1-19-2010

Arity-Generic Datatype-Generic Programming

Stephanie Weirich  
*University of Pennsylvania, sweirich@cis.upenn.edu*

Chris Casinghino  
*University of Pennsylvania*

Follow this and additional works at: [http://repository.upenn.edu/cis_papers](http://repository.upenn.edu/cis_papers)

Part of the Computer Sciences Commons

Recommended Citation


Stephanie Weirich and Chris Casinghino. *Arity-generic type-generic programming*. In *ACM SIGPLAN Workshop on Programming Languages Meets Program Verification (PLPV)*, pages 15-26, January 2010  
© ACM, 2010. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *ACM SIGPLAN Workshop on Programming Languages Meets Program Verification (2010)* [http://doi.acm.org/10.1145/1707790.1707799](http://doi.acm.org/10.1145/1707790.1707799)  
Email permissions@acm.org

This paper is posted at ScholarlyCommons. [http://repository.upenn.edu/cis_papers/631](http://repository.upenn.edu/cis_papers/631)  
For more information, please contact repository@pobox.upenn.edu.
Arity-Generic Datatype-Generic Programming

Abstract
Some programs are doubly-generic. For example, map is datatype-generic in that many different data structures support a mapping operation. A generic programming language like Generic Haskell can use a single definition to generate map for each type. However, map is also arity-generic because it belongs to a family of related operations that differ in the number of arguments. For lists, this family includes repeat, map, zipWith, zipWith3, zipWith4, etc. With dependent types or clever programming, one can unify all of these functions together in a single definition. However, no one has explored the combination of these two forms of genericity. These two axes are not orthogonal because the idea of arity appears in Generic Haskell: datatype-generic versions of repeat, map and zipWith have different arities of kind-indexed types. In this paper, we define arity-generic datatype-generic programs by building a framework for Generic Haskell-style generic programming in the dependently-typed programming language Agda 2.

Disciplines
Computer Sciences

Comments
Stephanie Weirich and Chris Casinghino. Arity-generic type-generic programming. In ACM SIGPLAN Workshop on Programming Languages Meets Program Verification (PLPV), pages 15-26, January 2010

© ACM, 2010. This is the author’s version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in {ACM SIGPLAN Workshop on Programming Languages Meets Program Verification (2010)} http://doi.acm.org/10.1145/1707790.1707799
Email permissions@acm.org
Arity-Generic Datatype-Generic Programming

Stephanie Weirich  Chris Casinghino
University of Pennsylvania
{sweirich,ccasin}@cis.upenn.edu

Abstract

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

With dependent types or clever programming, one can unify all of these functions together in a single definition.

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

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.3 [Programming Languages]: Language constructs and features—Data types and structures, frameworks

General Terms Design, Languages, Verification

Keywords Dependent types, Arity-generic programming, Agda, Generic Haskell

1. Introduction

This is a story about doubly-generic programming. Datatype-generic programming defines operations that may be instantiated at many different types, so these operations need not be redefined for each one. For example, Generic Haskell [5, 11] includes a generic map operation gmap that has instances for types such as lists, option-al values, and products (even though these types have different kinds).

\begin{align*}
gmap \langle [] \rangle &:: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
gmap \langle \text{Maybe} \rangle &:: (a \rightarrow b) \rightarrow \text{Maybe } a \rightarrow \text{Maybe } b \\
gmap \langle (,) \rangle &:: (a1 \rightarrow b1) \rightarrow (a2 \rightarrow b2) \rightarrow (b1, b2)
\end{align*}

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

However, there is another way to generalize map. Consider the following sequence of functions from the Haskell Prelude [20], all of which operate on lists.

\begin{align*}
\text{repeat} :: & a \rightarrow [a] \\
\text{map} :: & (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\text{zipWith} :: & (a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c] \\
\text{zipWith3} :: & (a \rightarrow b \rightarrow c \rightarrow d) \rightarrow [a] \rightarrow [b] \rightarrow [c] \rightarrow [d]
\end{align*}

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

As Fridlender and Indrika [7] 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.

Arity-genericity is not unique to the list instance of map. It is not difficult to imagine arity-generic versions of map for other types. Fridlender and Indrika’s technique immediately applies to all types that are applicative functors [16]. However, one may also define arity-generic versions of map at other types.

Other functions besides map have both datatype-generic and arity-generic versions. For example, equality can be applied to any number of arguments, all of the same type. Map has a dual operation called unzipWith that is similarly doubly-generic. Other examples include folds, enumerations, monadic maps, etc.

In this paper, we present the first doubly-generic definitions. For each of these examples, we can give a single definition that can be instantiated at any type or at any arity. Our methodology shows how these two forms of genericity can be combined in the same framework and demonstrates the synergy between them.

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 to the arities in map above. For example, the Generic Haskell version of repeat has arity one, its map has arity two, and zipWith arity three.

Unfortunately, Generic Haskell does not permit generalizing over arities, so a single definition cannot produce repeat, map and zipWith. Likewise, Generic-Haskell-style libraries encoded in Haskell, such as RepLib [29] or Extensible and Modular Generics for the Masses (EMGM) [6] specialize their infrastructure to specific arities, so they too cannot write arity-generic code.

However, Altenkirch and McBride [1] and Verbruggen et al. [26] have shown how to encode Generic-Haskell-style generic programming in dependently-typed programming languages. Although they do not consider arity-genericity in their work, be-
cause of the power of dependent-types, their encodings are flexible enough to express arity-generic operations.

In this paper, we develop an analogous generic programming framework in the dependently-typed language Agda 2 [19] and demonstrate how it may be used to define doubly-generic operations. We choose Agda because it is particularly tailored to dependently-typed programming, but we could have also used a number of different languages, such as Coq [25], Epigram [15], Ω mega [23], or Haskell with recent extensions [3, 21].

Our contributions are as follows:

1. We develop an arity-generic version of map that reveals commonality between gmap, gzipWith and gzipWith3. This correspondence has not previously been expressed, but we find that it leads to insight into the nature of these operations. Since the definitions are all instances of the same dependently-typed function, we have shown formally that they are related.

2. This example is developed on top of a reusable framework for generic programming in Agda. Although our framework has the same structure as previous work, our treatment of datatype isomorphisms is novel and requires less boilerplate.

3. We use our framework to develop other doubly-generic operations, such as equality and unzipWith. All of these examples shed light on arity support in a generic programming framework. In particular, there are not many operations that require arity of two or more: this work suggests what such operations must look like.

4. Finally, because we develop a reusable framework, this work demonstrates how a tool like Generic Haskell could be extended to arity-genericity.

We explain doubly-generic map and our generic programming infrastructure in stages. In Section 2 we start with an Agda definition of arity-generic map. Next, in Section 3, we describe a general framework for generic programming that works for all types (of any kind) formed from unit, pairs, sums and natural numbers. We use this framework to define doubly-generic map. In Section 4 we show how datatype isomorphisms may be incorporated, so that we can specialize doubly-generic operations to inductive datatypes. We discuss other doubly-generic examples in Section 5. Finally, Sections 6 and 7 discuss related work and conclude.

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

2. Arity-General Map

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

2.1 Programming with Dependent Types in Agda

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

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

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

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

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

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

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

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

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

The _⊛_ operator associates 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.

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

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

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

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

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

2.2 Arity-General Vector Map

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

We prefer to use natural numbers to express the arity of the mapping operation. Therefore, we must program with Agda types. For example, we can construct a vector of Agda types, Bool :: ℕ :: [], which has type Vec Set 2, and use standard vector operations (such as _⊛_) with this value.

Note that Unicode symbols are valid in Agda identifiers.
The first step is to define arrTy, which folds the arrow type constructor \( \rightarrow \) over a non-empty vector of types. Given such a vector, this operation constructs the type of the function that will be mapped over the n data structures.

\[
\text{arrTy} : \{ n : \text{Nat} \} \rightarrow \text{Vec Set} (\text{succ} n) \rightarrow \text{Set}
\]

\[
\text{arrTy} \{ 0 \} (A :: []) = A
\]

\[
\text{arrTy} \{ \text{succ} n \} (A :: As) = A \rightarrow \text{arrTy} As
\]

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 \( m \) and \( n \) as implicit arguments, so we need not always match against them in definitions nor provide them explicitly as arguments.

\[
\text{arrTyVec} : \{ m \ n : \text{Nat} \} \rightarrow \text{Vec Set} (\text{succ} n) \rightarrow \text{Set}
\]

\[
\text{arrTyVec} \{ m \} \text{As} = \text{arrTy} (\text{repeat} (\lambda A \rightarrow \text{Vec} A m) \odot \text{As})
\]

For example, we can define the sequence of types from Section 2.1 using these functions applied to lists of type variables.

\[
\text{map0} : \{ m : \text{Nat} \} \rightarrow (A :: [])
\]

\[
\text{map1} : \{ m \ n : \text{Nat} \} \rightarrow (A :: B :: [])
\]

\[
\text{map2} : \{ m \ n : \text{Nat} \} \rightarrow (A :: B :: C :: [])
\]

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

\[
\text{nvec-map} : \{ m : \text{Nat} \} \rightarrow (n : \text{Nat})
\]

\[
\rightarrow (A :: \text{Vec Set} (\text{succ} n))
\]

\[
\rightarrow \text{arrTy} As \rightarrow \text{arrTyVec} \{ m \} \text{As}
\]

Intuitively, the definition of nvec-map is a simple application of repeat and \( n \) copies of \( \odot \).

\[
\text{nvec-map} As f v1 v2 \ldots vn = \text{repeat} f \odot v1 \odot v2 \odot \ldots \odot vn
\]

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

\[
\text{nvec-map} n As f g = g \{ n \} As (\text{repeat f} \text{where})
\]

\[
g : \{ m \ n : \text{Nat} \}
\]

\[
\rightarrow (A :: \text{Vec Set} (\text{succ} n))
\]

\[
\rightarrow \text{Vec} (\text{arrTy} As) m \rightarrow \text{arrTyVec} \{ m \} \text{As}
\]

\[
g \{ 0 \} (A :: []) a = a
\]

\[
g \{ \text{succ} n \} (A :: A :: As) f = (\lambda a \rightarrow g As (f \odot a))
\]

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

\[
\forall \Rightarrow : \{ n : \text{Nat} \} \rightarrow (A :: \text{Set}) \rightarrow (\text{Vec} A n \rightarrow \text{Set})
\]

\[
\forall \Rightarrow \{ \text{zero} \} B = [B]
\]

\[
\forall \Rightarrow \{ \text{succ} n \} (A :: As) = \{ a : A \rightarrow \forall \Rightarrow \{ n \} (\lambda as \rightarrow B (a :: as))\}
\]

\[
\lambda \Rightarrow : \{ n : \text{Nat} \} \rightarrow (A :: \text{Set}) \rightarrow (\text{Vec} A n \rightarrow \text{Set})
\]

\[
\rightarrow ((X : \text{Vec A n}) \rightarrow B X) \rightarrow (\forall \Rightarrow B)
\]

\[
\lambda \Rightarrow \{ \text{zero} \} f = f [][
\]

\[
\lambda \Rightarrow \{ \text{succ} n \} (A :: As) = \lambda (a : A) \rightarrow \forall \Rightarrow \{ n \} (\lambda as \rightarrow f (a :: as))
\]

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

\[
\text{nmap} : \{ n : \text{Nat} \} \rightarrow \{ m : \text{Nat} \}
\]

\[
\rightarrow \forall \Rightarrow (\lambda (\text{As} : \text{Vec Set} (\text{succ} n))
\]

\[
\rightarrow \text{arrTy} As \rightarrow \text{arrTyVec} \{ m \} \text{As}
\]

\[
\text{nmap} n \{ m \} \Rightarrow (\text{nvec-map} \{ m \} n)
\]

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

\[
\{ m : \text{Nat} \} \rightarrow (A :: B :: \text{Set}) \rightarrow (A \rightarrow B)
\]

\[
\rightarrow (\text{Vec} A m) \rightarrow (\text{Vec} B m)
\]

and the expression

\[
\text{nmap} 1 (\lambda (x : x + 1) (1 :: 2 :: 3 :: []))
\]

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

\[
\{ m : \text{Nat} \} \rightarrow (1 :: 2 :: 3 :: []) \rightarrow (1 :: 2 :: 3 :: 6 :: [])
\]

\[
\rightarrow (\text{Vec} A m) \rightarrow (\text{Vec} C m)
\]

and the expression

\[
\text{nmap} 2 (\lambda \_ \_ (1 :: 2 :: 3 :: [])) (4 :: 5 :: 6 :: []]
\]

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

2.3 Towards Type Genericiity

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

Type-generic programming in Agda is done via a technique called "universe" [13, 18]. The idea is to define an inductive datatype Tyc, called a universe, which represents types, along with an interpretation function \([\_, \_]\) that maps elements of this universe to actual Agda types. A generic program is then an operation which manipulates this data structure.

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

Consider the following universe of codes for type constructors.

\[
\text{data} \ Tyc : \text{Set where}
\]

\[
\text{Nat} : \text{Tyc}
\]

\[
\text{Unit} : \text{Tyc}
\]

\[
\text{Prod} : \text{Tyc} \rightarrow \text{Tyc} \rightarrow \text{Tyc}
\]

\[
\text{Arr} : \text{Nat} \rightarrow \text{Tyc} \rightarrow \text{Tyc}
\]

\[
\text{Var} : \text{Tyc}
\]
Each of these codes can be decoded as an Agda type constructor of kind \texttt{Set} \rightarrow \texttt{Set}. For example, \texttt{T} is the unit type in Agda and \texttt{\times} constructs the (non-dependent) type of products.

\[
\begin{array}{ll}
| \_ \_ | & \rightarrow \text{Set} \\
| \texttt{Nat} | & a = \mathbb{N} \\
| \texttt{Unit} | & a = \top \\
| \texttt{Prod} t_1 t_2 | & a = [ t_1 \mid a \times t_2 \mid a ] \\
| \texttt{Arr} n t_1 | & a = \text{Vec} [ t_1 \mid a ] n \\
| \texttt{Var} | & a = a
\end{array}
\]

With these two definitions, we can implement type-generic versions of the repeat and \_@\_ functions.\footnote{Because these generic functions show that all type constructors in this universe are applicative, we cannot include a code for sum types. We return to this issue in Section 3.3.} They are implemented by recursion on the structure of the \texttt{Tyc}, but we elide the definitions for brevity.

\[
\begin{array}{ll}
g\text{repeat} : (t : \text{Tyc}) \rightarrow (a : \text{Set}) \rightarrow a \rightarrow [ t ] a \\
g\text{zg\text{ap}} : (t : \text{Tyc}) \rightarrow (a b : \text{Set}) \\
& \rightarrow [ t ] (a \rightarrow b) \rightarrow [ t ] a \rightarrow [ t ] b
\end{array}
\]

With these type-generic functions, we can generalize the definition for vectors to produce \texttt{gmap}, which works for all type constructors in the universe. This definition is a straightforward extension of \texttt{nmap} for vectors (replacing repeat and \_@\_ with \texttt{grepeat} and \texttt{gzap}), so we elide its definition and only show its type.

\[
g\text{map} : (t : \text{Tyc}) \rightarrow (n : \mathbb{N}) \\
& \rightarrow \forall b (\lambda : \text{Vec Set (suc n)}) \\
& \rightarrow \text{arrTy As } \rightarrow \text{arrTy} (\text{repeat} [ t ] \_ @ \_ a)
\]

We use \texttt{gmap} by supplying a type code and arity. For example,

\[
\begin{array}{l}
\text{example-map} : (m : \mathbb{N}) \rightarrow (A B : \text{Set}) \rightarrow (A \rightarrow B) \\
& \rightarrow \text{Vec} (A \times (A \times B)) m \\
& \rightarrow \text{Vec} (B \times (B \times T)) m
\end{array}
\]

\[
\begin{array}{l}
\text{example-map} = \\
\text{gmap} (\text{Arr } \_ (\text{Prod Var (Prod Var Unit)))) 1
\end{array}
\]

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

To extend arity-generic map to types of arbitrary kinds, we will redo our framework for type-generic programming using a kind-indexed universe. The kind determines the type of the decoding function \_@\_. With this kind-indexed universe, the concept of arity naturally shows up—following Generic Haskell, a generic function has a kind-indexed type of a particular arity. For example, generic repeat requires an arity one kind-indexed type, while generic map requires arity two, and generic \texttt{zipWith} requires arity three. Remarkably, but perhaps unsurprisingly, this notion of arity mirrors the arity found in arity-generic map.

What is new in this paper is that we generalize over the arities in the kind-indexed types to give a completely new definition of arity-generic type-generic map. This definition incorporates arity-genericity right from the start. In the current section we layered arity-genericity on top of type-genericity; in the next, our type-generic functions will be inherently arity-generic.

\section{Arity-Generic Type-Generic Map}

Next, we show how to generalize arity-generic map to arbitrary type constructors by implementing a framework for Generic Haskell style kind-indexed types. We develop our framework in stages, first including only primitive type constructors in the universe, then in Section 4, extending it to include user-defined datatypes.

\subsection{Universe Definition}

To write more general generic programs, we need a more expressive universe. The universe that we care about is based on the type language of F-omega [8]. It is the simply-typed lambda calculus augmented with a number of constants that form types. Therefore, to represent this language, we need datatypes for kinds, constants, and for the lambda calculus itself.

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

\texttt{data} \texttt{Kind} : \texttt{Set} \texttt{where} \\
\_@\_ : \texttt{Kind} \rightarrow \texttt{Kind} \rightarrow \texttt{Kind}

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

\texttt{data} \texttt{Const} : \texttt{Kind} \rightarrow \texttt{Set} \\
\_@\_ : \texttt{Kind} \rightarrow \texttt{Const} \\
\texttt{Unit} : \texttt{Const} \\
\texttt{Sum} : \texttt{Const} \rightarrow \texttt{Const} \rightarrow \texttt{Const} \\
\texttt{Prod} : \texttt{Const} \rightarrow \texttt{Const} \rightarrow \texttt{Const}

Constants are indexed by their kinds. For now, we will concentrate on types formed from natural numbers, unit, binary sums, and binary products. Note that these definitions include a code for sum types. Although doubly-generic map is partial for sums, many doubly-generic operations are not. On the other hand, most generic functions are partial for function types, so we do not include a code for them. Furthermore, because vectors are representable in terms of the other constructors, we do not include a code for them in this universe. This keeps the definitions of arity-generic functions simple. In Section 4, we discuss how our generic programming framework can interface directly with Agda datatypes like \texttt{Vec}.

\texttt{data} \texttt{Typ} : \texttt{Kind} \rightarrow \texttt{Set} \texttt{where} \\
\texttt{VZ} : \forall (k : \texttt{G} k) \rightarrow \texttt{Typ} (k :: \texttt{G} k) \\
\texttt{VS} : \forall (k k' : \texttt{G} k) \rightarrow \texttt{Typ} (k' :: \texttt{G} k)

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

\texttt{data} \texttt{Ctx} : \texttt{Set} \\
\texttt{Ctx} = \texttt{List} \texttt{Kind}

\texttt{data} \texttt{TyVar} : \texttt{Ctx} \rightarrow \texttt{Kind} \rightarrow \texttt{Set} \texttt{where} \\
\texttt{TyVar} : \forall (k : \texttt{G} k) \rightarrow \texttt{TyVar} (k :: \texttt{G} k) \\
\texttt{TyVar} : \forall (k k' : \texttt{G} k) \rightarrow \texttt{TyVar} (k' :: \texttt{G} k) \\
\texttt{TyVar} : \forall (k k' : \texttt{G} k) \rightarrow \texttt{TyVar} (k' :: \texttt{G} k)

\texttt{data} \texttt{Typ} : \texttt{Ctx} \rightarrow \texttt{Kind} \rightarrow \texttt{Set} \texttt{where} \\
\texttt{Typ} : \forall (k : \texttt{G} k) \rightarrow \texttt{Typ} (k :: \texttt{G} k) \\
\texttt{Typ} : \forall (k k' : \texttt{G} k) \rightarrow \texttt{Typ} (k' :: \texttt{G} k)
Lam : ∀ {G k1 k2} → TyG k1 → TyG k2 → TyG (k1 → k2)
App : ∀ {G k1 k2} → TyG (k1 ⇒ k2) → TyG k1 → TyG k2 → TyG k1
Con : ∀ {G k} → Const k → TyG k

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

Ty : Kind → Set
Ty = TyP [ ]

Now that we can represent type constructors, we need a mechanism to decode them as Agda types. To do so, we must have an environment to decode 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. Note that this definition overloads the [] and _:_ _ constructors, but Agda can infer which we mean.

data Env : List Kind → Set where
[] : Env []
_:_ : ∀ {G} → [ G ] → Env G → Env (k :: G)

sLookup : ∀ {G} → TyVar G k → Env G → [ k ]
sLookup VZ (v :: G) = v
sLookup (VS x) (v :: G) = sLookup x G

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

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

_ [] : ∀ {G} → TyG k → [ k ]
_ t : = interp t []

For example, the following type constructor Option (isomorphic to the standard Maybe datatype)

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

is represented with the following code:

option : Ty (⋆ ⇒ ⋆)
option =

Lam (App (App (Con Sum) (Con Unit)) (Var VZ))

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

3.2 Framework for Doubly-General-Programming

Next, we give the signature of a framework for defining arity-generic type-generic programs. For space reasons, we do not give the implementation of this framework here. The interested reader may consult Altenkirch and McBride [1], Verbruggen et al. [26], or our source code for more details.

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

Datatype-generic operations have different types when instantiated at different kinds, so they are described by kind-indexed types [11]. For example, consider the type of the standard map function for the Option type constructor, of kind ⋆ ⇒ ⋆:

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

And map for the type constructor _×_, of kind ⋆ ⇒ ⋆ ⇒ ⋆:


Though different, the types of option-map1 and pair-map1 are instances of the same kind-indexed type. In Generic Haskell, kind-indexed types are defined by recursion on the kind of the type arguments. For example, here is the Generic Haskell definition of map’s type [12]:

type Map (⋆) t1 t2 = t1 → t2

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

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

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

pair-map0 : ∀ {A B : Set} → A → B → A × B

We compute the type of a generic function instance from four pieces of information: the arity of the operation (given with an implicit argument n), a function b to construct the type in the base case, the kind k itself and a vector v of n Agda types, each of kind k. Reminiscent of Generic Haskell, our kind-indexed type is written b (k) v:

b (k) v : = b v

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

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

We do not allow these vectors to be empty because few generic functions make sense at arity zero. If we had allowed empty vectors we would have to add a degenerate zero case for the majority of generic functions. It would be straightforward, but tedious, to remove this restriction. As a result, the number provided here as an arity (n) is one less than the corresponding Generic Haskell arity.

We refer to this reduced number as the arity for convenience.
We define generic functions with ngen, whose type is shown below. This operation produces a value of a kind-indexed type given ce, a mapping from constants to appropriate definitions.

\[
\begin{align*}
\text{ngen} : \{ n : N \} \cdot \{ b : \text{Vec Set (suc n)} \rightarrow \text{Set} \} \cdot \{ k : \text{Kind} \} \rightarrow (t : \text{Ty} k) \\
\rightarrow (\text{ce} : \text{TyConstEnv n b}) \\
\rightarrow b (k) (\text{repeat } t)
\end{align*}
\]

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

\[
\begin{align*}
\text{TyConstEnv} : \{ n : N \} \rightarrow (b : \text{Vec Set n} \rightarrow \text{Set}) \\
\rightarrow (k : \text{Kind}) \rightarrow (c : \text{Const k}) \rightarrow b (k) (\text{repeat } c)
\end{align*}
\]

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

\[
\begin{align*}
\text{Map} : \text{Vec Set} 2 \rightarrow \text{Set} \\
\text{Map} (A :: B :: []) = A \rightarrow B
\end{align*}
\]

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

\[
\begin{align*}
\text{gmap-const} : \text{TyConstEnv GMap} \\
\text{gmap-const Nat} = \lambda x \rightarrow x \\
\text{gmap-const Unit} = \lambda x \rightarrow x \\
\text{gmap-const Prod} = \lambda f g x \rightarrow (f (\text{proj1 x}), g (\text{proj2 x})) \\
\text{gmap-const Sum} = g
\end{align*}
\]

where

\[
\begin{align*}
g & : \{ A1 B1 A2 B2 : \text{Set} \} \\
& \rightarrow (A1 \rightarrow B1) \rightarrow (A2 \rightarrow B2) \\
& \rightarrow A1 \times A2 \rightarrow B1 \times B2 \\
& \text{g fa \ fb (inj1 xa) = inj1 (fa xa)} \\
& \text{g fa \ fb (inj2 xb) = inj2 (fb xb)}
\end{align*}
\]

Generic map then calls ngen with this argument.

\[
\begin{align*}
\text{gmap} : \{ k : \text{Kind} \} \rightarrow (t : \text{Ty} k) \\
\rightarrow \text{Map} (k) (\{ t \} :: \{ t \} :: []) \\
\text{gmap t = ngen t gmap-const}
\end{align*}
\]

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

\[
\begin{align*}
\text{option-map1} : (A : B : \text{Set}) \rightarrow (A \rightarrow B) \\
\rightarrow \text{Option} A \rightarrow \text{Option} B \\
\text{option-map1 = gmap option}
\end{align*}
\]

### 3.3 Doubly-generic Map

To use ngen to implement a doubly-generic function, we must also supply b and ce to ngen, but this time both of those arguments must generalize over the arity. For doubly-generic map, we call these pieces NGmap and ngen-const. NGmap is simply the arrTy function from Section 2.2, which takes the arity as an implicit argument.

\[
\begin{align*}
\text{NGmap} : \{ n : N \} \rightarrow \text{Vec Set (suc n)} \rightarrow \text{Set} \\
\text{NGmap} = \text{arrTy}
\end{align*}
\]

We are simplifying the example somewhat because generic zips (which are generic maps at arities greater than one) are partial functions—they may fail if instantiated at a sum type and passed mismatched injections. To account for this possibility in Generic Haskell, the library function zipWith returns a Maybe. However, we would like to keep our presentation as simple as possible, so we use an error term to indicate failure. A version of doubly-generic map that returns a Maybe is included with our sources. Because Agda lacks Haskell’s error function, we use a postulate:

\[
\text{postulate error} : (A : \text{Set}) \rightarrow A
\]

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

\[
\begin{align*}
\text{ngmap-const} : \{ n : N \} \rightarrow \text{TyConstEnv NGmap} \\
\text{ngmap-const Nat} = \text{defNat n} \\
\text{ngmap-const Unit} = \text{defUnit n} \\
\text{ngmap-const Prod} = \text{defPair n} \\
\text{ngmap-const Sum} = \text{defSum n}
\end{align*}
\]

Recalling the definition of NGmap, for the first two cases, we must return arity-n functions with the types \( N \rightarrow N \rightarrow \ldots \rightarrow N \) and \( T \rightarrow T \rightarrow \ldots \rightarrow T \). For \( N \), when \( n \) is zero, we pick a default element to return arbitrarily. When \( n \) is one, we return the argument. For larger arities, we check that the inputs are identical. We choose to reject unequal nats to mirror the behavior of map at sum types.

\[
\begin{align*}
\text{defNat} : \{ n : N \} \rightarrow \text{NGmap } (\ast ) \rightarrow \text{(repeat } \{ \text{suc n} \} N) \\
\text{defNat zero} = \text{zero } -- \text{arbitrary N} \\
\text{defNat (suc zero)} = \lambda x \rightarrow x -- \text{return what was given} \\
\text{defNat (suc suc n)} = \\
\text{\lambda x \rightarrow \text{if eqNat x y then defNat (suc n) y else error}} \\
\text{defUnit} : \{ n : N \} \rightarrow \text{NGmap } (\ast ) \rightarrow \text{(repeat } \{ \text{suc n} \} T) \\
\text{defUnit zero} = \text{tt} \\
\text{defUnit (suc n)} = \lambda x \rightarrow \text{(defUnit n)}
\end{align*}
\]

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

\[
\begin{align*}
\text{NGmap} (\ast \mapsto \ast \mapsto \ast ) \rightarrow \text{(repeat } \{ \text{suc n} \} \ldots \ldots ) = \\
\forall \Rightarrow \lambda (b : \text{Vec Set n}) \rightarrow \text{arrTy As} \rightarrow \\
\forall \Rightarrow \lambda (B : \text{Vec Set n}) \rightarrow \text{arrTy Bs} \rightarrow \\
\text{arrTy } (\text{repeat } \ast \ast ) \otimes A \otimes B
\end{align*}
\]

If we imagine writing out As as \( A1 :: A2 :: \ldots :: An :: [] \) and Bs as \( B1 :: B2 :: \ldots :: Bn :: [] \) the type simplifies to:

\[
\begin{align*}
\text{NGmap} (\ast \mapsto \ast \mapsto \ast ) \rightarrow \text{(repeat } \{ \text{suc n} \} \ldots \ldots ) = \\
\{ A1 A2 ... An :: Set \} \rightarrow (B1 \rightarrow A2 \rightarrow \ldots \rightarrow An) \\
\rightarrow \{ B1 B2 ... Bn :: Set \} \rightarrow (B1 \rightarrow B2 \rightarrow \ldots \rightarrow Bn) \\
\rightarrow (A1 \times B1) \rightarrow (A2 \times B2) \rightarrow \ldots \rightarrow (An \times Bn)
\end{align*}
\]

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

\[
\begin{align*}
\text{defPairAux} : \{ n : N \} \rightarrow \{ \text{As : Vec Set (suc n)} \} \rightarrow \text{arrTy As} \\
\rightarrow \{ \text{Bs : Vec Set (suc n)} \} \rightarrow \text{arrTy Bs} \\
\rightarrow \text{arrTy } (\text{repeat } \ast \ast ) \otimes A \otimes B \\
\text{defPairAux zero } (A :: []) a (B :: []) = (a, b) \\
\text{defPairAux (suc n) (A1 :: As) a (B1 :: Bs) b =} \\
\lambda p \mapsto \\
\text{(defPairAux n As (a (proj1 p))) Bs (b (proj2 p)))}
\end{align*}
\]

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

\[
defSumLeft : (n : N) \\
→ (As : Vec Set (succ n)) → arrTy As \\
→ (Bs : Vec Set (succ n)) → arrTy (repeat _×_ ⊎ As ⊎ Bs) \\
defSumLeft zero (A :: []) a (B :: []) = inj₁ a \\
defSumLeft (succ n) (A :: As) a (B :: Bs) = f \\
where \f : A1 ⊔ B1 → arrTy (repeat _×_ ⊎ As ⊎ Bs) \\
f (inj₁ a₁) = defSumLeft n As (a₁ B₁ Bs) \\
f (inj₂ b₁) = defSumRight n As (a ∈ error A1) Bs
\]

Below, defSumAux checks if the first argument is an inj₁ or an
inj₂, then calls defSumLeft or defSumRight which require that all
subsequent arguments match. In the degenerate case, there where
are no arguments, we arbitrarily choose the right injection. Because
defSumRight is analogous to defSumLeft, we elide its definition below.

\[
defSumRight : (n : N) \\
→ (As : Vec Set (succ n)) → arrTy As \\
→ (Bs : Vec Set (succ n)) → arrTy (repeat _×_ ⊎ As ⊎ Bs) \\
defSumRight zero (A :: []) a (B :: []) = inj₂ a \\
defSumRight (succ n) (A :: As) a (B :: Bs) = f \\
where \f : A1 ⊔ B1 → arrTy (repeat _×_ ⊎ As ⊎ Bs) \\
f (inj₂ b₁) = defSumLeft n As (a₁ B₁ Bs) \\
f (inj₁ b₁) = defSumRight n As (a ∈ error A1) Bs
\]

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

\[
defSum : (n : N) \\
→ NGmap (⋆ ⊔ ⋆ → ⋆) (repeat {suc n} _×_) \\
defSum n = k-curry (⋆ ⊔ ⋆ → ⋆) (defSumAux n)
\]

We can then define nmap by instantiating ngen.

\[
nmap : (n : N) → k : Kind) → (e : Ty k) \\
nmap (k) (repeat {suc n} [ e ]) = ngen e nmap-const
\]

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

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

\[
option-map₁ : { A : : Set} → (A → B) \\
→ Option A → Option B
\]

\[
option-map₁ = nmap₁ option
\]

4. Datatype Isomorphisms

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

4.1 Representing Datatypes

There are at least two ways to support datatypes. The current sys-
tem already can encode datatypes on an ad hoc basis, in a manner
described by Verbruggen et al. [26]. However, this encoding re-
quires some tedious applications of coercions between the datatype
and its isomorphism for each datatype instance of the generic op-
eration. Instead, we move that boilerplate to the generic function
itself by adding a new constructor to the Typ universe. This new
constructor, Data, contains information about a particular datatype.

\[\text{data Typ : Ctx → Kind → Set where}\]

\[\text{Data : } \forall \{ \text{G} \} \rightarrow \text{DT G} \rightarrow \text{Typ G *}\]

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

\[\text{data DT (G : Ctx) : Set where}\]

\[\text{mkDT : } (t : \text{Typ G *}) \\
→ (s : \text{Env G} → \text{Set}) \\
→ (to : ((e : \text{Env G}) → \text{interp e t e → s e})) \\
→ (from : ((e : \text{Env G}) → \text{e → interp t e})) \\
→ \text{DT G}\]

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

We describe this mechanism in more detail below.

We define a number of accessor functions for retrieving the parts
of a DT, called DT-s, DT-t, DT-from and DT-to (here elided).
Because interp is mentioned by the components of DT, it must
be defined mutually with Env, sLookup, Typ, DT, and its accessor.
Finally, we extend the interpretation function for codes by look-
ing up the Agda type and giving it the current environment.

\[\text{interp : } \forall \{ k \text{ G} \} \rightarrow \text{Typ G k} \rightarrow \text{Env G} → [ k ]\]

\[\text{... interp (Data dt) e = DT-s dt e}\]

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

\[\text{option-map₂ : } \{ A B C : \text{Set} \} → (A → B → C) \\
→ \text{Option A} → \text{Option B} → \text{Option C}\]

\[\text{option-map₂ = nmap₂ 2 option}\]
data Orange : Set where
toOrange : N \rightarrow Orange

We can form the code for this datatype as below.

fromOrange : Orange \rightarrow N
fromOrange (toOrange x) = x

orange : \{ G : Ctx \} \rightarrow Typ G *

orange = Data (mkDT
  (Con Nat) -- t
  (\lambda e \rightarrow Orange) -- s
  (\lambda e \rightarrow toOrange) -- to
  (\lambda e \rightarrow fromOrange) -- from)

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

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

maybeDef : \{ G : Ctx \} \rightarrow Typ G \star \rightarrow Typ G *
maybeDef t = (App (App (Con Sum) (Con Unit)) t)

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

toMaybe : \{ G : Ctx \}, \{ e : Env G \} \rightarrow (t : Typ G \star) \rightarrow Maybe t
toMaybe t (inj1 _) = nothing
toMaybe t (inj2 x) = just x

fromMaybe : \{ G : Ctx \}, \{ e : Env G \} \rightarrow (t : Typ G \star) \rightarrow (\lambda e \rightarrow Maybe t e)
fromMaybe t (just x) = inj2 x
fromMaybe t nothing = inj1 tt

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

maybe : \{ G : Ctx \} \rightarrow Typ G (\star \Rightarrow \star)
maybe \{ G \} = Lam (Data
  (mkDT (maybeDef (Var VZ))
    (\lambda e \rightarrow Maybe (interp (Var VZ) e))
    (\lambda e \rightarrow toMaybe \{ * :: G \} \{ e \} (Var VZ))
    (\lambda e \rightarrow fromMaybe \{ * :: G \} \{ e \} (Var VZ))))

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

vecDef : \{ G : Ctx \} \rightarrow Typ G \star \rightarrow (n : N) \rightarrow Typ G *
vecDef 0 = Con Unit
vecDef n = (App (App (Con Prod) t) (vecDef t n))

fromVec : \{ n : N \} \{ G : Ctx \}
  \{ t : Typ G \star \} \{ e : Env G \} \rightarrow Vec (interp t e) n \rightarrow (interp (vecDef t n) e)
fromVec {} [] = tt
fromVec \{ suc n \} (x :: xs) = (x, fromVec xs)

toVec : \{ n : N \} \{ G : Ctx \} \{ t : Typ G \star \} \{ e : Env G \} \rightarrow Vec (interp t e) n \rightarrow Vec (interp t e) n
toVec 0 = []
toVec (suc n) (x :: xs) = (x :: toVec xs)

 vec : \{ G : Ctx \} \rightarrow (n : N) \rightarrow Typ G (\star \Rightarrow \star)
 vec \{ G \} \{ n \} = Lam (Data
  (mkDT (vecDef (Var VZ) n)
    (\lambda e \rightarrow Vec (interp (Var VZ) e) n)
    toVec
    fromVec))

Lists are somewhat trickier to represent. The type of a list does not tell us its length, so the direct recursive representation of List is an infinite structure. Agda’s termination checker will be unable to prove that uses of this representation terminate. Another option is to encode dependent pairs of lists and proofs they have finite lengths, rather than encoding lists directly. Examples of both these encodings are included with our source code.

4.2 Adding Data Support to ngen

Although Data provides a mechanism for coding datatypes, we cannot use it to define generic functions until we extend ngen to handle Data. However, there is a complication—it is not clear how to do so. Without getting too much into the technicalities, the issue is that in this definition we need to produce a result of type^3

b (repeat (DT-s dt) \circ envs)

but we only have a value of type

b (repeat (interp (DT-t dt)) \circ envs)

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

ngen : \{ n : N \} \{ b : Vec Set (suc n) \rightarrow Set \} \{ k : Kind \} \rightarrow (t : Ty k) \rightarrow TyConstEnv n b \rightarrow DataGen b
ngen \{ n \} \{ b \} = (b \{ k \} (repeat [ t ]))

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

DataGen : \{ n : N \} \rightarrow (b : Vec Set (suc n) \rightarrow Set) \rightarrow Set
DataGen \{ n \} b =
  \{ G : Ctx \} \rightarrow (dt : DT G)
  \rightarrow (envs : Vec (Env G) (suc n))
  \rightarrow (t : Ty k)
  \rightarrow TyConstEnv n b
  \rightarrow DataGen b
  \rightarrow b \{ k \} (repeat [ t ])

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

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

The definition of the DataGen coercion for the case where b is NGmap, called ngmap-data below, proceeds by induction on

^3 Here b : Vec Set (suc n) \rightarrow Set describes the type of the generic operation and envs : Vec (Env G) (suc n) is a vector of environments for the free variables.
the arity. In the base case of \( n=0 \), \texttt{ngmap-data} must coerce a result from the representation type to the Agda type using the to component.

For higher \( n \), \texttt{ngmap-data} is provided with a vector of environments \( e_1::es \) and a function of type:

\[
\text{interp} \ (\text{DT-t dt}) \ e_1 \\
\rightarrow \text{arrTy} \ (\text{repeat} \ (\text{interp} \ (\text{DT-t dt})) \ @ \ es)
\]

Its result type is:

\[
(\text{DT-s dt}) \ e_1 \rightarrow \text{arrTy} \ (\text{repeat} \ (\text{DT-s dt}) \ @ \ es)
\]

This case takes in a \((\text{DT-s dt}) \ e_1\) and uses the from function to convert it to an \((\text{DT-t dt}) \ e_1\), then coerces the result of the provided function by calling \texttt{ngmap-data} recursively.

\[
\text{ngmap-data} \ : \ \{n : \mathbb{N}\} \rightarrow \text{DataGen} \ (\text{NGmap} \ \{n\})
\]

\[
\text{ngmap-data} \ \{0\} \ n \ e = (\text{dt} \ (e :: []) \ (\text{bt} \ = \ \text{DT-to dt bt})
\]

\[
\text{ngmap-data} \ \{\text{n} \ \text{n} \ \text{n}\} \ n \ e = \lambda x \rightarrow \text{ngmap-data} \ \{\text{n}\} \ n \ e \ (\text{bt} \ = \ (\text{DT-from dt x}))
\]

\subsection{Using \texttt{ngen} at Datatypes}

With the \texttt{ngmap-data} function from the previous section, we may instantiate the updated \texttt{ngen} for \texttt{NGmap}.

\[
\text{ngmap} \ : \ (n : \mathbb{N}) \rightarrow (k : \text{Kind}) \rightarrow (e : \text{Typ} \ [] \ k) \rightarrow \text{NGmap} \ \{n\} \ (k) \ (\text{repeat} \ (\text{interp} \ e []))
\]

\[
\text{ngmap} n \ e = \text{ngen e ngmap-const ngmap-data}
\]

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

\[
\text{maybe-map1} :: [A : \text{Set}] \rightarrow (A \rightarrow B) \\
\rightarrow \text{Maybe} A \rightarrow \text{Maybe} B
\]

\[
\text{maybe-map1} = \text{ngmap} \ \text{maybe}
\]

\[
\text{vec-map0} :: [A : \text{Set}] \ (n : \mathbb{N}) \rightarrow A \rightarrow \text{Vec} A n
\]

\[
\text{vec-map0} = \text{ngmap} \ \text{vec}
\]

\[
\text{vec-map1} :: [A : \text{Set}] \ (n : \mathbb{N}) \\
\rightarrow (A \rightarrow B) \rightarrow \text{Vec} A n \rightarrow \text{Vec} B n
\]

\[
\text{vec-map1} = \text{ngmap} \ \text{vec}
\]

Observe that instantiating \texttt{ngmap} at a datatype is no different than any other type we have seen. The codes for \texttt{Maybe} and \texttt{Vec} work for any generic operation. Although the definition of \texttt{ngmap} needed the \texttt{DataGen} argument, this argument must be implemented once per generic operation, just like \texttt{TyConstEnv}. In contrast, previous work \cite{26} could not define a general code for datatypes like \texttt{Maybe} and \texttt{Vec}, and required significant boilerplate at every instantiation of a generic function with a specific datatype.

\section{Other Doubly-Generic Operations}

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

\subsection{Equality}

We saw in Section 3.3 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:

\[
\text{nvec-eq} :: \{m : \mathbb{N}\} \ A \rightarrow \text{Bool}
\]

\[
\text{nvec-eq} \ f v_1 \ldots v_n = \text{all} (\text{repeat} f \ (\ldots v_n))
\]

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-two equality for natural numbers returns true only if all three match:

\[
\text{nat-eq2} :: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Bool}
\]

Likewise, the arity-two 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.

\[
\text{pair-eq2} :: [(A_1 \times \mathbb{N}) \ C_1 \times \mathbb{N} \ C_2 : \text{Set}] \\
\rightarrow (A_1 \rightarrow (B_1 \rightarrow C_1 \rightarrow \text{Bool}))
\]

\[
\rightarrow (A_2 \rightarrow (B_2 \rightarrow C_2 \rightarrow \text{Bool})) \\
\rightarrow (A_1 \times A_2) \rightarrow (B_1 \times B_2) \rightarrow (C_1 \times C_2) \rightarrow \text{Bool}
\]

The definition of \texttt{ngeq}, which can define all of these operations, is similar to that of \texttt{ngmap}, so we will only highlight the differences.\footnote{The complete definition may be found in \texttt{ngeq.agda} with our sources.} One occurs in the definition of the arity-indexed type, \texttt{NGeq}. This function returns a boolean value rather than one of the provided types, which means that \texttt{ngeq} makes sense even for \( n = 0 \). In that case its type is simply \texttt{Bool}.

\[
\text{NGeq} :: \{n : \mathbb{N}\} \rightarrow (\text{Vec Set n}) \rightarrow \text{Set}
\]

\[
\text{NGeq} \ \{\text{zero}\} \ [] = \text{Bool}
\]

\[
\text{NGeq} \ \{\text{Suc n}\} (A_1 : \text{As}) = A_1 \rightarrow \text{NGeq} \ As
\]

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

\[
\text{defUnit} :: (n : \mathbb{N}) \rightarrow \text{NGeq} \ (\text{repeat} \ n)
\]

\[
\text{defUnit zero} = \lambda x \rightarrow \text{true}
\]

\[
\text{defUnit suc n} = \lambda x \rightarrow \text{defUnit n}
\]

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

\[
\text{constFalse} :: \{n : \mathbb{N}\} \rightarrow (\text{Vec Set n}) \rightarrow \text{NGeq v}
\]

\[
\text{constFalse} \ \{\text{zero}\} [] = \text{false}
\]

\[
\text{constFalse} \ \{\text{Suc m}\} (\text{A1} : \text{As}) = \lambda a \rightarrow \text{constFalse As}
\]

\[
\text{defNat} :: (n : \mathbb{N}) \rightarrow \text{NGeq} \ (\text{repeat} \ n)
\]

\[
\text{defNat zero} = \text{true}
\]

\[
\text{defNat suc zero} = \lambda x \rightarrow \text{true}
\]

\[
\text{defNat suc suc n} = \lambda x \rightarrow \lambda y \rightarrow \text{if EqNat x y then defNat suc n} y \rightarrow \text{else constFalse} \ (\text{repeat} \ n)
\]
\[ \lambda s \to bt \ (DT-from \ dt \ s) \]
\[ ngeq-data \ (\{ n : \text{Nat} \}) \ (e :: es) \ bt = \]
\[ \lambda s \to ngeq-data \ dt \ es \ (bt \ (DT-from \ dt \ s)) \]

With these pieces defined, the definition of ngeq is a straightforward application of ngen:
\[ ngeq \ : \ (n : \text{Nat}) \to \{ k : \text{Kind} \} \to (e : \text{Ty} \ k) \to \]
\[ \text{NGeq} \ (k) \ (\text{repeat} \ \{ \text{Suc} \ n \} \ (\{ e \})) \]
\[ ngeq \ n \ e = \ ngen \ e \ ngeq-const \ ngeq-data \]

5.2 Splitting

The Haskell prelude and standard library include the functions

\[ \text{unzip} :: \]
\[ \rightarrow \]
\[ \to \rightarrow \]
\[ \rightarrow \to \]
\[ \rightarrow \rightarrow \]
\[ \rightarrow \rightarrow \]
\[ \rightarrow \rightarrow \]
\[ \rightarrow \rightarrow \]

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 Maybe or Trees.

\[ \text{unzipMaybe} :: \text{Maybe} \ (a,b) \to (\text{Maybe} \ a, \text{Maybe} \ b) \]
\[ \text{unzipTree} :: \text{Tree} \ (a,b) \to (\text{Tree} \ a, \text{Tree} \ b) \]

Indeed, unzip is also datatype-generic, and Generic Haskell includes the function gunzipWith that can generate arity-one unzips for any type constructor.

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

For example, here is an instance of ngsplit, specialized to the Option type and arity 2. Note that this function is more general than unzipMaybe above, the Maybe need not contain pairs so long as we have some way to split the data.

\[ \text{unzipWithMaybe2} :: \{ A \ B \ C : \text{Set} \} \to (A \to B \times C) \]
\[ \to (\text{Maybe} \ A \to \text{Maybe} \ B \times \text{Maybe} \ C) \]
\[ \text{unzipWithMaybe2} = \text{ngsplit} \ 2 \text{ maybe} \]

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

\[ \text{unzipMaybe2} :: \{ A \ B : \text{Set} \} \to \text{Maybe} \ (A \times B) \]
\[ \to (\text{Maybe} \ A \times \text{Maybe} \ B) \]
\[ \text{unzipMaybe2} = \text{unzipWith2} \ (\lambda x \to x) \]

The function NGsplit gives the type of ngsplit at base kinds. The first type in the vector passed to NGsplit is the type to split. 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 \( \times \) constructor across a vector of types.

\[ \text{prodTy} : \{ n : \text{Nat} \} \to (\text{As} : \text{Vec} \text{ Set} \ n) \to \text{Set} \]
\[ \text{prodTy} \ \{ 0 \} \quad \text{=} \quad \top \]
\[ \text{prodTy} \ \{ 1 \} \quad \text{=} \quad (\text{A :: []}) \quad \text{=} \quad A \]
\[ \text{prodTy} \ \{ \text{Suc} \ (\text{Suc} \ n) \} \quad (\text{A :: As}) \quad \text{=} \quad (A \times \text{prodTy} \ As) \]

\[ \text{NGsplit} : \{ n : \text{Nat} \} \to (v : \text{Vec} \text{ Set} (\text{Suc} \ n)) \to \text{Set} \]
\[ \text{NGsplit} \ (\text{A} :: \text{As}) \quad = \quad \text{A1} \to \text{prodTy} \ As \]

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

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

\[ \text{prodn} : \{ n : \text{Nat} \} \to (\text{As} : \text{Vec} \text{ Set} \ n) \]
\[ 
\rightarrow \text{prodTy} \ As \to \text{prodTy} \ Bs \]
\[ 
\rightarrow \text{prodTy} \ (\text{repeat}_n \ (\times) \ As \times Bs) \]
\[ \text{prodn} \ \{ 0 \} \quad = \quad \bot \quad a b = \quad tt \]
\[ \text{prodn} \ \{ 1 \} \ (A :: []) \ (B :: []) \ a b = \quad (a, b) \]
\[ \rightarrow \text{prodTy} \ (\text{repeat}_n \ (\times) \ As \times Bs) \]
\[ \text{defPair} : \{ n : \text{Nat} \} \]
\[ 
\rightarrow (\text{As} : \text{Vec} \text{ Set} (\text{Suc} \ n)) \to (\text{NGsplit} \ As) \]
\[ 
\rightarrow (\text{Bs} : \text{Vec} \text{ Set} (\text{Suc} \ n)) \to (\text{NGsplit} \ Bs) \]
\[ 
\rightarrow \text{NGsplit} \ (\text{repeat}_n \ (\times) \ As \times Bs) \]
\[ \text{defPair} \ n \ (A :: As) \ (B :: Bs) \ b = \]
\[ \lambda p \to \text{prodn} \ n \ (\lambda (a \ (\text{proj1} \ p)) \ (b \ \text{proj2} \ p)) \]

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

\[ \text{defSum} : \{ n : \text{Nat} \} \]
\[ 
\rightarrow (\text{As} : \text{Vec} \text{ Set} (\text{Suc} \ n)) \to (\text{NGsplit} \ As) \]
\[ 
\rightarrow (\text{Bs} : \text{Vec} \text{ Set} (\text{Suc} \ n)) \to (\text{NGsplit} \ Bs) \]
\[ 
\rightarrow \text{NGsplit} \ (\text{repeat}_n \ (\times) \ As \times Bs) \]
\[ \text{defSum} \ 0 \ (A :: []) \ (\text{af} \ (B :: [])) \ b f = \lambda _ \to \text{tt} \]
\[ \text{defSum} \ (\text{Suc} \ n) \ (A :: \text{As}) \ (\text{af} \ (B :: Bs)) \ b f = \]
\[ \text{where} \ f : A \times B \to \text{prodTy} \ (\text{repeat}_n \ (\times) \ As \times Bs) \]
\[ f \ (\text{inj1} \ x1) = \text{injLeft} \ n \ (\text{af} \ x1) \]
\[ f \ (\text{inj2} \ x1) = \text{injRight} \ n \ (\text{bf} \ x1) \]

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

Splitting is a good example of datatype-generic programming’s potential to save time and eliminate errors. Defining a separate instance of split for vectors is not simple. For example, we would need a function to transpose vectors of products, transforming \( \text{Vec} \ m \ (A1 \times A2 \times \ldots \times An) \) into \( \text{Vec} \ m \times \text{Vec} \ A2 \times \ldots \times \text{Vec} \ An \ m \). 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.

5.3 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. However, this work has given us new insight into what other doubly-generic functions might look like. Furthermore, though the 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.

6. Related Work

Only a few sources discuss arity-generic programming. Fridlender and Indrika [7] 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 mention a few other arity-generic programs: tauto which determines whether a boolean expression of n variables is a tautology, and variations on loth, curry and uncurry from the Haskell prelude. It is not clear whether any of these functions could be made datatype-generic. McBride [14] 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. [24] 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 [22] translates Fridlender and Indrika’s example to the Omega 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 Omega, the two ideas are not combined.

Several researchers have used dependent types (or their encodings) to implement Generic-Haskell-style datatype-genericity. In previous work, we encoded representations of types using Church encodings [28] and GADTs [29] and showed how to implement a number of datatype-generic operations such as map. Hinze [10], 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. [26]. 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. This work extends those developments by considering examples not possible in Generic Haskell and showing a technique for writing generic programs which work on source-language datatypes.

The idea of generic programming in dependent type theory via universes has seen much attention since it was originally proposed [13, 18]. While demonstrating a new form of doubly genericity, this paper covers only one part of what is possible in a dependently typed language. In particular, our codes do not extend to all inductive families and so we cannot represent all types that are available (see Benke et al. [2] and Morris et al. [17] for more expressive universes). A dependently-typed language also permits the definition of generic proofs about generic programs. Chlipala [4] uses this technique in the Coq proof assistant to generically define and prove substitution properties of programming languages. Verbruggen et al. [27] use Coq’s dependent types to develop a framework for proving properties about generic programs.

7. Discussion

Generic programming in a dependently-typed language As we mentioned in the introduction, there are several dependently-typed languages that we could have used for this development. We selected Agda because the focus of its design has been this sort of programming. Like Coq, Agda is a full-spectrum dependently typed language. That has allowed us the flexibility to use universes to directly implement generic programming. We had the full power of the computational language available to express the relationships between values and types. A phase-sensitive language, such as Omega 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 rigorous type inference, especially combined with pattern matching. Although Coq can also infer implicit arguments, if we had written the functions in Coq we would have had to add many more type annotations. Additionally, developing in Agda allowed us to deal with non-termination more conveniently—while Coq must be able to see that a definition terminates before moving on, Agda shows the user where it can not prove termination and allows other work to continue.

On the other hand, using Coq would have 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 typecheck the (elided) implementation of ngen. However, we did not see any need for tactics in any of the uses of ngen to define doubly-generic operations. More importantly, as discussed below, differences in the way Coq and Agda handle type levels forced us to use Agda’s --type-in-type flag to clarify the presentation.

Type levels in Agda Although we have hidden it, Agda actually has an infinite hierarchy of type levels. Set, also known as Set0, is the lowest level in the type hierarchy. Terms like Set0 and Set0 → Set0 have type Set1, which itself has type Set2, 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, so to demonstrate that we are not using it in an unsound way, we have also implemented a version of the code that may be compiled without the flag. That version can be found in the notypeintype subdirectory of our source tarball.

Three differences between Coq and Agda make this explicit version more complicated than the one presented here. First, Agda currently lacks universe polymorphism [9], a feature which allows definitions to work on multiple type levels. As a result, many of the data structures in this paper must actually be duplicated at the level of Set1, creating significant clutter. Second, since Set is not impredicative in Agda, many definitions that could live at the level of Set in Coq must be at the level of Set1 instead. Finally, because Set0 is not a subtype of Set1 in Agda, we found it necessary to explicitly lift types from Set0 to Set1.

Future work and Conclusions Because we are working in the flexible context of a dependently-typed programming language, our work here will allow us to adapt and extend orthogonal results in generic programming to this framework. For example, we would like to use Agda as a proof assistant to reason about the properties of the generic programs that we write. We would also like to extend our universe so that it may encode more of Agda’s type system, such as arbitrary indexed datatypes. Finally, we would like to gain more experience with doubly-generic programming by creating and analyzing additional examples.
In this paper, we have combined arity-generic and datatype-generic programming into a single framework. Crucially, this combination takes advantage of the natural role that arities play in the definition of kind-indexed types. This framework has provided us with new understanding of the definition and scope of doubly-generic programs.

Acknowledgments Thanks to Andres Löh and Tim Sheard for discussion, and to the anonymous reviewers for many helpful comments. This paper was generated from literate Agda sources using \texttt{1has2TeX}.

References


