) = App (App (Con Prod) (Var VZ)) (f n)
```

[3] Compare this definition to Kiselyov's versions [16].

4.4 Type-generic operations

The last step is to define generic operations by "interpreting" codes as Agda functions. The crucial idea is that the type of the generic function depends on the kind of the code that it is given. To express this relationship we must use a *kind-indexed type* [12]:

```
_⟨_⟩_ : (Set → Set) → (k : Kind) → ⟦ k ⟧ → Set
b ⟨ ⋆ ⟩        t = b t
b ⟨ k1 ⇒ k2 ⟩ t = ∀ {A} → b ⟨ k1 ⟩ A → b ⟨ k2 ⟩ (t A)
```

In this definition, b is a type function that gives the type of the operation when the code represents a type at kind Set. For any b, the term b ⟨ k ⟩ t evaluates to the type of the corresponding generic operation for terms of type t (which has kind k). For example, we can describe the types of a generic equality function by using the following type function for b:

```
Eq : Set → Set
Eq A = A → A → Bool
```

With this b, the kind-indexed type can compute the types of the equality operation for various arguments. In each case, the given type normalizes to the same type that we declared Section 4.1 (indicated in comments).

```
eq-bool   : Eq ⟨ ⋆ ⟩ Bool
  -- Bool → Bool → Bool
eq-list   : Eq ⟨ ⋆ ⇒ ⋆ ⟩ MyList
  -- ∀ A → (A → A → Bool) → (MyList A → MyList A → Bool)
eq-choice : Eq ⟨ ⋆ ⇒ ⋆ ⇒ ⋆ ⟩ Choice
  -- ∀ A → (A → A → Bool) → ∀ B → (B → B → Bool)
  -- → (Choice A B → Choice A B → Bool)
```

A generic function is an interpretation of the Typ universe as an Agda term with a kind-indexed type. For example, the type of generic equality should be:

```
geq : ∀ {k} → (t : Ty k) → Eq ⟨ k ⟩ ⌊ t ⌋
```

We will define geq as an "interpreter" for the code t. For constants, this interpreter is not too difficult to define. For example, the type for product equality provides equality functions for the components of the product. Therefore the interpretation of this constant only needs to use these functions to check if corresponding components of the products are equal. Likewise, for disjoint unions, we must make sure that both arguments are the same injection, and then selectively use the provided functions.

```
geq-prod : ∀ {A} → (A → A → Bool) → ∀ {B} → (B → B → Bool)
           → (A × B) → (A × B) → Bool
geq-prod ra rb (x₁,x₂) (y₁,y₂) = ra x₁ y₁ ∧ rb x₂ y₂
geq-sum  : ∀ {A} → (A → A → Bool) → ∀ {B} → (B → B → Bool)
           → (A ⊎ B) → (A ⊎ B) → Bool
geq-sum ra rb (inj₁ x₁) (inj₁ x₂) = ra x₁ x₂
geq-sum ra rb (inj₂ x₁) (inj₂ x₂) = rb x₁ x₂
geq-sum _ _ _ _ = false
```

We put these together in a function that works for all constants.

```
geq-c : {k : Kind} → (c : Const k) → Eq ⟨ k ⟩ ⌊ Con c ⌋
geq-c Unit = λ t1 t2 → true
geq-c Sum  = geq-sum
geq-c Prod = geq-prod
```

For the full definition of generic equality, we must consider the complete collection of codes—roughly the simply typed lambda calculus with recursion. Because of λ, we must generalize this geq interpreter to codes for types with free variables. We will pass in an environment that we can use to interpret those free variables.

We define the environment as a list containing the interpretation of each variable in some context. It is indexed by the context for which it provides interpretations, and is parameterized by b so that it may be used for any generic operation.

```
data VarEnv (b : Set → Set) : Ctx → Set where
  []   : VarEnv b []
  _::_ : {k : Kind} {Γ : Ctx} {a : ⟦ k ⟧}
         → b ⟨ k ⟩ a → VarEnv b Γ → VarEnv b (k :: Γ)
```

With this definition of an environment, we also need a way to look up the interpretation of a variable. However, the type of this lookup function is problematic. How do we specify the return type? (See the ? marked in the type below.)

```
vLookup : ∀ {Γ k} {b : Set → Set} → (v : V Γ k) → (ve : VarEnv b Γ)
          → b ⟨ k ⟩ ?
vLookup VZ     (v :: ve) = v
vLookup (VS x) (v :: ve) = vLookup x ve
```

The return type we want is the one that appears in the provided VarEnv at the position corresponding to v. To get it, we use a new function called toEnv. It converts a VarEnv to an Env, so that we can use sLookup in the type of vLookup.

```
toEnv : {Γ : Ctx} {b : Set → Set} → VarEnv b Γ → Env Γ
toEnv [] = []
toEnv (_::_ {_} {_} {a} _ r) = a :: toEnv r
vLookup : ∀ {Γ k} {b : Set → Set} → (v : V Γ k) → (ve : VarEnv b Γ)
  → b ⟨ k ⟩ (sLookup v (toEnv ve))
vLookup VZ     (v :: ve) = v
vLookup (VS x) (v :: ve) = vLookup x ve
```

Finally, we can define the interpreter for lambda calculus terms. Note that variables are just looked up in the environment, lambda expressions map to functions, and application expressions map to applications. The recursion operator maps to the interpretation of its unrolling, and we use the function geq-c from above for the interpretation of constants.

```
geq-mu   : ∀ {A} → Eq (A (μ A)) → Eq (μ A)
geq-mu f = λ x y → f (unroll x) (unroll y)
geq-open : {Γ : Ctx} {k : Kind}
  → (ve : VarEnv Eq Γ)
  → (t : Typ Γ k) → Eq ⟨ k ⟩ (interp t (toEnv ve))
geq-open ve (Var v)      = vLookup v ve
geq-open ve (Lam t)      = λ y → geq-open (y :: ve) t
geq-open ve (App t1 t2)  = (geq-open ve t1) (geq-open ve t2)
geq-open ve (Mu t)       = geq-mu (geq-open ve (App t (Mu t)))
geq-open ve (Con c)      = geq-c c
```

We can define generic equality by providing the empty var environment

```
geq : {k : Kind} → (t : Typ Γ k) → Eq ⟨ k ⟩ ⌊ t ⌋
geq t = geq-open [] t
```

4.5 A General Framework

We have defined a very generic version of equality, but what about the next polykinded operation? Only the interpretation of constants and the rolling/unrolling in the Mu case changes with each generic function. Therefore, we can parametrize the above code to define a reusable framework. This reusable framework is a standard interpreter for the simply-typed lambda calculus. To use this interpreter it suffices to provide an interpretation of the type constants and implement the necessary rolling/unrolling in the Mu case.

The first step is a general type for the interpretation of constants. We use a first-class function for that interpretation. For generic equality, this function is exactly geq-c.

```
ConstEnv : (Set → Set) → Set
ConstEnv b = ∀ {k} → (c : Const k) → b ⟨ k ⟩ ⌊ Con c ⌋
```

The more difficult case is the treatment of Mu. We need to lift a generic definition for the unrolled type into a generic definition of the recursive type. This depends on the definition of b (for generic equality, we used geq-mu). To accommodate different operations, our generic framework accepts an argument that describes how to do this lifting.

```
MuGen : (Set → Set) → Set
MuGen b = ∀ {A} → b (A (μ A)) → b (μ A)
```

With these additional parameters, we can define our generic framework:

```
gen-open : {b : Set → Set} {Γ : Ctx} {k : Kind}
          → ConstEnv b → (ve : VarEnv b Γ) → MuGen b
          → (t : Typ Γ k) → b ⟨ k ⟩ (interp t (toEnv ve))
gen-open ce ve d (Var v)       = vLookup v ve
gen-open ce ve d (Lam t)       = λ y → gen-open ce (y :: ve) d t
gen-open ce ve d (App t1 t2) =
  (gen-open ce ve d t1) (gen-open ce ve d t2)
gen-open ce ve d (Con c)       = ce c
gen-open ce ve d (Mu t)        =
  d (gen-open ce ve d (App t (Mu t)))
```

Finally, we specialize gen-open to closed types.

```
gen : {b : Set → Set} {k : Kind} → ConstEnv b → MuGen b
    → (t : Ty k) → b ⟨ k ⟩ ⌊ t ⌋
gen c b t = gen-open c [] b t
```

This framework works for many generic operations. Observe that can use it to define equality as above.

```
geq : {k : Kind} → (t : Ty k) → Eq ⟨ k ⟩ ⌊ t ⌋
geq = gen geq-c geq-mu
```

Another example is a generic counting function, which returns 0 for unit and adds up the components of products and sums.

```
Count : Set → Set
Count A = A → ℕ
gcount : {k : Kind} → (t : Ty k) → Count ⟨ k ⟩ ⌊ t ⌋
gcount = gen gcount-c gcount-mu where
  gcount-c : ConstEnv Count
```

```
gcount-c Unit = λ t → 0
gcount-c Sum = gcount-sum where
  gcount-sum : ∀ {A} → _ → ∀ {B} → _ → (A ⊎ B) → ℕ
  gcount-sum ra rb (inj₁ x) = ra x
  gcount-sum ra rb (inj₂ x) = rb x
gcount-c Prod = gcount-prod where
  gcount-prod : ∀ {A} → _ → ∀ {B} → _ → (A × B) → ℕ
  gcount-prod ra rb (x₁,x₂) = ra x₁ + rb x₂
gcount-mu : MuGen Count
gcount-mu f = λ x → f (unroll x)
```

The Count example shows why it is important to make the type parameters explicit in the representation. This function can be instantiated to count the number of elements in an aggregate data structure

```
gsize : (t : Ty (⋆ ⇒ ⋆)) → ∀ {A} → ⌊ t ⌋ A → ℕ
gsize t = gcount t (λ x → 1)
```

and also sum them up if they all happen to be natural numbers.

```
gsum : (t : Ty (⋆ ⇒ ⋆)) → ⌊ t ⌋ ℕ → ℕ
gsum t = gcount t (λ x → x)
```

For example, for this list

```
exlist2 : MyList ℕ
exlist2 = cons 1 (cons 2 (cons 3 nil))
```

we have

```
gsize mylist exlist2 ≡ 3
gsum mylist exlist2 ≡ 6
```

and for this vector of numbers[4]

```
exvec2 : MyVec ℕ 3
exvec2 = vcons {2} 1 (vcons {1} 2 (vcons {0} 3 (vnil {ℕ})))
```

we already know its length, but we can calculate its sum in the same way.

```
gsum (myvec 3) exvec2 ≡ 6
```

[4] Note that Agda is unable to infer the implicit size parameter to vcons for the structural definition of vectors.

5 Arity-Generic Type-Generic Map

Unfortunately, the general framework presented in the last section is not expressive enough to give a type-generic version of map. Consider the various instances that we would like to generate:

$$\begin{array}{lll}
\mathsf{map\text{-}vec} & : & \forall\,\{\mathsf{A}_1\ \mathsf{A}_2\ \mathsf{n}\} \rightarrow (\mathsf{A}_1 \rightarrow \mathsf{A}_2) \rightarrow \mathsf{Vec}\ \mathsf{A}_1\ \mathsf{n}\ \ \rightarrow \mathsf{Vec}\ \mathsf{A}_2\ \mathsf{n} \\
\mathsf{map\text{-}maybe} & : & \forall\,\{\mathsf{A}_1\ \mathsf{A}_2\}\ \ \ \rightarrow (\mathsf{A}_1 \rightarrow \mathsf{A}_2) \rightarrow \mathsf{Maybe}\ \mathsf{A}_1 \rightarrow \mathsf{Maybe}\ \mathsf{A}_2 \\
\mathsf{map\text{-}choice} & : & \forall\,\{\mathsf{A}_1\ \mathsf{A}_2\ \mathsf{B}_1\ \mathsf{B}_2\} \rightarrow (\mathsf{A}_1 \rightarrow \mathsf{A}_2) \rightarrow (\mathsf{B}_1 \rightarrow \mathsf{B}_2) \\
 & \multicolumn{2}{l}{\rightarrow \mathsf{Choice}\ \mathsf{A}_1\ \mathsf{B}_1 \rightarrow \mathsf{Choice}\ \mathsf{A}_2\ \mathsf{B}_2}
\end{array}$$

Thus, we want the polykinded type to give us something like this:

$$\begin{array}{l}
\mathsf{Map}\ \langle\, \star \,\rangle\qquad \mathsf{T}\ =\ \mathsf{T} \rightarrow \mathsf{T} \\
\mathsf{Map}\ \langle\, \star \Rightarrow \star \,\rangle\ \mathsf{T}\ =\ \forall\,\{\mathsf{A}\ \mathsf{B}\} \rightarrow (\mathsf{A} \rightarrow \mathsf{B}) \rightarrow (\mathsf{T}\ \mathsf{A} \rightarrow \mathsf{T}\ \mathsf{B}) \\
\mathsf{Map}\ \langle\, \star \Rightarrow \star \Rightarrow \star \,\rangle\ \mathsf{T}\ =\ \forall\,\{\mathsf{A}_1\ \mathsf{B}_1\ \mathsf{A}_2\ \mathsf{B}_2\} \\
\quad \rightarrow (\mathsf{A}_1 \rightarrow \mathsf{B}_1) \rightarrow (\mathsf{A}_2 \rightarrow \mathsf{B}_2) \rightarrow (\mathsf{T}\ \mathsf{A}_1\ \mathsf{A}_2 \rightarrow \mathsf{T}\ \mathsf{B}_1\ \mathsf{B}_2)
\end{array}$$

But there is no definition of Map that has this behavior because each case takes too many type arguments. One way to solve this problem would be to start all over again and define an "arity-2" kind-indexed type:

$$\begin{array}{l}
\_\langle\_\rangle_2\ :\ (\mathsf{Set} \rightarrow \mathsf{Set} \rightarrow \mathsf{Set}) \rightarrow (\mathsf{k}\ :\ \mathsf{Kind}) \rightarrow [\![\, \mathsf{k} \,]\!] \rightarrow [\![\, \mathsf{k} \,]\!] \rightarrow \mathsf{Set} \\
\mathsf{b}\ \langle\, \star \,\rangle_2\ =\ \lambda\, \mathsf{t}_1\ \mathsf{t}_2 \rightarrow \mathsf{b}\ \mathsf{t}_1\ \mathsf{t}_2 \\
\mathsf{b}\ \langle\, \mathsf{k}_1 \Rightarrow \mathsf{k}_2 \,\rangle_2\ =\ \lambda\, \mathsf{t}_1\ \mathsf{t}_2 \rightarrow \forall\,\{\mathsf{a}_1\ \mathsf{a}_2\} \\
\qquad\qquad\quad \rightarrow (\mathsf{b}\ \langle\, \mathsf{k}_1 \,\rangle_2)\ \mathsf{a}_1\ \mathsf{a}_2 \rightarrow (\mathsf{b}\ \langle\, \mathsf{k}_2 \,\rangle_2)\ (\mathsf{t}_1\ \mathsf{a}_1)\ (\mathsf{t}_2\ \mathsf{a}_2)
\end{array}$$

In that case, the simple definition

$$\begin{array}{l}
\mathsf{Map}\ :\ \mathsf{Set} \rightarrow \mathsf{Set} \rightarrow \mathsf{Set} \\
\mathsf{Map}\ \mathsf{A}\ \mathsf{B}\ =\ \mathsf{A} \rightarrow \mathsf{B}
\end{array}$$

exactly specifies the types of map that we would like above.

$$\mathsf{gmap}\ :\ \forall\,\{\mathsf{k}\} \rightarrow (\mathsf{t}\ :\ \mathsf{Ty}\ \mathsf{k}) \rightarrow \mathsf{Map}\ \langle\, \mathsf{k} \,\rangle_2\ \lfloor\, \mathsf{t} \,\rfloor\ \lfloor\, \mathsf{t} \,\rfloor$$

However, this approach would require redefining our entire framework for arity-2 functions (we would need $\mathsf{ConstEnv}_2$, VarEnv_2, $\mathsf{gen\text{-}open}_2$, gen_2, etc.). That is not very generic!

5.1 Generic programming at multiple arities

Instead, we would like a single framework for all arities of generic functions. We can get that single framework by making the first argument to the polykinded type, b, take a *vector* of arguments instead of just one or two.

If we make that change, then the kind-indexed type is defined as follows:

```
_⟨_⟩_ : ∀ {n : ℕ} → (Vec Set n → Set) → (k : Kind)
      → Vec ⟦ k ⟧ n → Set
b ⟨ ⋆ ⟩        v = b v
b ⟨ k₁ ⇒ k₂ ⟩ v = {a : Vec ⟦ k₁ ⟧ _} → b ⟨ k₁ ⟩ a → b ⟨ k₂ ⟩ (v ⊛ a)
```

Recall that v ⊛ a applies vector of functions to vector of arguments pointwise.

If we extend ConstEnv and MuGen in a similar way we can define a generic framework that supports multiple arities. We give the signature of that function, called ngen below. For simplicity, we defer the details of its implementation to the Appendix. This operation produces a value of a kind-indexed type given a mapping from constants to appropriate definitions.

```
  -- interpretation of constants
ConstEnv   : {n : ℕ} → (b : Vec Set (suc n) → Set) → Set
ConstEnv b = {k : Kind} (c : Const k) → b ⟨ k ⟩ repeat ⌊ Con c ⌋
  -- folding function for recursive types
MuGen : (n : ℕ) → (Vec Set (suc n) → Set) → Set
MuGen n b = ∀ {A} → b (A ⊛ (repeat μ ⊛ A)) → b (repeat μ ⊛ A)
  -- type-generic framework for multiple arities
ngen : ∀ {n : ℕ} {b : Vec Set n → Set} {k : Kind}
    → (t : Ty k) → ConstEnv b → MuGen n b → b ⟨ k ⟩ (repeat ⌊ t ⌋)
```

Recall that repeat ⌊ t ⌋ returns a vector with n copies of ⌊ t ⌋, where the length of the vector is automatically determined by the context.

With ngen, we can define several different type-generic mapping operations (at different arities). For example, a generic repeat operation is the arity-one version of map. This program generalizes repeat (shown for vectors in Section 2) to all types in our universe.

```
Repeat : Vec Set 1 → Set
Repeat (A :: []) = A
grepeat : {k : Kind} → (t : Ty k) → Repeat ⟨ k ⟩ (⌊ t ⌋ :: [])
grepeat t = ngen t grepeat-c (λ {As} → grepeat-mu {As}) where
  grepeat-c : ConstEnv Repeat
  grepeat-c Unit = tt
  grepeat-c Sum = λ {A} → grepeat-sum {A} where
    grepeat-sum : Repeat ⟨ ⋆ ⇒ ⋆ ⇒ ⋆ ⟩ (_⊎_ :: [])
    grepeat-sum {A :: []} ra {B :: []} rb = inj₂ rb
  grepeat-c Prod = λ {A} → grepeat-prod {A} where
    grepeat-prod : Repeat ⟨ ⋆ ⇒ ⋆ ⇒ ⋆ ⟩ (_×_ :: [])
    grepeat-prod {A :: []} ra {B :: []} rb = (ra,rb)
  grepeat-mu : ∀ {As} → Repeat (As ⊛ ((μ :: []) ⊛ As)) → Repeat ((μ :: []) ⊛ As)
  grepeat-mu {A :: []} = roll
```

Note that in the case for sums, grepeat has a choice, it can either choose the first or the second injection. We arbitrarily put the second injection above because that is the one that generates the familiar repeat for lists—by always choosing inj_2 we generate a list of infinite length. Note that grepeat list expects a vector of types (of length 1) as its first argument. We create that vector below, relying on type inference to automatically fill in the implicit argument A.

```
repeat-list : ∀ {A} → A → MyList A
repeat-list = grepeat list {_ :: []}
```

Likewise, the type-generic mapping operation has arity two. This function depends on map-sum and map-prod, which define mapping over sums and products.

```
Map : Vec Set 2 → Set
Map (A :: B :: []) = A → B
map-sum : Map ⟨ ⋆ ⇒ ⋆ ⇒ ⋆ ⟩ (_⊎_ :: _⊎_ :: [])
map-sum {A1 :: B1 :: []} ra {A2 :: B2 :: []} rb = g where
  g : (A1 ⊎ A2) → B1 ⊎ B2
  g (inj₁ x) = inj₁ (ra x)
  g (inj₂ x) = inj₂ (rb x)
map-prod : Map ⟨ ⋆ ⇒ ⋆ ⇒ ⋆ ⟩ (_×_ :: _×_ :: [])
map-prod {A1 :: B1 :: []} ra {A2 :: B2 :: []} rb = g where
  g : (A1 × A2) → B1 × B2
  g (x,y) = (ra x,rb y)
gmap-mu : ∀ {As} → Map (As ⊛ ((μ :: μ :: []) ⊛ As)) → Map ((μ :: μ :: []) ⊛ As)
gmap-mu {_ :: _ :: []} = λ x y → roll (x (unroll y))
gmap : ∀ {k : Kind} → (t : Ty k) → Map ⟨ k ⟩ (⌊ t ⌋ :: ⌊ t ⌋ :: [])
gmap t = ngen t gmap-c gmap-mu where
  gmap-c : ConstEnv Map
  gmap-c Unit = λ x → x
  gmap-c Sum = map-sum
  gmap-c Prod = map-prod
```

Finally, type-generic zipWith has arity three. It again depends on zipping operations for sums and products. In the case of zipping for sums, we must be partial. If the two arguments are not the same case of the sum, then they cannot be zipped together. Because Agda lacks Haskell's error function, we use a postulate that will halt the program if it is ever encountered. This partiality means that the generic zipWith that we define here differs from the zipWith defined for lists in the Haskell prelude. When given lists of unequal length, this function will fail, whereas the prelude function will ignore the extra elements in the longer list. As a result, we cannot use this zipWith to show that every parameterized type is an applicative functor.

```
postulate error : (A : Set) → A

ZW : Vec Set 3 → Set
ZW (A :: B :: C :: []) = A → B → C

zip-sum : ZW ⟨ ⋆ ⇒ ⋆ ⇒ ⋆ ⟩ (_⊎_ :: _⊎_ :: _⊎_ :: [])
zip-sum {A1 :: A2 :: A3 :: []} ra {B1 :: B2 :: B3 :: []} rb = g where
  g : (A1 ⊎ B1) → (A2 ⊎ B2) → A3 ⊎ B3
  g (inj₁ x) (inj₁ y) = inj₁ (ra x y)
  g (inj₂ x) (inj₂ y) = inj₂ (rb x y)
  g _ _               = error

zip-prod : ZW ⟨ ⋆ ⇒ ⋆ ⇒ ⋆ ⟩ (_×_ :: _×_ :: _×_ :: [])
zip-prod {A1 :: A2 :: A3 :: []} ra {B1 :: B2 :: B3 :: []} rb = g where
  g : (A1 × B1) → (A2 × B2) → A3 × B3
  g (x,y) (w,z) = (ra x w,rb y z)

gzipWith : ∀ {k} → (t : Ty k) → ZW ⟨ k ⟩ (⌊ t ⌋ :: ⌊ t ⌋ :: ⌊ t ⌋ :: [])
gzipWith t = ngen t gzip-c gzip-mu where
  gzip-c : ConstEnv ZW
  gzip-c Unit = λ x y → x
  gzip-c Sum = zip-sum
  gzip-c Prod = zip-prod

  gzip-mu : ∀ {As} → ZW (As ⊛ ((μ :: μ :: μ :: []) ⊛ As))
    → ZW ((μ :: μ :: μ :: []) ⊛ As)
  gzip-mu {_ :: _ :: _ :: []} = λ x y z → roll (x (unroll y) (unroll z))
```

Because of the partiality in this definition, the definition of gzipWith is not exactly the same as the one for lists in Haskell's standard library. There, when given two lists of different lengths the function truncates the zip. Here, zipWith is defined *only* for lists of the same length. Even if we redefined the above to make the partiality explicit, by returning a Maybe instead of using error, it would not produce the same behavior as Haskell's library function.

5.2 Doubly Generic map

The last challenge is to combine grepeat, gmap, and gzipWith into a single, doubly generic operation, using ngen. To define this operation, we must first define b, ConstEnv and MuGen arguments that make sense at any arity. For doubly generic map, we call these pieces NGmap, ngmap-const and ngmap-mu.

NGmap is similar to the arrTy function from Section 3.1, which takes the arity as an implicit argument.

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

For ngmap-const, we assemble the const environment out of specific cases (to be defined below):

```
ngmap-const : {n : ℕ} → ConstEnv {n} NGmap
ngmap-const {n} Unit  =  defUnit n
ngmap-const {n} Prod  =  defPair n
ngmap-const {n} Sum   =  defSum n
```

For the unit case, we return an arity-n function with type $\top \to \top \to \ldots \to \top$.

```
defUnit : (n : ℕ) → NGmap {n} ⟨ ⋆ ⟩ (repeat ⊤)
defUnit zero      =  tt
defUnit (suc n)  =  λ x → (defUnit n)
```

Because the Prod and Sum constants have higher kinds, the return type of ngmap-const changes in these cases. Consider Prod first.

```
defPair : (n : ℕ)
          → {As : Vec Set (suc n)} → NGmap As
          → {Bs : Vec Set (suc n)} → NGmap Bs
          → NGmap (repeat _×_ ⊛ As ⊛ Bs)
defPair zero     {A :: []}           a {B :: []}           b  =  (a,b)
defPair (suc n) {A1 :: A2 :: As} a {B1 :: B2 :: Bs} b  =
          λ p → defPair n {A2 :: As} (a (proj₁ p)) {B2 :: Bs} (b (proj₂ p))
```

In the zero case of defPair, a and b are arguments of type A and B respectively—the function simply pairs them up. In the successor case, a and b are functions with types A1 → NGmap As and B1 → NGmap Bs. We want to produce a result of type A1 × B1 → NGmap (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.

In the case of Sum, we must check that the terms provided have the same structure (are either all inj_1 or all inj_2). If the supplied sums are not all constructed with the same injections, there will not be enough arguments to apply a or b. One possibility is to check the structure first and fail immediately if we see mixed inj_1s and inj_2s, but we prefer a lazy approach. Below, we recursively accumulate the results of a and b, but use the error term to fill in the missing arguments. When all the injections agree, a or b will build up the correct result. When they do not, the error is triggered.

```
defSum : (n : ℕ)
    → {As : Vec Set (suc n)} → NGmap As
    → {Bs : Vec Set (suc n)} → NGmap Bs
    → NGmap (repeat _⊎_ ⊛ As ⊛ Bs)
defSum zero     {A :: []}             a {B :: []}             b  =  (inj₂ b)
defSum (suc 0) {A1 :: (A2 :: [])}  a {B1 :: (B2 :: [])} b  =  f
  where
    f : A1 ⊎ B1 → A2 ⊎ B2
    f (inj₁ a1)  =  inj₁ (a a1)
    f (inj₂ b1)  =  inj₂ (b b1)
defSum (suc n) {A1 :: (A2 :: As)} a {B1 :: (B2 :: Bs)} b  =  f
  where
    f : A1 ⊎ B1 → NGmap (repeat _⊎_ ⊛ (A2 :: As) ⊛ (B2 :: Bs))
    f (inj₁ a1)  =  defSum n {A2 :: As} (a a1)    {B2 :: Bs} (b error)
    f (inj₂ b1)  =  defSum n {A2 :: As} (a error) {B2 :: Bs} (b b1)
```

Note that the type of arity zero map for sums is A → B → A ⊎ B, and we arbitrarily pick the second injection.

Lastly, we specify the behavior of map for recursive types. This function essentially unrolls each argument, applies f, and then rolls up the result.

```
MuGen      : (n : ℕ) → (Vec Set (suc n) → Set) → Set
MuGen n b  = ∀ {As} → b (As ⊛ (repeat μ ⊛ As)) → b (repeat μ ⊛ As)

ngmap-mu : ∀ {n} → MuGen n NGmap
ngmap-mu {zero}  {A :: []}           =  roll
ngmap-mu {suc n} {A1 :: A2 :: As}  =  λ f x →
  ngmap-mu {n} {A2 :: As} (f (unroll x))
```

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 (λ {As} → ngmap-mu {n} {As})
```

This definition is truly *doubly generic*. We may instantiate it to derive map at any arity and any type in our universe. For example, in the case of lists, we have the following definitions. Note that repeat is ngmap 0, map is ngmap 1 and zipWith is ngmap 2.

```
repeat-ml   : ∀ {B} → B → MyList B
repeat-ml   =  ngmap 0 list {_ :: []}
map-ml      : ∀ {A₁ B} → (A₁ → B) → MyList A₁ → MyList B
map-ml      =  ngmap 1 list {_ :: _ :: []}
zipWith-ml  : ∀ {A₁ A₂ B} → (A₁ → A₂ → B)
             → MyList A₁ → MyList A₂ → MyList B
zipWith-ml  =  ngmap 2 list {_ :: _ :: _ :: []}
```

6 Other Doubly Generic Operations

Map is not the only arity-generic function. In this section, we examine two others and discuss their implementations.

6.1 Equality

We saw in the previous section that doubly generic map must check that its arguments have the same structure. We can define doubly generic equality in a similar manner. This function takes n arguments, returning true if they are all equal, and false otherwise. Unlike map, equality is not partial for sums as it returns false in the case that the injections do not match.

In the specific case of vectors, arity-generic equality looks a lot like arity-generic map. Each instance of this function follows the same pattern. Given an n-ary equality function for the type argument, we can define n-ary equality for vectors as:

```
nvec-eq : {m : ℕ} {A1 : Set} ... {An : Set} → (A1 → ... → An → Bool)
        → Vec A1 m → ... → Vec An m → Bool
nvec-eq f v1 ... vn = all (repeat f ⊛ v1 ⊛ ... ⊛ vn)
```

However, again this definition does not help us make equality type-generic as well as arity-generic. For type genericity, the type of the equality function depends on the kind of the type constructor.

For example, the definition of arity-three equality for natural numbers returns true only if all three match:

```
nat-eq3 : ℕ → ℕ → ℕ → Bool
```

Likewise, the arity-three equality for pairs requires equalities for all of the components of the pair. Furthermore, the type arguments need not be the same. We can pass any sort of comparison functions in to examine the values carried by the three products.

```
pair-eq3 : {A1 B1 C1 A2 B2 C2 : Set}
         → (A1 → B1 → C1 → Bool) → (A2 → B2 → C2 → Bool)
         → (A1 × A2) → (B1 × B2) → (C1 × C2) → Bool
pair-eq3 f g (a1,a2) (b1,b2) (c1,c2) = f a1 b1 c1 ∧ g a2 b2 c2
```

For sums, we also may pass in two different comparison functions to examine the values carried by the three sums. However, those three sums must all match in order to use the comparison functions.

```
sum-eq3 : {A1 B1 C1 A2 B2 C2 : Set}
        → (A1 → B1 → C1 → Bool) → (A2 → B2 → C2 → Bool)
        → (A1 ⊎ A2) → (B1 ⊎ B2) → (C1 ⊎ C2) → Bool
```

```
sum-eq3 f g (inj₁ a1) (inj₁ b1) (inj₁ c1)  =  f a1 b1 c1
sum-eq3 f g (inj₂ a2) (inj₂ b2) (inj₂ c2)  =  g a2 b2 c2
sum-eq3 f g _ _ _  =  false
```

The definition of ngeq, which can define all of these operations, is similar to that of ngmap, so we will only highlight the differences. 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 arity zero. In that case its type is simply Bool.

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

Next we must define a ConstEnv 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 ⊤ 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 products, NGeq must project the corresponding components of each of the tuples and pass them to the two n-ary comparison functions.

```
defPair : (n : ℕ) →
   {as : Vec Set (suc n)} → (NGeq as) →
   {bs : Vec Set (suc n)} → (NGeq bs) →
   NGeq ((repeat _×_ ⊛ as) ⊛ bs)
defPair zero     {a :: []}          at {b :: []}          bt  =
   λ x → at (proj₁ x) ∧ bt (proj₂ x)
defPair (suc n) {a1 :: a2 :: as} af {b1 :: b2 :: bs} bf  =
   λ x → (defPair n {a2 :: as} (af (proj₁ x))
                        {b2 :: bs} (bf (proj₂ x)))
```

The case for sums is similar in structure. The important part of this case is that after the first argument as been discriminated, all remaining arguments must match it. So this branch dispatches to two helper functions that require all of the remaining arguments to be either first or second injections, return false if the a mismatched argument is supplied.

```
   -- the n-ary constant false function
constFalse : {n : ℕ} → (v : Vec Set n) → NGeq v
constFalse {zero} [] = false
constFalse {suc m} (A1 :: As) = λ a → constFalse As
defSumFirst : (n : ℕ) →
    {as : Vec Set (suc n)} → (NGeq as) →
    {bs : Vec Set (suc n)} →
    NGeq (repeat _⊎_ ⊛ as ⊛ bs)
defSumFirst zero {a :: []} at {b :: []} = f
  where f : a ⊎ b → Bool
    f (inj₁ x1) = at x1
    f (inj₂ x1) = false
defSumFirst (suc n) {a1 :: a2 :: as} af {b1 :: b2 :: bs} = f
  where f : a1 ⊎ b1 → NGeq (repeat _⊎_ ⊛ (a2 :: as) ⊛ (b2 :: bs))
    f (inj₁ x1) = defSumFirst n (af x1)
    f (inj₂ x1) = constFalse (repeat _⊎_ ⊛ (a2 :: as) ⊛ (b2 :: bs))
```

```
defSumSecond : (n : ℕ) →
    {as : Vec Set (suc n)} →
    {bs : Vec Set (suc n)} → (NGeq bs) →
    NGeq (repeat _⊎_ ⊛ as ⊛ bs)
defSumSecond zero {a :: []} {b :: []} bt = f
  where f : a ⊎ b → Bool
    f (inj₁ x1) = false
    f (inj₂ x1) = bt x1
defSumSecond (suc n) {a1 :: a2 :: as} {b1 :: b2 :: bs} bf = f
  where f : a1 ⊎ b1 → NGeq (repeat _⊎_ ⊛ (a2 :: as) ⊛ (b2 :: bs))
    f (inj₁ x1) = constFalse (repeat _⊎_ ⊛ (a2 :: as) ⊛ (b2 :: bs))
    f (inj₂ x1) = defSumSecond n (bf x1)
```

```
defSum : (n : ℕ) →
    {as : Vec Set (suc n)} → (NGeq as) →
    {bs : Vec Set (suc n)} → (NGeq bs) →
    NGeq (repeat _⊎_ ⊛ as ⊛ bs)
defSum zero {a :: []} at {b :: []} bt = f
  where f : a ⊎ b → Bool
    f (inj₁ x1) = at x1
    f (inj₂ x1) = bt x1
defSum (suc n) {a1 :: a2 :: as} af {b1 :: b2 :: bs} bf = f
  where f : a1 ⊎ b1 → NGeq (repeat _⊎_ ⊛ (a2 :: as) ⊛ (b2 :: bs))
    f (inj₁ x1) = defSumFirst n (af x1)
    f (inj₂ x1) = defSumSecond n (bf x1)
```

```
ngeq-const : {n : ℕ} → ConstEnv {n} NGeq
ngeq-const {n} Unit = defUnit n
ngeq-const {n} Prod = defPair n
ngeq-const {n} Sum = defSum n
```

Finally, because we wish to use ngeq for recursive data structures, we must define an instance of MuGen. 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, unroll it to coerce it to the appropriate type, and recurse.

```
ngeq-mu : ∀ {n} → MuGen n NGeq
ngeq-mu {zero}  {A :: []}         = λ g x → g (unroll x)
ngeq-mu {suc n} {A1 :: A2 :: As}  = λ g x → ngeq-mu (g (unroll x))
```

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 ⌊ e ⌋)
ngeq n e = ngen e ngeq-const (λ {As} → ngeq-mu {n} {As})
```

6.2 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 that unifies all of these definitions.

Furthermore, it makes sense that we should be able to unzip data structures other than lists, such as Options or Choices.

```
unzipOption :: Option (a,b) → (Option a,Option b)
unzipTree   :: Choice (a1,a2) (b1,b2) → (Choice a1 b1,Choice a2 b2)
```

Indeed, unzip is also datatype-generic, and Generic Haskell includes the function gunzipWith that can generate unzips for any type (of any kind). The generic function gunzipWith is a little more general than unzip above, as the data structure need not contain pairs. For example, in the instance for Options, it requires an additional function to describe how to divide the optional value into two pieces.

```
gunzipWith {|Option|} : {A B C : Set} → (A → B × C)
                         → (Option A → Option B × Option C)
```

By supplying the identity function, we can derive unzipOption above.

```
unzipOption : {A B : Set} → Option (A × B) → (Option A × Option B)
unzipOption = gunzipWith {|Option|} (λ x → x)
```

Here, we describe the definition of ngsplit, which generates unzipWith for arbitrary data structures at arbitrary arities. In some sense, ngsplit is the inverse 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.

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. 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 case Unit is straightforward, so we do not show it. It simply makes n copies of the argument.

To split a product (x,y), we first split x and y, then combine together the results. That is, we need an arity-generic function to take in arguments of types (A1 × A2 × ... × An) and (B1 × B2 × ... × Bn) and produce a result of type:

```
(A1 × B1) × (A2 × B2) × ... × (An × Bn)
```

We call this helper function prodn

```
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)
```

and use it to define the case for products.

```
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 first or second injection, and uses the appropriate provided function to split the inner expression. Then we use either injFirst or injSecond (defined below), which simply map inj_1 or inj_2 onto the members of the resulting tuple.

```
injFirst : {n : ℕ} {As Bs : Vec Set n}
  → prodTy As
  → prodTy (repeat _⊎_ ⊛ As ⊛ Bs)
injFirst {0} {[]}     {[]}     tt = tt
injFirst {1} {A :: []} {B :: []} a = inj₁ a
injFirst {suc (suc n)} {A :: As} {B :: Bs} (a,as) =
  (inj₁ a,injFirst {suc n} as)
injSecond : {n : ℕ} {As Bs : Vec Set n}
        → prodTy Bs
        → prodTy (repeat _⊎_ ⊛ As ⊛ Bs)
injSecond {0} {[]}     {[]}     tt = tt
injSecond {1} {A :: []} {B :: []} b = inj₂ b
injSecond {suc (suc n)} {A :: As} {B :: Bs} (b,bs) =
  (inj₂ b,injSecond {suc n} bs)
```

```
defSum : (n : ℕ)
        → {As : Vec Set (suc n)} → (NGsplit As)
        → {Bs : Vec Set (suc n)} → (NGsplit Bs)
        → NGsplit (repeat _⊎_ ⊛ As ⊛ Bs)
defSum n {A :: As} af {B :: Bs} bf = f
  where f : A ⊎ B → prodTy (repeat _⊎_ ⊛ As ⊛ Bs)
        f (inj₁ x1) = injFirst  {n} (af x1)
        f (inj₂ x1) = injSecond {n} (bf x1)
```

As before, the definition of split-const dispatches to the branches above in the standard way.

```
split-const : {n : ℕ} → ConstEnv {n} NGsplit
split-const {n} Unit = defUnit n
```

```
split-const Prod = defPair _
split-const Sum = defSum _
```

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 functions are similar to previous examples, except that we are converting a pair instead of an arrow.

```
  -- roll all of the components of a product
roll-all : ∀ {n : ℕ} {As : Vec (Set → Set) n} →
  prodTy (As ⊛ (repeat μ ⊛ As)) →
  prodTy (repeat μ ⊛ As)
roll-all {0} {[]} tt = tt
roll-all {1} {A :: []} x = roll x
roll-all {suc (suc n)} {A1 :: A2 :: As} (x,xs) = (roll x,roll-all {suc n} xs)
split-mu : {n : ℕ} → MuGen n NGsplit
split-mu {0}           {A :: []}        = λ g → λ x → g (unroll x)
split-mu {1}           {A1 :: A2 :: []} = λ g → λ x → roll (g (unroll x))
split-mu {suc (suc n)} {A1 :: A2 :: As} =
  λ g → λ x → roll-all {suc (suc n)} {A2 :: As} (g (unroll x))
```

With split-const, we can define ngsplit as usual.

```
ngsplit : (n : ℕ) → {k : Kind} → (e : Ty k)
      → NGsplit {n} ⟨ k ⟩ (repeat ⌊ e ⌋)
ngsplit n e = ngen e split-const split-mu
```

Splitting is a good example of generic programming's potential to save time and eliminate errors. Defining a separate instance of split for vectors is tricky. For example, we would need a function 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 with generic programming we get the vector split for free. Moreover, we may reason once about the correctness of the general definition of split rather than reasoning individually about each of its arity and type instances.

6.3 More 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. Though the known collection of doubly 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.

7 Related Work

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 [33] and GADTs [34] and showed how to implement a number of datatype-generic operations such as map. Hinze [13], 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. [30, 32]. They use the Coq programming language to define a framework for generic programming, but do not consider arity-genericity. Altenkirch and McBride [1] show a similar development in Oleg. Though these authors do not consider arity-genericity, their frameworks should easily support it thanks to their dependently typed settings.

The idea of generic programming in dependent type theory via universes has seen much attention since it was originally proposed [17, 21, 14, 6]. This tutorial 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. [3] and Morris et al. [20] for more expressive universes). A dependently typed language also permits the definition of generic proofs about generic programs. Chlipala [7] uses this technique in the Coq proof assistant to generically define and prove substitution properties of programming languages. Verbruggen et al. [31, 32] use Coq's dependent types to develop a framework for proving properties about generic programs.

At a more theoretical level, Hoogendijk and Backhouse [15] have provided a foundation for polytypic programming in the theory of allegories. They consider the operations that result from "commuting" any two datatypes, and derive generic zip operations as the special case when pairs are chosen as one of the types. They observe that this framework may be used with datatypes of various arities to construct different operations, but do not consider arity-genericity itself.

Only a few sources discuss arity-generic programming. Fridlender and Indrika [9] 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 [18] 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.

Many Scheme functions, such as map, are arity-generic (or variable-arity, in Scheme parlance). Strickland et al. [28] extend Typed Scheme with support for variable-arity polymorphism by adding new forms for variable-arity functions to the type language. They are able to check many examples, but do not consider datatype genericity.

Sheard [26] translates Fridlender and Indrika's example to the Ωmega programming language, using that language's native indexed datatypes instead of the Church encoding. He also demonstrates one other arity-generic program, n-ary addition. Although the same work also includes an implementation of datatype-generic programming in Ωmega, the two ideas are not combined.

8 Discussion

Termination checking. Because we wanted to model generic programming in Haskell, we need recursive datatypes and recursive functions. Such definitions run afoul of Agda's termination checker, so we have disabled it using the flags `-no-termination-check` and `-no-positivity-check`. These flags make Agda behave like the Cayenne programming language [2].

Dependently-typed programming languages that do not guarantee termination are unsound when viewed as logics. Indeed, looping terms inhabit every type, so every proposition is provable. However, such languages still satisfy the property of type soundness, and ensure that programs do not crash. This weakens the reasoning that can be done in such languages, because a "proof" might diverge, but does not negate the benefits of dependency in the type system.

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 Ωmega 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. Though some recent work has shown how to add Agda-style pattern matching to Coq, this is still only available as an experimental language extension [27]. 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 lead 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 type check the 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, etc.

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[5], but in previous work [36] we have shown that we are not using it in an unsound way by implementing the `ngen` function and several arity-generic functions without it.

Three differences between Coq and Agda make this explicit version more complicated than the one presented here. First, Coq supports *universe polymorphism* [11], a feature which allows definitions to work on multiple type levels. Recent versions of Agda support a new form of this feature, but they require each universe-polymorphic function to explicitly quantify over and manipulate universe levels. This substantially clutters the definitions. 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, it would be necessary to explicitly coerce types back and forth between Set0 and Set1.

Conclusions. This tutorial served several purposes. It introduced type genericity and arity genericity, and showed how they can be combined in powerful, doubly generic operations. Just as importantly, it showed how a rich framework for these operations can be defined within dependently typed programming languages. We believe generic programming can be a *killer app* for dependently typed programming. Languages like Agda provide a nearly perfect environment for investigating more generic operations and reasoning about them.

[5] But note, type-in-type does not make a dependent type system unsound [4]

Acknowledgments Thanks to Andres Löh and Tim Sheard for discussion. This paper was generated from literate Agda sources using `lhs2TeX`. Thanks to Jeremy Gibbons and Ralf Hinze for comments on the draft.

References

[1] T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *Proceedings of the IFIP TC2 Working Conference on Generic Programming*, Dagstuhl, Germany, July 2003.

[2] L. Augustsson. Cayenne—a language with dependent types. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ICFP '98, pages 239–250, New York, NY, USA, 1998. ACM. ISBN 1-58113-024-4. doi: 10.1145/289423.289451. URL `http://doi.acm.org/10.1145/289423.289451`.

[3] M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4):265–289, 2003.

[4] L. Cardelli. A polymorphic lambda calculus with type:type. Technical Report 10, 1986.

[5] M. M. T. Chakravarty, G. Keller, and S. 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.

[6] J. Chapman, P.-E. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *ICFP '10*, pages 3–14, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: http://doi.acm.org/10.1145/1863543.1863547.

[7] A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 54–65, New York, NY, USA, 2007. ACM.

[8] D. Clarke, R. Hinze, J. Jeuring, A. Löh, and J. de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Utrecht University, 2001.

[9] D. Fridlender and M. Indrika. Do we need dependent types? *Journal of Functional Programming*, 10(4):409–415, July 2000.

[10] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[11] R. Harper and R. Pollack. Type checking with universes. *Theoretical Computer Science*, 89:107–136, 1991.

[12] R. Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2–3):129–159, 2002. MPC Special Issue.

[13] R. Hinze. Generics for the masses. *Journal of Functional Programming*, 16 (4-5):451–483, 2006.

[14] R. Hinze and A. Löh. Generic programming in 3d. *Science of Computer Programming*, 74(8):590 – 628, 2009. doi: 10.1016/j.scico.2007.10.006. Special Issue on Mathematics of Program Construction (MPC 2006).

[15] P. Hoogendijk and R. Backhouse. When do datatypes commute? In *Proceedings of the 7th International Conference on Category Theory and Computer Science*, pages 242–260, London, UK, 1997. Springer-Verlag. ISBN 3-540-63455-X. URL `http://dl.acm.org/citation.cfm?id=648335.755730`.

[16] O. Kiselyov. Typed tagless final interpreters. In J. Gibbons, editor, *Generic and Indexed Programming*, Lecture Notes in Computer Science. Springer-Verlag, 2012. In this volume.

[17] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis-Napoli, 1984.

[18] C. McBride. Faking it: Simulating dependent types in Haskell. *Journal of Functional Programming*, 12(5):375–392, 2002.

[19] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[20] P. Morris, T. Altenkirch, and N. 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.

[21] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: an introduction.* Oxford University Press, 1990.

[22] U. 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.

[23] S. 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/`.

[24] S. L. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 50–61, Portland, OR, USA, Sept. 2006.

[25] T. Sheard. Putting Curry-Howard to work. In *Proceedings of the ACM SIGPLAN 2005 Haskell Workshop.* ACM Press, September 2005.

[26] T. Sheard. Generic programming in Ωmega. In R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 258–284. Springer, 2006.

[27] M. Sozeau. Equations: A dependent pattern-matching compiler. In *Interactive Theorem Proving*, July 2010.

[28] T. Strickland, S. Tobin-Hochstadt, and M. Felleisen. Practical variable-arity polymorphism. In G. Castagna, editor, *Programming Languages and Systems*, volume 5502 of *Lecture Notes in Computer Science*, pages 32–46. Springer Berlin / Heidelberg, 2009.

[29] 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/`.

[30] W. Verbruggen, E. de Vries, and A. Hughes. Polytypic programming in Coq. In *WGP '08: Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, pages 49–60, New York, NY, USA, 2008. ACM.

[31] W. Verbruggen, E. de Vries, and A. Hughes. Polytypic 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.
[32] W. Verbruggen, E. de Vries, and A. Hughes. Formal polytypic programs and proofs. *Journal of Functional Programming*, 20:213–270, 2010.
[33] S. Weirich. Type-safe run-time polytypic programming. *Journal of Functional Programming*, 16(10):681–710, Nov. 2006.
[34] S. Weirich. RepLib: A library for derivable type classes. In *Haskell Workshop*, pages 1–12, Portland, OR, USA, Sept. 2006.
[35] S. Weirich. Generic programming with dependent types: Lectures presented at the Spring School on Generic and Indexed Programming, 2010.
[36] S. Weirich and C. Casinghino. Arity-generic datatype-generic programming. In *PLPV '10: Proceedings of the 4th Workshop on Programming Languages Meets Program Verification*, 2010.

A Complete definition of arity-generic, type-generic framework

```
  -- an environment of vectors
data NGEnv {n : ℕ} (b : Vec Set (suc n) → Set)
                : Ctx → Set where
  NNil   : NGEnv b []
  NCons  : {k : Kind} {G : Ctx}
          → (a : Vec ⟦ k ⟧ (suc n))
          → b ⟨ k ⟩ a
          → NGEnv b G
          → NGEnv b (k :: G)
  -- interpret a type with a vector of different environments.
interp∗ : ∀ {G k n} → Typ G k → Vec (Env G) n
  → Vec ⟦ k ⟧ n
interp∗ t vs = repeat (interp t) ⊛ vs
  -- "transpose" an environment of vectors to a vector of environments
transpose : {n : ℕ} {b : Vec Set (suc n) → Set}
             {G : Ctx}
          → NGEnv b G → Vec (Env G) (suc n)
transpose NNil = repeat []
transpose (NCons a _ nge) =
  (repeat _::_) ⊛ a ⊛ (transpose nge)
```

The generic function generator needs some equalities to type check that cannot be shown automatically by Agda. The next few definitions prove those equalities.

```
  -- application is congruent
≡-app : ∀ {A} {b : A → Set} {t1} {t2} → t1 ≡ t2 → b t1 → b t2
≡-app refl x = x
  -- cons is congruent
≡-tail    : ∀ {A} {n} {t1 t2 : Vec A n} {x : A}
           → t1 ≡ t2
           → _≡_ {_} {Vec A (suc n)} (x :: t1) (x :: t2)
≡-tail {A} {n} refl = refl {_} {Vec A (suc n)}
  -- kind-indexed types are congruent
≡-KIT : {n : ℕ} {b : Vec Set (suc n) → Set}
           {k : Kind} {t1 t2 : Vec ⟦ k ⟧ (suc n)}
           → t1 ≡ t2
           → b ⟨ k ⟩ t1
           → b ⟨ k ⟩ t2
≡-KIT refl x = x
c1 : {n : ℕ} {k : Kind} {G : Ctx}
   → (a : Vec ⟦ k ⟧ n)
   → (envs : Vec (Env G) n)
   → a ≡ interp∗ (Var VZ) (repeat _::_ ⊛ a ⊛ envs)
c1 {zero} []        []        = refl
c1 {suc n} (t :: ts) (x :: xs) = ≡-tail (c1 {n} ts xs)
c2 : {n : ℕ} {k k' : Kind} {G : Ctx}
   → (x : V G k')
   → (t1 : Vec ⟦ k ⟧ n)
   → (envs : Vec (Env G) n)
   →  interp∗ (Var x) envs ≡
      interp∗ (Var (VS x)) (repeat _::_ ⊛ t1 ⊛ envs)
c2 {zero} x [] []              = refl
c2 {suc n} x (t :: ts) (y :: ys) = ≡-tail (c2 x ts ys)
c3 : {n : ℕ} {k k' : Kind} {G : Ctx}
   → (t : Typ (k' :: G) k)
   → (envs : Vec (Env G) n)
   → (as : Vec ⟦ k' ⟧ n)
   → (interp∗ (t) (repeat _::_ ⊛ as ⊛ envs))
   ≡ (interp∗ (Lam t) envs) ⊛ as
c3 {zero} t [] [] = refl
c3 {suc n} t (a :: as) (b :: bs) = ≡-tail (c3 t as bs)
c4 : {n : ℕ} {k1 k2 : Kind} {G : Ctx}
    → (t1 : Typ G (k1 ⇒ k2))
    → (t2 : Typ G k1)
    → (envs : Vec (Env G) n)
    →    (interp∗ (t1) envs) ⊛ (interp∗ (t2) envs)
      ≡ interp∗ (App t1 t2) envs
c4 {zero} _ _ [] = refl
```

```
c4 {suc n} t1 t2 (a :: as) = ≡-tail (c4 t1 t2 as)
c5 : {n : ℕ} {k : Kind} {G : Ctx}
  → (c : Const k)
  → (envs : Vec (Env G) n)
  → repeat ⌊ Con c ⌋ ≡ interp∗ (Con c) envs
c5 {zero} _ [] = refl
c5 {suc n} c (a :: as) = ≡-tail (c5 c as)
c6 : {n : ℕ} {G : Ctx}
    → (t2 : Typ G (⋆ ⇒ ⋆))
    → (envs : Vec (Env G) n)
    → (interp∗ t2 envs ⊛ (repeat μ ⊛ (interp∗ t2 envs)))
      ≡ interp∗ (App t2 (Mu t2)) envs
c6 {zero} _ [] = refl
c6 {suc n} t2 (a :: as) = ≡-tail (c6 t2 as)
c6' : {n : ℕ} {G : Ctx}
    → (t2 : Typ G (⋆ ⇒ ⋆))
    → (envs : Vec (Env G) n)
    → (repeat μ ⊛ (interp∗ t2 envs))
      ≡ interp∗ (Mu t2) envs
c6' {zero} _ [] = refl
c6' {suc n} t2 (a :: as) = ≡-tail (c6' t2 as)
c7 : {n : ℕ} {A B : Set} {f : A → B} {x : A} →
  (repeat {n} f) ⊛ repeat x ≡ repeat (f x)
c7 {zero} = refl
c7 {suc n} = ≡-tail c7


nLookup : {n : ℕ} {b : Vec Set (suc n) → Set}
            {k : Kind} {G : Ctx}
          → (v : V G k)
          → (nge : NGEnv b G)
          → b ⟨ k ⟩ (interp∗ (Var v) (transpose nge))
nLookup {n} {b} {k} VZ (NCons a e nge) =
  ≡-KIT (c1 a (transpose nge))   e
nLookup (VS x) (NCons a _ nge) =
  ≡-KIT (c2 x a (transpose nge)) (nLookup x nge)
MuGen : (n : ℕ) → (Vec Set (suc n) → Set) → Set
MuGen n b = ∀ {A} → b (A ⊛ (repeat μ ⊛ A)) → b (repeat μ ⊛ A)
ngen-open : {n : ℕ} {b : Vec Set (suc n) → Set} {G : Ctx} {k : Kind} →
  (t : Typ G k) → (ve : NGEnv b G) →
  (ce : ConstEnv b) → MuGen n b →
  b ⟨ k ⟩ (interp∗ t (transpose ve))
ngen-open (Var x) ve ce d = nLookup x ve
ngen-open {n} {b} (Lam {k1 = k1} t) ve ce d =
λ {a : Vec ⟦ k1 ⟧ (suc n)} (nwt : b ⟨ k1 ⟩ a) →
  ≡-KIT (c3 t (transpose ve) a)
  (ngen-open t (NCons a nwt ve) ce d)
ngen-open {n} {b} {G} (App {k1 = k1} {k2 = k2} t1 t2) ve ce d =
  ≡-KIT (c4 t1 t2 (transpose ve))
    ((ngen-open {n} {b} {G} {k1 ⇒ k2} t1 ve ce d)
    {(interp∗ t2 (transpose ve))} (ngen-open t2 ve ce d))
ngen-open (Con c) ve ce d = ≡-KIT (c5 c (transpose ve)) (ce c)
ngen-open {n} {b} (Mu t) ve ce d
        with (ngen-open (App t (Mu t)) ve ce d)
... | ng with d {(interp∗ t (transpose ve))}
... | BS = ≡-app {_} {b} (c6' t (transpose ve))
  (BS (≡-app {_} {b} (sym (c6 t (transpose ve))) ng))

ngen : {n : ℕ} {b : Vec Set (suc n) → Set} {k : Kind} →
  (t : Ty k) → (ConstEnv b) → MuGen n b → b ⟨ k ⟩ (repeat ⌊ t ⌋)
ngen {n} {b} {k} t ce d = ≡-KIT {n} {b} {k} c7 (ngen-open t NNil ce d)
```