# Generic Programming in $\Omega$mega.

Tim Sheard

Computer Science Department
Maseeh College of Engineering and Computer Science
Portland State University

## 1 Introduction

> "Generic programming is about making programs more adaptable by making them more general. Generic programs often embody non-traditional kinds of abstraction; ordinary programs are obtained from them by suitably instantiating their parameters. In contrast with normal programs, the parameters of a generic program are often quite rich in structure; for example they may be other programs, types or type constructors, class hierarchies, or even programming paradigms."

I wrote these words for the introduction of the first workshop on generic programming. They remain true today. But, in the intervening years, the languages in which we program have acquired ever richer abstraction mechanisms, and many non-traditional kinds of abstraction have become ordinary. This is certainly true of languages which support dependent types where "generic programs" often become ordinary programs. The language $\Omega$mega supports a variant of dependent types (where types may depend upon type indexes, rather than values), and the purpose of these notes is to explain how we can exploit these types to build generic programs in $\Omega$mega. The first point I wish to make, is that the richer the type system, the more generic the programs possible.

The second point I wish to make is that generic programs, while very general, tend to introduce a layer of interpretation, and may suffer from certain kinds of inefficiencies. The use of staging is a natural mechanism for removing these inefficiencies. I demonstrate that the use of index-dependent types is orthogonal to the use of staging[1], and in general the two techniques often complement each other.

### 1.1 Genericity and the Curry-Howard Isomorphism

$\Omega$mega's type system makes it possible to realize the *Curry-Howard Isomorphism*. This states that programs are proofs, *and* types are properties. When we write $prog_1 :: type_1$, we are stating *both* that $prog_1$ has type $type_1$, *and* $prog_1$ is a proof of the property $type_1$.

[1] Fully dependent types are also orthogonal to staging, but that isn't illustrated in this paper.

In $\Omega$mega, this duality is the basis of generic programming. $\Omega$mega's type system allows us to type many programs that are not typable in a language like Haskell. This allows $\Omega$mega to support a number of new programming patterns including a large class of programs, which in other languages, are considered generic. Generic programming is all about abstraction. We show how to use the rich structures available in $\Omega$mega to build unusual abstractions with all the properties of generic programs.

## 2 The Structure of $\Omega$mega

We have adopted the following structure for the $\Omega$mega language. $\Omega$mega is a language with an infinite hierarchy of computational levels: value, type, kind, sort, etc. Computation at the value level is performed by reduction, and is largely unconstrained. Computation at all higher levels is constrained in several ways. First, all "data" at the type level and above is inductively defined data (no floats or primitive data for example). Second, functions at the type level and above must be inductively sequential (see Appendix A). This is a constraint on the form of the definition, not on the expressiveness of the language.

Terms at each level are classified by terms at the next level. Thus values are classified by types, types are classified by kinds, kinds are classified by sorts, etc. Programmers are allowed to introduce new terms and functions at every level, but any particular program will have terms at only a finite number of levels. We illustrate the level hierarchy for the many of the examples given in this paper in Figure 1.

We maintain a strict phase distinction — the classification of a term at level $n$ cannot depend upon terms at lower levels. For example, no types can depend on values, and no kinds can depend on types. We formalize properties of programs by exploiting the Curry-Howard isomorphism. Terms at computational level $n$, are used as proofs about terms at level $n+1$. We use indexed types to maintain a strict and formal connection between the two levels, and singleton types to maintain the strict separation between values and types.

## 3 A simple example

To illustrate the hierarchy of computational levels we give the following two-level example which uses natural numbers as a type index to lists that record their length in their type.

First, we introduce tree-like data (the natural numbers, `Nat`) at the *type level* by using the `data` introduction form. This form is a generalization over the `data` declaration in Haskell [15].

```
data Nat :: *1 where
  Z :: Nat
  S :: Nat ~> Nat
```

The line "`data Nat :: *1 where`" indicates that `Nat` is classified by `*1` (rather than `*0`), which tells the programmer that `Nat` is a kind (rather than a type), and that `Z` and `S` are types (rather than values). The classifiers `*0`, `*1`, `*2`, etc. indicate the level of a term. All values are classified by types that are classified by `*0`. All types are classified by kinds that are classified by `*1`. All kinds are classified by sorts that are classified by `*2`, etc. This is illustrated with great detail in Figure 1.

Second, we write a function at the type level over this data (`plus`). At the type level and higher, we distinguish function application from constructor application by surrounding function application by braces (`{` and `}`). For example, we write `S x` for constructor application, and `{plus x y}` for function application.

```
plus:: Nat ~> Nat ~> Nat
{plus Z m} = m
{plus (S n) m} = S {plus n m}
```

Third, using the `data` introduction form at the *value level*, we introduce the algebraic data structure (`Seq`). The types of such values are indexed by the natural numbers. These indexes describe an invariant about the constructed values — their length appears in their type — consider the type of `l1`.

```
data Seq:: *0 ~> Nat ~> *0 where
  Snil :: Seq a Z
  Scons:: a -> Seq a n -> Seq a (S n)

l1 = (Scons 3 (Scons 5 Snil)) :: Seq Int (S(S Z))
```

Finally, we introduce a function at the value level over `Seq` values (`app`). The type of `app` describes one of its important properties — there is a functional relationship between the lengths of its two inputs, and the length of its output.

```
app:: Seq a n -> Seq a m -> Seq a {plus n m}
app Snil ys = ys
app (Scons x xs) ys = Scons x (app xs ys)
```

To see that the `app` is well typed, the type checker does the following. The expected type is the type given in the function prototype. We compute the type of both the left- and right-hand-side of the equation defining a clause. We compare the expected type with the computed type for both the left- and right-hand-sides. This comparison generates some necessary equalities (for each side) to make the expected and computed types equal. We assume the left-hand-side equalities to prove the right-hand-side equalities. To see this in action, consider the second clause of the definition of `app`.

| | | | | |
|---|---|---|---|---|
| expected type | `Seq a n` | $\rightarrow$ `Seq a m` | $\rightarrow$ `Seq a {plus n m}` | |
| equation | `app (Scons x xs)` | `ys` | `= Scons x (app xs ys)` | |
| computed type | `Seq a (S b)` | $\rightarrow$ `Seq a m` | $\rightarrow$ `Seq a (S {plus b m})` | |
| equalities | | `n = (S b)` | $\Rightarrow$ `{plus n m}= S({plus b m})` | |

The left-hand-side equalities let us assume `n` = `S b`. The right-hand-side equalities, require us to establish that `{plus n m}` = `S{plus b m}`. Using the assumption that `n` = `S b`, we are left with the requirement that `{plus (S b) m}` = `S{plus b m}`, which is easy to prove using the definition of `plus`.

The different levels of the objects introduced in this example (and elsewhere in the paper) are plotted in Figure 1. The reader may wish to consult the figure to help visualize the relationships involved.

### 3.1 Overview

The following sections tell a rather complex story that will culminate with some generic programs written as ordinary programs in Ωmega. To reach that end, we introduce a few tools. The tools include some backward compatible additions to Haskell, a number of interesting programming patterns that exploit the additions, and a set of illustrative examples. The new features we have added to Haskell that are discussed in this paper include:

– **Data Structures at All Levels.** Kinds are a type system for classifying types. Sorts are a type system for classifying kinds. There is no practical limit to this hierarchy. In Ωmega, programmers can introduce new tree-like structures at any level. In Haskell all introduced datatypes are classified by `*0`. I.e. the introduced types classify only values. In Figure 1, Haskell types are illustrated by `Tree`, which is a type constructor which classifies its constructor functions (`Fork`, `Node`, and `Tip`) which are values. In Ωmega, the `data` declaration is generalized to all levels.
– **GADTs.** Generalized Algebraic Datatypes allow constructor functions to have more general types than the types supported by the `data` declaration in Haskell. GADTs are important because the additional generality allows programmer to express properties of types as witness types, proof objects, or singleton types. GADTs are the machinery that support the Curry-Howard isomorphism in Ωmega. In Figure 1, the types `Seq`, `Degree`, `LE` and `Even` require the generality introduced by GADTs.
– **Functions at All Levels.** Ωmega supports functions over the tree-like structures at all levels. Such functions are written by pattern matching equations, in much the same manner one writes functions over data at the value level in Haskell. We restrict

```
<- value name space | type name space ->
value             | type                                   | kind                     | sort
                  | Tree                                   :: *0 ~> *0                :: *1
Fork             :: Tree a -> Tree a -> Tree a             :: *0                      :: *1
Node             :: a -> Tree a                            :: *0                      :: *1
Tip              :: Tree a                                 :: *0                      :: *1
                  | Z                                      :: Nat                     :: *1
                  | S                                      :: Nat ~> Nat              :: *1
                  | plus                                   :: Nat ~> Nat ~> Nat       :: *1
                  | {plus #1 #3 }                          :: Nat                     :: *1
                  | Seq                                    :: *0 ~> Nat ~> *0         :: *1
Snil             :: Seq a Z                                :: *0                      :: *1
Scons            :: a -> Seq a b -> Seq a (S b)            :: *0                      :: *1
app              :: Seq a n -> Seq a m -> Seq a {plus n m} :: *0                      :: *1
                  | Fahrenheit                             :: TempUnit                :: *1
                  | Celsius                                :: TempUnit                :: *1
                  | Kelvin                                 :: TempUnit                :: *1
                  | Degree                                 :: TempUnit ~> *0          :: *1
F                :: Float -> Degree Fahrenheit             :: *0                      :: *1
C                :: Float -> Degree Celsius                :: *0                      :: *1
K                :: Float -> Degree Kelvin                 :: *0                      :: *1
plusK            :: Degree a -> Degree b -> Degree Kelvin  :: *0                      :: *1
                  | T                                      :: Boolean                 :: *1
                  | F                                      :: Boolean                 :: *1
                  | le                                     :: Nat ~> Nat  > Boolean   :: *1
                  | {le #0 #2}                             :: Boolean                 :: *1
                  | LE                                     :: Nat ~> Nat  > *0        :: *1
LeZ              :: LE Z a                                 :: *0                      :: *1
LeS              :: LE n m -> LE (S n) (S m)               :: *0                      :: *1
                  | Even                                   :: Nat ~> *0               :: *1
EvenZ            :: Even Z                                 :: *0                      :: *1
EvenSS           :: Even n -> Even (S(S n))                :: *0                      :: *1
```

**Fig. 1.** The level hierarchy for some of the examples in the paper.

the form of such definitions to be inductively sequential (See Appendix A). This ensures a sound and complete strategy for answering certain type-checking time questions by the use of narrowing. The class of inductively sequential functions is a large one, in fact every Haskell function has an inductively sequential definition. The inductively sequential restriction affects the form of the equations, and not the functions that can be expressed. In Figure 1, `plus` and `le` are functions at the type level.

– **Code Constructing Quasi-Quotes.** $\Omega$mega supports the run-time generation of code, along the lines of MetaML [18] and Template Haskell [19]. The meta-programming ability of code generation allows us to remove a layer of interpretation from our generic programs, that makes them efficient as well as general.

Some of the following sections are labeled with *Feature* if they are an addition to Haskell, *Pattern* if they are a paradigmatic use of the features to accomplish a particular end, or *Example* if they illustrate an important concept.

### 3.2 Relation to other systems

In order to make $\Omega$mega accessible to as broad an audience as possible, it is built around a framework which appears to the user to be a pure but strict version of Haskell. $\Omega$mega was designed, first and foremost, to be to be a programming language. Our goal was to design a language where program specifications, program properties, program analyses, proofs about programs, and programs themselves, are all represented using a single unifying notion of term. Thus programmers communicate many different things using the same language.

Our second goal was to make $\Omega$mega a logic, in which our reasoning would be sound. This is the basis of our decision to make $\Omega$mega strict. We made this design decision because the use of GADTs as proof objects requires that bottom not be an inhabitant of certain types. Strictness is part of our eventual strategy to accomplish that goal. This goal is not yet achieved.

There are many systems where soundness was the principal goal, and has been achieved. All of the examples, except for the staged examples, could be done in these languages as well. Such systems were principally designed to be logical frameworks or theorem provers. These include Inductive Families [7, 9], theorem provers (Coq [23], Isabelle [14]), logical frameworks (Twelf [16], LEGO [11]), and proof assistants (ALF [13], Agda [6]). Recently, there has been much interest in systems that use dependent types to build "practical" systems that are part language, part reasoning system. These systems include Augustsson's Cayenne language [3, 2], McBride's Epigram [12], Stump's Rogue-Sigma-Pi [21, 24], Xi and Pfenning's Dependent ML [27, 8], and Xi's Applied Type Systems [26, 5]. In fact, we owe a large debt to all these systems for inspiration.

We realize that just *a little* loss in soundness makes all our reasoning claims vacuous, but we are working to fill these gaps. Our goal is to do this in a different manner than the system listed above, which require all functions to be total in order to ensure soundness. We wish to use types to separate terminating functions from non-terminating functions. And make logical claims only about the terminating fragment of the language. This seems almost a necessary condition for a system that claims to be a programming language. In any case, these issues have little effect on our use of $\Omega$mega to program generic programs, since logical soundness is not an issue in this domain.

## 4 Introduction to $\Omega$mega

Throughout this section we introduce the features of $\Omega$mega by comparing and contrasting them with the features of Haskell. We assume a basic understanding of Haskell programs. In particular, we assume the reader understands about the introduction of algebraic data types with the `data` declaration (which introduces a type constructor and some constructor constants and functions), and the use of writing pattern matching equations to define functions.

**Feature: Kinds.** We can introduce new tree-like data at any level, including the type level and higher. The data declaration introduces both the constructors for tree-like data and the object that classifies these structures. We indicate the level where these objects reside using `*0`, `*1`, `*2`, etc. in the `data` declaration. Consider the kinds `Nat` (introduced earlier), `Boolean`, and `TempUnit`:

```
data Nat :: *1 where
  Z :: Nat
  S :: Nat ~> Nat

data Boolean:: *1 where
  T:: Boolean
  F:: Boolean
```

```
data TempUnit:: *1 where
  Fahrenheit:: TempUnit
  Celsius:: TempUnit
  Kelvin:: TempUnit
```

The new tree-like data at the type level are constructed by the type-constants (`Z`, `T`, `F`, `Fahrenheit`, `Celsius`, and `Kelvin`), and type constructors (`S`). The kinds `Nat`, `Boolean`, and `TempUnit` classify these structures, as shown explicitly in the declaration. For example `Celsius` is classified by `TempUnit`, and `S` is a constructor from `Nat` to `Nat`. Think of the operator `~>` as an function arrow at the type level. Note that while `Z`, `T`, `F`, `Fahrenheit`, `Celsius`, and `Kelvin` live at the type level, there are no values classified by them. Again, see Figure 1 to see where these objects reside in the larger picture.

Even though there are no values classified by these types, they are very useful. Instead of using them to classify values, we use them as indexes to value level data, i.e. types like `Degree Celsius` and `Seq a (S Z)`. The indexes like `Celsius` and `S z` indicate static (type-checking time) properties of values. For example, a value with type `Seq a (S Z)` is statically guaranteed to have length 1.

**Feature: GADTs.** Generalized Algebraic Datatypes allow constructor functions to have more general types than the types supported by `data` declaration in Haskell. GADTs are important because the additional generality allows programmer to express properties of types using type indexes and witnesses (or proof) objects. The `data` declaration in $\Omega$mega defines generalized algebraic datatypes (GADT). These are characterized by explicitly classifying constructors in a `data` declaration with their full types. The additional generality arises because the range of a constructor in a GADT is not constrained to be the type constructor applied to only type variables. For example consider the value level types:

```
data Seq:: *0 ~> Nat ~> *0 where
  Snil :: Seq a Z
  Scons:: a -> Seq a n -> Seq a (S n)

data Degree:: TempUnit ~> *0 where
  F :: Float -> Degree Fahrenheit
  C :: Float -> Degree Celsius
  K :: Float -> Degree Kelvin
```

Note, that instead of ranges like `Degree a` and `Seq a b`, where only type variables like `a` and `b` can be used as parameters, the ranges contain sophisticated instantiations such as `Degree Celsius` and `Seq a (S n)`. Note that the second index to `Seq` (the one of kind `Nat`) is used to describe an invariant about the length of the sequence. The constructors of `Degree` each lift a scalar `Float` to a `Degree` with a different unit. Operations on `Degree` are easy to define. Note that in $\Omega$mega, `(#+)` is the infix operator for addition on `Float`s.

```
add :: Degree a -> Degree a -> Degree a
add (F n) (F m) = F(n #+ n)
add (C n) (C m) = C(n #+ n)
add (K n) (K m) = K(n #+ n)
```

An interesting observation is that while the definition for `add` contains only three of the possible nine combinations of the constructors for `Degree`, it is total function. That is because any of the missing six patterns representing pairs of arguments, cannot both have the same `TempUnit` index: `a`, as declared by the prototype declaration.

Using kinds in this fashion is more expressive than just using phantom types. For example one might be tempted to use phantom types and write:

```
data Degree unit  = T Float
fah  :: Float -> Degree Fahrenheit
fah   = T
cel  :: Float -> Degree Celsius
cel   = T
kel  :: Float -> Degree Kelvin
kel   = T

add             :: Degree a -> Degree a -> Degree a
add (T x) (T y) = T (x #+ y) -- no need for 3 cases
```

This is strictly less general. First it admits nonsense types like `(Degree Bool)`. Using new kinds, only `Fahrenheit`, `Celsius`, and `Kelvin` are classified by `TempUnit`, so types like `Degree Bool` are rejected. The kind system plays the role of a *type system* for types.

Second, with the GADT approach, one can write functions that do different things depending on the type of their inputs. For example we can write coercing operators that take inputs of any units, but always return outputs of a standard unit (say `Kelvin`).

```
plusK :: Degree a -> Degree b -> Degree Kelvin
plusK (K x) (K y) = K(x #+ y)
plusK (C x) (K y) = K(273.0 #+ x #+ y)
```

For brevity, we have shown here only two of the nine possible cases.

**Feature: Type functions.** Kind declarations allow us to introduce new tree-like structures at the type level. We can use these structures to parameterize data at the value level as we did with `Nat` or `TempUnit`, or we can compute over these tree-like structures. Such functions are written by pattern matching equations, in much the same manner one writes functions over data at the value level. Several useful functions over types defined earlier are:

```
even :: Nat ~> Boolean
{even Z} = T
{even (S Z)} = F
{even (S (S n))} = {even n}

le:: Nat ~> Nat ~> Boolean
{le Z n} = T
{le (S n) Z} = F
{le (S n) (S m)} = {le n m}
```

```
plus:: Nat ~> Nat ~> Nat
{plus Z m} = m
{plus (S n) m} = S {plus n m}

and:: Boolean ~> Boolean ~> Boolean
{and T x} = x
{and F x} = F
```

Like functions at the value level, the type functions `plus`, `and`, `even`, and `le` are expressed using equations. The function `and` is a binary function that combines two `Booleans`. The property `even` is a unary predicate that distinguishes odd from even numbers, and the property `le` is a binary less-than-or-equal-to predicate. All the type functions are strict total (terminating) functions.

**Pattern: Witnesses.** GADTs can be used to witness relational properties between types. This is because the parameters to types introduced using the GADT mechanism can play different roles. The natural number argument of the type constructor `Seq` (from Section 3) plays a qualitatively different role than type arguments in ordinary ADTs. Consider the declaration for a binary tree datatype in Haskell:

```
data Tree a = Fork (Tree a) (Tree a) | Node a | Tip
```

In this declaration the type parameter `a` is used to indicate that there are subcomponents of `Tree`s that are of type `a`. In fact, `Tree`s are parametric. Any type of value can be placed in the "sub component" of type `a`. The type of the value placed there is reflected in the `Tree`'s type. Contrast this with the `n` in `(Seq a n)`. Instead, the parameter `n` is used to stand for an abstract property (the length of the list represented). When we use a type parameters in this way we call it a type index [25, 28] rather than a type parameter.

We can use indexes to GADTs to define value level data that we can think of as proofs, or witnesses to type level properties. This is a powerful idea. Consider the introduction of two new parameterized types `Even` and `LE`. Note that these are ordinary data structures that exist at the value level, but describe properties at the type level.

```
data Even:: Nat ~> *0 where
  EvenZ:: Even Z
  EvenSS:: Even n -> Even (S (S n))

data LE:: Nat ~> Nat ~> *0 where
  LeZ:: LE Z n
  LeS:: LE n m -> LE (S n) (S m)
```

```
data Plus:: Nat ~> Nat ~> Nat ~> *0 where
  PlusZ:: Plus Z m m
  PlusS:: Plus n m z -> Plus (S n) m (S z)
```

These declarations introduce value-level constants (`EvenZ`, `PlusZ`, and `LeZ`) and constructor functions (`EvenSS`, `PlusS`, and `LeS`). Let's examine the types of several values constructed with these constructors. To make it easier to handle types of kind `Nat`, in *Ω*mega, we have special syntactic sugar for entering and displaying them. `Z` = `#0`, `S Z` = `#1`, `S(S Z)` = `#2`, etc. We may also write `#(1+x)` for `S x`, and `#(2+x)` for `S(S x)`, etc.

```
EvenZ:: Even #0
(EvenSS EvenZ):: Even #2
(EvenSS (EvenSS EvenZ)):: Even #4

p1 ::Plus #2 #3 #5
p1 = PlusS (PlusS PlusZ)
```

```
LeZ:: LE #0 a
(LeS LeZ):: LE #1 #(1+a)
(LeS (LeS LeZ)):: LE #2 #(2+a)
```

We write `#0` for `Z`, and `#1` for `S Z`, etc. to emphasize that we should view `LE`, `Plus` and `Even` as a relationships between natural numbers. The important thing to notice is that we may view ordinary values with types `LE n m` and `Even n` as proofs, since the types of all legally constructed values witness only true statements about `n` and `m`. For example we cannot build a term of type `Even #1`. This is the essence of the Curry-Howard isomorphism. We can view `(EvenSS Even):: Even #2` as either the statement that `EvenSS EvenZ` has type `Even #2`, or that `EvenSS EvenZ` is a proof of the property `Even #2`. All this follows directly from the introduction of new types with tree-like structure, and the ability to *compute* over them.

**Pattern: Witness vs. Type Function.** The reader may have noticed that `even` and `Even` are two different ways to express the same notion. Either we write a (`Boolean`) function at the type level (`even`), or introduce a witness type (`Even`) at the value level. For every $n$-ary function at the type level, we can build an $(n+1)$-ary witness type. The witness type turns the $n$-ary function into an $(n+1)$-ary type constructor. Each clause in the function definition is named by a constructor function in the witness. If the right-hand-side of a clause has $m$ recursive calls, the constructor function becomes an $m$-ary constructor. The right-hand-side of each clause becomes the

$(n+1)^{th}$ argument of the range, where every recursive call to the function in the right-hand-side, is replaced with a variable. Each recursive call becomes one of the $m$ arguments. The $(n+1)^{st}$ argument to these calls is the new variable replacing the corresponding recursive call in the $(n+1)^{st}$ argument of the range. For example: The clause of the binary function `{plus (S n) m} = S {plus n m}`, becomes a ternary predicate `Plus (S n) m (S {plus n m})`. By replacing the recursive call with `z`, and making `z` be the $(n+1)^{st}$ parameter to the first argument, we get the type of the unary constructor `PlusS:: Plus n m z -> Plus (S n) m (S z)`.

Witnesses and type functions express the same ideas, but can be used in very different ways. Type functions are only useful at compile-time (they're static) and their structure cannot be observed (they can only be applied, so we say they are extensional). Witnesses, on the other hand, are actual data that is manipulated at run time (they're dynamic). Their structure can also be observed and taken apart (we say they're intensional). They are true data. A big difference between the two ways of representing properties is the computational mechanisms used to ensure that programs adhere to such properties.

**Pattern: Singleton Types.** Sometimes it is useful to direct computation at the type level, by writing functions at the value level. Even though types cannot depend on values directly, this can be simulated by the use of singleton types. The idea is to build a completely separate isomorphic copy of the type in the value world, but still retain a connection between the two isomorphic structures. This connection is maintained by indexing the value-world type with the corresponding type-world kind. This is best understood by example. Consider reflecting the kind `Nat` into the value-world by defining the type constructor `SNat` using a `data` declaration.

```
data SNat:: Nat ~> *0 where
  Zero:: SNat Z
  Succ:: SNat n -> SNat (S n)

three = (Succ (Succ (Succ Zero))):: SNat(S(S(S Z)))
```

Here, the value constructors of the `data` declaration for `SNat` mirror the type constructors in the `kind` declaration of `Nat`. We maintain the connection between the two isomorphic structures by the use of `SNat`'s natural number index. This type index is in one-to-one correspondence with the shape of the value. Thus, the type index of `SNat` exactly mirrors its shape. For example, consider the example `three` above, and pay particular attention to the structure of the type index, and the structure of the value with that type.

This kind of relationship between values and types is called a *singleton type* because there is only one element of any singleton type. For example only `Succ (Succ Zero)` inhabits the type `SNat(S (S Z))`. It is possible to define a singleton type for any first order type (of any kind). All Singleton types always have kinds of the form `I ~> *0` where `I` is the index we are reflecting into the value world. We sometimes call singleton types *representation types*. We cannot over emphasize the importance of the singleton property. Every singleton type completely characterizes the structure of its single inhabitant, and the structure of a value in a singleton type completely characterizes its type. Thus we can compute over a value of a singleton type, and the computation at the value level can express a property at the type level. By using singleton types we completely avoid the use of dependent types where types depend on values [20, 17]. The cost associated with this avoidance is the possible duplication of data structures and functions at several levels.

**Pattern: A pun: Nat'.** We now define the type `Nat'`, which is in all ways isomorphic to the type `SNat`. The type `Nat'` is also a singleton type representing the natural numbers, but it relies on an feature of the *Ω*mega type system. In *Ω*mega (as in Haskell) the name space for values is separate from the name space for types. Thus it is possible to have the same name stand for two things. One in the value space, and the other in the type space. The pun is because we use the names `S` and `Z` in both the value and type name spaces. We exploit this ability by writing:

```
data Nat':: Nat ~> *0 where
  Z:: Nat' Z
  S:: Nat' n -> Nat' (S n)
```

The value constructors `Z:: Nat' Z` and `S:: Nat' n -> Nat' (S n)` are ordinary values whose types mention the type constructors they pun. The name space partition, and the relationship between `Nat` and `Nat'` is illustrated below.

| ← *value name space* | *type name space* → | | |
|---|---|---|---|
| *value* | *type* | *kind* | *sort* |
| | Z | :: Nat | :: *1 |
| | S | :: Nat ~> Nat | :: *1 |
| Z | :: Nat' Z | :: *0 | :: *1 |
| S | :: Nat' m -> Nat' (S m) | :: *0 | :: *1 |

In `Nat'`, the singleton relationship between a `Nat'` value and its type is emphasized even more strongly, as witnessed by the example `three'`.

```
three' = (S(S(S Z))):: Nat'(S(S(S Z)))
```

Here the shape of the value, and the type index appear isomorphic. We further exploit this pun, by extending the syntactic sugar for writing natural numbers at the type level (`#0`, `#1`, etc.) to their singleton types at the value level. Thus we may write `(#2:: Nat' #2)`.

**Pattern: Computing Programs and Properties Simultaneously.** We can write programs that compute an indexed value along with a witness that the value has some additional property. For example, when we add two static length lists, the resulting list has a length that is related to the lengths of the two input lists, and we can simultaneously produce a witness to this relationship.

```
data Plus:: Nat ~> Nat ~> Nat ~> *0 where
  PlusZ:: Plus Z m m
  PlusS:: Plus n m z -> Plus (S n) m (S z)

app1:: Seq a n -> Seq a m -> exists p . (Seq a p,Plus n m p)
app1 Snil ys = Ex(ys,PlusZ)
app1 (Scons x xs) ys = case (app1 xs ys) of { Ex(zs,p) ->  Ex(Scons x zs,PlusS p) }
```

The keyword `Ex` is the "pack" operator of Cardelli and Wegner [4]. Its use turns a normal type `(Seq a p,Plus n m p)` into an existential type `exists p.(Seq a p,Plus n m p)`. The $\Omega$mega compiler uses a bidirectional type checking algorithm to propagate the existential type in the signature inwards to the `Ex` tagged expressions. This allows it to abstract over the correct existentially quantified variables.

**Pattern: Staging.** $\Omega$mega supports the staging annotations: brackets: (`[| _ |]`), escape (`$( _ )`), and the two staging functions: `lift::(forall a . a -> Code a)` and `run::(forall a . (Code a) -> a)` for building and manipulating code. $\Omega$mega uses the Template Haskell [19] conventions for creating code. Brackets (`[| _ |]`) are a quasi-quotation mechanism, and escape (`$( _ )`) escapes from the effects of quasi-quotation. For example.

```
inc x = x + 1
c1a = [| 4 + 3 |]
c2a = [| \ x -> x + $c1a |]
c3 = [| let f x = y - 1 where y = 3 * x in f 4 + 3 |]
c4 = [| inc 3 |]
c5 = [| [| 3 |] |]
c6 = [| \ x -> x |]
```

In the examples above, `inc` is a normal function. The variable `c1a` names a piece of code with type `Code Int`. The variable `c2a` names a piece of code with type `Code(Int -> Int)`. It is constructed by splicing the code `c1a` into the body of the lambda abstraction. The variable `c3` names a piece of code with type `Code Int`. It illustrates the ability to define rich pieces of code with embedded `let` and `where` clauses. The variable `c4` names a piece of code with type `Code Int`. It illustrates that functions defined in earlier stages (`inc`) can be lifted (or embedded) in code. The variable `c5` names a piece of code with type `Code (Code Int)`. It illustrates that code can be nested.

The purpose of the staging mechanism is to have finer control over evaluation order, which is exactly what we want to do when removing the interpretive overhead of generic programming. $\Omega$mega supports many of the features of MetaML [18, 22].

# 5 Generic Programming

We now know enough about $\Omega$mega to write some generic functions. Generic programming is nothing more than abstracting over objects rich in structure. Usually this structure is some internal compiler structure not accessible to the programmer. Without some sort of reflection mechanism, the language cannot access this structure. Even with a reflection mechanism, the reflected structure is usually too complex to capture in a well-typed way. Instead an untyped abstraction of the internal structure is used. This often has no precise formal connection to real internal structure. In $\Omega$mega, with its rich types, including the use of type indexes, we can often remedy these objections.

## 5.1 Datatype Generic Programs

The classic generic programming example is datatype generic programming, where a single program suffices to define a function over many different datatypes. When a function is defined over a fixed set of types we call it overloaded, when it is defined over all types, we call it generic.

The key to overloaded programs, is defining the overloaded functions over the structure of types. In order to do this effectively, the values representing the structure of types should be intimately connected to the actual type structure. In $\Omega$mega we can use GADTs to build such a connection.

We will build a representation of types using GADTs and then generate several program families by abstracting over the type representation. Our example is extremely simple, but depends only on the use of GADTs. It makes no use of $\Omega$mega's other features. In the companion paper in this collection, *Generic Programming, Now!*, by Ralf Hinze and Andres Löh, there are many richer examples that follow exactly this paradigm, which could also be programmed in $\Omega$mega.

To build the connection between types, and their structure we use GADTs to define an indexed structure that represents $\Omega$mega types. This GADT will have a type index. This index is the type that being represented.

Our example will represent base types, products, and sum types, but we could easily extend it to other types. In $\Omega$mega the type `a+b` is like `Either a b` in Haskell. Vaules of sum types are created with the constructor functions `L:: a -> (a+b)` and `R:: b -> (a+b)`. Note also, that the type `Rep` is a singleton type.

```
data Rep:: *0 ~> *0 where
  Rint:: Rep Int
  Rchar:: Rep Char
  Runit:: Rep ()
  Rpair:: Rep a -> Rep b -> Rep (a,b)
  Rsum::  Rep a -> Rep b -> Rep (a+b)

t1 :: Rep ((Int,Char)+((),Int))
```

```
t1 = Rsum(Rpair Rint Rchar) (Rpair Runit Rint)
```

Note, in example `t1`, that the index `((Int,Char)+((),Int))` is isomorphic to the shape of its value `Rsum (Rpair Rint Rchar) (Rpair Runit Rint)`. Next, we define an overloaded function to sum up all the integer components of a rich structure with type: `Rep a -> a -> Int`.

```
sumR :: Rep a -> a -> Int
sumR Rint n = n
sumR (Rpair r s) (x,y) = sumR r x + sumR s y
sumR (Rsum r s) (L x) = sumR r x
sumR (Rsum r s) (R x) = sumR s x
sumR _ x = 0
```

As we interpret the shape of the `Rep`, we recursively descend over sums and pairs (summing the two components from each part of the pair), components of type `Int` are returned, and other base types contribute zero. The overhead of interpreting the type representation adds considerable cost to computing the result. Staging the sum function can remove this overhead, and is simply a matter of placing the staging annotations in the correct places.

```
sum2 :: Rep a -> Code a -> Code Int
sum2 Rint x = x
sum2 (Rpair x y) z =
  [| let (a,b) = $z in $(sum2 x [| a |]) + $(sum2 y [| b |]) |]
sum2 (Rsum r s) x =
  [| case $x of
       L m -> $(sum2 r [| m |])
       R m -> $(sum2 s [| m |]) |]
sum2 _ x = [| 0 |]

sumG:: Rep a -> Code (a -> Int)
sumG x = [| \ w -> $(sum2 x [| w |]) |]

ans3 = sumG t1
```

Note the use of the generated pattern match in the `Rpair` case to avoid the duplication of the code `z`.

The strategy we use in the example above is a common one when staging a function. Rather than define a function with type `Rep a -> Code (a -> Int)` in one go, we do it in two steps. First defining a function (`sum2`) with type `Rep a -> Code a -> Code Int`, and then using this function to build the function (`sumG`) with the type we really want. We do this because it is easier to manipulate code with first-order types. If we built `sumG` directly we would generate code with unnecessary explicit beta-redices and eta-expansions. The example `ans3` illustrates the residual code.

```
[| \ bbf ->
     case bbf of
       (L ebf) ->
         let (hbf, ibf) = ebf
         in hbf + 0
       (R jbf) ->
         let (mbf, nbf) = jbf
         in 0 + nbf |] : Code (((Int,Char)+((),Int)) -> Int)
```

Many richer examples of datatype generic polymorphism are illustrated in the companion paper *Generic Programming, Now!*. Such examples depend only upon the GADT mechanism and are programmed in the GHC extensions to Haskell. These examples could benefit additionally from staging. We now move to some examples that rely on the richer type mechanisms of $\Omega$mega.

### 5.2 The n-sum example

We will build a well-typed generic program that captures the family of programs:

```
1.  \ x -> x
2.  \ x -> \ y -> x+y
3.  \ x -> \ y -> \ z -> x+y+z
4.  \ x -> \ y -> \ z -> \ w -> x+y+z+w
```

First, we define a function at the type level that maps a natural number $i$ to the type of the $i^{th}$ function in this family.

```
sumTy :: Nat ~> *0
{sumTy Z} = Int
{sumTy (S n)} = Int -> {sumTy n}
```

For example, the type `{sumTy #3}` normalizes to the type `Int -> Int -> Int -> Int`. With this type function we can now capture the type of the generic sum over `n` integers: `nsum:: Nat' n -> {sumTy n}`. Which behaves as:

```
(nsum #1) =  \ x -> x
(nsum #2) =  \ x -> \ y -> x+y
(nsum #3) =  \ x -> \ y -> \ z -> x+y+z
```

We define the n-way sum function in two stages. We write a helper function `nsumAcc`, and then write a program `nsum` which captures the family of functions:

```
nsumAcc :: Nat' n -> Int -> {sumTy n}
nsumAcc Z x = x
nsumAcc (S n) x = \ y -> (nsumAcc n (x+y))

nsum :: Nat' n -> {sumTy n}
nsum n = nsumAcc n 0

testsum = (nsum #3 1 3 5) == (nsum #2 0 9)
```

Note how the number of integers `nsum` can consume depends upon its `Nat'` argument. It is truly generic in the sense that it abstracts over objects not in the vocabulary of most languages. Note also, that type checking this function must solve a narrowing problem.

| | |
|---|---|
| expected type | `Nat' n` $\longrightarrow$ `Int` $\longrightarrow$ `{sumTy n}` |
| equation | `nsumAcc (S n)` `x` = $\lambda$ `y` $\rightarrow$ `(nsumAcc n (x+y))` |
| computed type | `Nat' (S a)` $\longrightarrow$ `Int` $\longrightarrow$ `Int` $\rightarrow$ `{sumTy a}` |
| equalities | `n = (S a)` $\Rightarrow$ `Int` $\rightarrow$ `{sumTy a}` = `{sumTy n}` |

We must show that under the assumption `n = (S a)`, the type `Int -> {sumTy a}` is equal to the type `{sumTy n}`. Substituting for `n`, we must solve `Int -> {sumTy a} = {sumTy (S a)}`. Using the definition of `sumTy` we get `Int -> {sumTy a} = Int -> {sumTy a}`.

This program interprets the structure of its `Nat'` argument. It would be well to stage the function so that this interpretive over head is removed. This is easy to do using $\Omega$mega's staging annotations. To stage the `nsum` program we write the nsum generator `nsumG`.

```
nsumG :: Nat' n -> Code Int -> Code {sumTy n}
nsumG Z x = x
nsumG (S n) x = [| \ y -> $(nsumG n [| $x + y |]) |]

sumGen:: Nat' n -> Code {sumTy n}
sumGen n = nsumG n (lift 0)
```

This is essentially the `nsumAcc` program with a few brackets and escapes judicially sprinkled over the program, to force the interpretation to happen in the first stage, and to produce the residual code. By executing `sumGen` we obtain the residual code when the size is `#3`.

```
sumGen (S(S(S Z)))

[| \ y1 -> \ y3 -> \ y5 -> 0 + y1 + y3 + y5 |]
:: Code (Int -> Int -> Int -> Int)
```

**Analysis of the generic examples.** The process followed in this and the previous example is common to many generic programming exercises. We review it here.

- First, develop some indexed data structure to capture the details of the problem. The index captures a semantic property of the generic program. In the `nsum` example the indexed data structure is the singleton natural numbers, and the index is related to the number of arguments. In the datatype generic example, the index relates the structure (`Rep`) to the type it represents.
- Second, write a type function(s) that computes the type of the generic result as a function of the specification structure. In the `nsum` example, `sumTy` plays this role. In the datatype generic sum function the type function was the identity and was not needed.

– Third, write a function that interprets the specification to produce the result (`nsum` and `sum2`).
– Fourth, stage the generic function to produce a generic generator (`nsumG` and `sumG`).

### 5.3 Generic N-way Zip

In this example we illustrate the use of a type relation (or witness type) to specify a generic function. Sometimes such a witness type is more useful than a type function.

Consider the zip function that maps a ternary function over the elements of three lists. One way to approach this problem was suggested by Fridlender and Indrika in their paper *Do We Need Dependent Types* [10]. They develop an $n$-way zip that uses only polymorphism rather than indexed or dependent types. Their approach depends upon the following observation, that $n$-way zip is nothing more than repeated list-based application(`apply`), where a list of functions is applied to a list of arguments. We illustrate for $n = 3$.

```
zip3 f [1,2,3] [True,False] ['a','b','c','d']                                 -->
[f,f,f,f, ...] `apply` [1,2,3] `apply` [True,False] `apply` ['a','b','c','d']  -->
[f 1, f 2, f 3] `apply` [True,False] `apply` ['a','b','c','d']                -->
[f 1 True, f 2 False] `apply` ['a','b','c','d']                               -->
[f 1 True 'a', f 2 False 'b']

apply (f:fs) (x:xs) = (f x) : (apply fs xs)
apply _ _ = []
```

The function `zip3` first creates an infinite list of the function `f`, and then iterates the list-based application function over the three lists. Note how the length of the shortest of the three lists determines the length of the result. An infinite list of `fs` is necessary since the function `zip3` cannot know how long its inputs are without testing them. The approach is to build the iterator as higher-order ordinals, i.e. functions that behave like zero and successor functions that build the natural numbers.

```
zero:: a -> a
zero x = x

succ:: ([a] -> b) -> [c -> a] -> [c] -> b
succ n fs xs = n(fs `apply` xs)

zero:: a -> a
(succ zero):: [a -> b] -> [a] -> [b]
(succ(succ zero)):: [a -> b -> c] -> [a] -> [b] -> [c]
```

Note how successive application of `succ` to previous ordinals construct functions that use the list-based application strategy. The definition of `zip` is trivial, first building the infinite list of functions, and then applying the ordinal to do the work.

```
zip:: ([a] -> b) -> a -> b
zip n f = n (repeat f)
```

There are three problems with this approach. First, the ordinals are functions and not data structures. Because of this, their structure cannot be observed. This precludes building a staged implementation, which means there will always be a layer of interpretation that cannot be removed. Second, this strategy depends in a crucial way on the use of laziness. The infinite list of functions can only be built in a lazy language. In a strict language (like ML), the strategy must be severely modified to test for the maximum length of the $n$ lists. This also adds some unnecessary overhead. Third, the approach builds $n$ intermediate lists, none of which are absolutely necessary.

We will attack these problems one at a time. First, we can use GADTs to construct the ordinals. Using this approach the ordinals become data structures that can be observed. Consider the GADT `Zip` and the types of the first three ordinals:

```
data Zip:: *0 ~> *0 ~> *0 where
  Zero :: Zip [a] [a]
  Succ :: Zip [x] y -> Zip [a -> x] ([a] -> y)

zero :: Zip [a] [a]
zero = Zero

one :: Zip [a -> b] ([a] -> [b])
oned = Succ Zero

two:: Zip [a -> b -> c] ([a] -> [b] -> [c])
two = Succ(Succ Zero)
```

In general the type of the $n$ way map has the type `a -> b` where the $n$th ordinal has type `Zip [a] b`. We can write these functions following the pattern of Fridlender and Indrika.

```
zip :: Zip [x] y -> x -> y
zip n f = zip' n (repeat f)
  where zip' :: Zip x y -> x -> y
        zip' Zero xs = xs
        zip' (Succ n) fs = \ xs -> zip' n (apply fs xs)
```

First, create the infinite list of `fs`, and then repeatedly apply this list of functions to the next list of elements. Because the ordinal is a data structure we can easily stage this function.

```
zipGen :: Zip [x] y -> Code(x -> y)
zipGen n = [| \ f -> $(zip' n [| repeat f |]) |]
  where zip' :: Zip x y -> Code x -> Code y
        zip' Zero xs = xs
        zip' (Succ n) fs = [| \ xs -> $(zip' n [| apply $fs xs |] ) |]
```

```
zip:: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
zip f = \ list1 -> case list1 of
           (x : xs) ->
             \ list2 -> case list2 of
                (y : ys) ->
                   \ list3 -> case list3 of
                      (z : zs) -> f x y z : zip f xs ys zs
                      [] -> []
                [] -> \ _ -> []
           [] -> \ _ -> \ _ -> []
```

**Fig. 2.** An example $n$-way zip, for $n$=3.

Here the iteration over $n$ becomes plain. For each $n$ the function `zip'` unfolds generating a lambda expression over one of the $n$ lists. For example: `zipGen (Succ (Succ Zero))` generates the following $\Omega$mega code:

```
[| \ f -> \ xs -> \ ys -> %apply (%apply (%repeat f) xs) ys |]
```

Unfortunately, this code still suffers from the use of laziness, and the creation of intermediate lists. An alternate approach is illustrated in Figure 2. The 3-ary zip illustrated is an instance of a generic $n$-way zip. It should be clear that there are two iterations going on. The first iteration is over $n$. For $n$=3, we see that the function unfolds into 3 lambdas and 3 `case` expressions. The second iteration is over the length of the shortest of the $n$ lists. Note that this iteration begins only after the $n$ way iteration. Note how this strategy neither depends upon laziness, nor builds unnecessary intermediate lists.

To type the general $n$-way zip (rather than the specific 3-way zip) consider the sequence of terms that appear in the definition of the 3-way zip.

```
f        :: (a -> b -> c -> d)
(f x)    :: (b -> c -> d)
(f x y)  :: (c -> d)
(f x y z):: d

(zip f)         :: ([a] -> [b] -> [c] -> [d])
(zip f xs)      :: ([b] -> [c] -> [d])
(zip f xs ys)   :: ([c] -> [d])
(zip f xs ys zs):: [d]
```

There is a relationship between the types of the two sequences of terms. Note that the types of the second sequence can be obtained from the first by applying the list type constructor to each of the type variables in the types of the first sequence. This motivates the following witness type, that relates the number $n$, and the types of these two sequences.

```
data Zip' :: *0 ~> *0 ~> *0 where
  Zero':: Zip' a [a]
  Succ':: Zip' b c -> Zip' (a -> b) ([a] -> c)
```

Like the GADT `Zip`, the new values of the new GADT `Zip'` are isomorphic to the natural numbers, and `Zip'` is a relation between the two sequences of types given above. By observing the type of several successive values we can visualize this relationship.

```
zero' = Zero'                :: Zip' a [a]
one'  = Succ' Zero'          :: Zip' (a -> b) ([a] -> [b])
two'  = Succ' (Succ' Zero') :: Zip' (a -> b -> c) ([a] -> [b] -> [c])
```

It should be clear, that the type of the $n$-way zip can be captured by using either `Zip` or `Zip'`

`zip:: Zip [a] b -> a -> b`. and

`zip:: Zip' a b -> a -> b`.

To define `zip` using `Zip'`, we write helper function, `help :: Zip' a b -> a -> b -> b`. The basic idea is captured by the table below.

| | | | |
|---|---|---|---|
| 2 | `f:: (a -> b -> c)` | `zip f:: [a] -> [b] -> [c]` | `help two' f (zip f):: [a] -> [b] -> [c]` |
| 1 | `f x:: (b -> c)` | `zip f xs:: [b] -> [c]` | `help one' (f x) (zip f xs):: [b] -> [c]` |
| 0 | `f x y:: c` | `zip f xs ys:: [c]` | `help zero' (f x y) (zip f xs ys):: [c]` |

As $n$ decreases, the two arguments to `help` become more saturated, by applying them to the heads and tails of the n lists. We put this all together below:

```
zip n = let zip f =  help n f (\ x -> zip f x) in zip
 where default:: Zip' a b -> b
       default Zero' = []
       default (Succ' n) = \ ys -> default n

       help :: Zip' a b -> a -> b -> b
       help Zero' x xs = (x : xs)
       help (Succ' n) f rcall =
          (\ ys -> case ys of
               (z:zs) -> help n (f z) (rcall zs)
               _ -> default n)
```

There are two subtleties. The function `default` builds a lambda of n-arguments, that ignores all of them and returns the empty list. This is necessary since the early termination of the $m$th-list, means we must skip over the $(m+1)$st-list through $n$th-list, before we return the empty list. The second subtlety is the eta-expansion of `map f` as the third argument to help. Without this eta-expansion the definition of `map` would loop since $\Omega$mega is a strict language. It is easy to stage this function.

```
zipGen:: Zip' a b -> Code (a -> b)
zipGen n = [| let zip f =  $(zip' n [|f|] [|zip f|]) in map |]
 where default:: Zip' a b -> Code b
       default Zero' = [| [] |]
       default (Succ' n) = [| \ ys -> $(default n) |]

       zip' :: Zip' a b -> Code a -> Code b -> Code b
       zip' Zero' x xs = [| $x : $xs |]
       zip' (Succ' n) f rcall =
          [| \ ys -> case ys of
              (z:zs) -> $(zip' n [| $f z |] [| $rcall zs |])
              _ -> $(default n) |]
```

Because of the staging we no-longer need the eta-expansion of `map f`. We apply `mapGen` to a few ordinals to visualize the results. In the code below we have alpha-renamed some of the variables in the code that was actually generated to make the structure more obvious.

```
zipGen Zero'   ---->
[| let zip0 x = x : zip0 x in zip0 |] : forall a.Code (a -> [a])

zipGen  one'    ---->
[| let zip1 f =
        \ xs ->
             case xs of
               (y : ys) -> f y : zip1 f ys
               _ -> []
   in zip1 |] : forall a b.Code ((a -> b) -> [a] -> [b])


zipGen two'  ---->
[| let zip2 f =
        \ xs ->
             case xs of
               (e28 : f28) ->
                 \ ys ->
                      case ys of
                        (k28 : l28) -> f e28 k28 : zip2 f f28 l28
                        _ -> []
               _ -> \ n28 -> []
   in zip2 |] : forall a b c.Code ((a -> b -> c) -> [a] -> [b] -> [c])
```

For $n$=3, this is exactly the same function which we used as our starting point in Figure 2. Moreover, we have fixed all three of the problems associated with the list-based application approach, with code that is only marginally longer.

**Analysis of the witness based examples.** The witness based approach to generic programming is common. We review it here.

– First, develop some indexed data structure to witness the relationships between the input and output types of the generic function. The GADTs `Zip` and `Zip'` played this role.
– Second, write a function that interprets the witness type to produce the result (the two versions of `zip`).
– Third, stage the generic function to produce a generic generator (the two versions of `zipGen`).

### 5.4 Using $\Omega$mega's hierarchy of levels

Our last example illustrates the utility of abstracting over kinds and other higher level entities in addition to the normal ability of abstracting over types. In the companion paper *Generic Programming, Now!*, kind-indexed families of generic functions are illustrated. To enable this, separate representations of types for each kind must be defined. For example, a representation for types with kind: `*0` (`TypeS`), a representation for types with kind: `*0 ~> *0` (`TypeS_S`), and a representation for types with kind: `*0 ~> *0 ~> *0` (`TypeS_S_S`).

```
data TypeS:: *0 ~> *0  where
  Char:: TypeS Char
  Int:: TypeS Int
  AppS_S:: TypeS_S phi -> TypeS a -> TypeS (phi a)

data TypeS_S:: (*0 ~> *0) ~> *0  where
  List:: TypeS_S []
  Tree:: TypeS_S Tree
  AppSS_S:: TypeS_S_S phi -> TypeS a -> TypeS_S (phi a)

data TypeS_S_S:: (*0 ~> *0 ~> *0) ~> *0  where
  Pair:: TypeS_S_S (,)
```

An approach with a different representation for types, for types at every different kind, is a work-around at best. It requires multiple new datatypes, where logically only one is needed, and it requires duplicating the application type representation at every kind. While sufficient for the examples in the paper the three representation types are not complete. Type constructors with kind `(*0 ~> *0) ~> *0` and `*0 ~> *0 ~> *0 ~> *0`, are rare, but not unheard of, and each would require an additional representation type. In $\Omega$mega, only a single kind-indexed datatype is required.

```
data Type:: forall (k:: *2) (t::k) . k ~> t ~> *0  where
  Char:: Type *0 Char
  Int:: Type *0 Int
  App:: forall (i:: *1) (j:: *1) (f:: i~>j) (x:: i) .
         Type (i ~> j) f -> Type i x -> Type j (f x)
  List:: Type (*0 ~> *0) []
```

```
  Tree:: Type (*0 ~> *0) Tree
  Pair:: Type (*0 ~> *0 ~> *0) (,)
```

The representation type `Type k t` represents a type `t` classified by kind `k`. The type constructor `Type` is polymorphic-recursively kinded. It is applied at different kinds in its own definition.

To write a kind-indexed function, we define a type function that computes over kinds. For example to compute a generic `count` function we proceed as follows.

```
countTy:: forall (k:: *2) (t::k) . k ~> t ~> *0
{countTy *0 a} = a -> Int
{countTy (i ~> k) phi} = {countTy i a} -> {countTy k (phi a)}

count::  forall (k:: *1) (t::k) . Type k t -> {countTy k t}
```

Note that the range of `count` is a function of the kind that classifies `t`. This requires a single function, with one clause for each of the constructors of `Type`.

```
count::  forall (k:: *1) (t::k) . Type k t -> {countTy k t}
count Char = const 0
count Int = const 0
count (App f a) = (count f) (count a)     where theorem countTyInjective1
count List = \ c -> sumList . mapList c
count Tree = \ c -> sumTree . mapTree c
count Pair = \ c1 c2 (x,y) -> c1 x + c2 y
```

Typing this function requires a little bit of specialized knowledge. In order to type the `(App f a)` clause, we must prove that `{countTy _a e} -> {countTy _c (_d e)}` is equal to `{countTy _a _b} -> {countTy _c (_d _b)}`. I.e. we must find bindings for the existentially quantified variable `e` that make it true.

This can be proven by assigning the value `_b` to `e`, if we realize that `countTy` is injective in its first argument. I.e. `{countTy a x}={countTy a y}` if and only if `x=y`. This is the purpose of the theorem `countTyInjective1` in the where clause for the `(App f a)` case.

The theorem declaration is an important way of directing the type checker. It uses ordinary terms, with types that can be read logically, as specifcations for new type checking strategies. In $\Omega$mega, there are currently three types of theorems. Each is used to direct the type checker to do things it might not ordinarily do. In the case of `countTyInjective1` the type is:

```
countTyInjective1:: Kind' a -> Equal {countTy a x} {countTy a y} -> Equal x y
```

Logically, this can be read as, *If* `a` *is a well formed kind, and* `{countTy a x}` *is equal to* `{countTy a y}`*, then* `x` *is equal to* `y`. The theorem clause directs the type checker to try proving `x=y` as one step in a strategy for proving `{countTy a x}={countTy a y}`.

Such a theorem is proved by induction over the structure of the kind `k`. In $\Omega$mega, this manifests itself as recursive term, representing a proof by induction.

```
prop Kind':: *1 ~> *0 where
  Star :: Kind' *0
  Karr :: Kind' a -> Kind' b -> Kind' (a ~> b)

countTyInjective1:: Kind' a -> Equal {countTy a x} {countTy a y} -> Equal x y
countTyInjective1 Star Eq = Eq
countTyInjective1 (Karr x y) Eq = Eq
  where theorem ih1 = countTyInjective1 x,
                ih2 = countTyInjective1 y
```

Note, that the proof (i.e. the term that defines `countTyInjective1`), uses induction hypotheses, which manifest themselves as calls to `countTyInjective1` on smaller sub terms.

Finally we can define two generic functions `size` and `sum` for all types with kind `*0 ~> *0`.

```
size:: Type (*0 ~> *0) f -> f a -> Int
size f = count f (const 1)

sum:: Type (*0 ~> *0) f -> f Int -> Int
sum f = count f id
```

## 6 $\Omega$mega's approach to dependent types

In a dependently typed program, the type of some terms are dependent on the values of other terms. In the design of $\Omega$mega, we separate values from types, so in $\Omega$mega we use indexed types instead. For example, reconsider the `nsum` family of functions that sums 1, 2, 3, ... $n$ inputs from Section 5. We could write such function with a dependent type: `f :: Pi (n:Natural). Int -> {sumTy n}`

In our hypothetical dependently-typed language:

```
f :: Pi (n::Natural). Int -> {sumTy n}
f 0  x = x
f n x = \ y -> f (n-1) (x+y)
```

Note how the type of `f` depends on the value of `f`'s first parameter. This violates our design decision separating values and types. But by using singleton types in $\Omega$mega, we defined the `nsum` function from Section 5 with type `nsum :: Nat' n -> Int -> {sumTy n}`

Note how the range of `nsum` is not a function of the value of its first parameter. Instead it is a function of `nsum`'s first parameters' type index, `n`, which is still a type. This example illustrates our strategy for writing dependently typed functions. *Write functions whose types depend on the type-indexes of their arguments rather than the values of their arguments.*

### 6.1 Conclusion

$\Omega$mega is a language which supports a limited kind of dependent typing, and supports the types-as-properties paradigm. We separate values from types to maintain a familiar functional programming style. Singleton types and other indexed types bridge the gap created by the phase distinction.

The generic programs we have written are ordinary $\Omega$mega programs which abstract over rich values (like `Rep`, `Nat'`, `Zip`, `Zip'`, and `Type`). In addition, the overhead of interpretation of the generic specifications can be completely removed by staging.

## References


1. Sergio Antoy. Definitional trees. In *ALP*, pages 143–157, 1992.
2. Lennart Augustsson. Cayenne — a language with dependent types. *ACM SIGPLAN Notices*, 34(1):239–250, January 1999.
3. Lennart Augustsson. Equality proofs in Cayenne, July 11 2000. `http://www.cs.chalmers.se/~augustss/cayenne/eqproof.ps`.
4. Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.
5. Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *ICFP 2005*, 2005. `http://www.cs.bu.edu/~hwxi/`.
6. Catarina Coquand. Agda is a system for incrementally developing proofs and programs. Web page describing AGDA: `http://www.cs.chalmers.se/~catarina/agda/` .
7. T. Coquand and P. Dybjer. Inductive definitions and type theory: an introduction. (preliminary version). *Lecture Notes in Computer Science*, 880:60–76, 1994.
8. Rowan Davies. A refinement-type checker for Standard ML. In *International Conference on Algebraic Methodology and Software Technology*, volume 1349 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
9. P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. *Lecture Notes in Computer Science*, 1581:129–146, 1999.
10. Daniel Fridlender and Mia Indrika. Do we need dependent types? *J. Funct. Program*, 10(4):409–415, 2000.
11. Zhaohui Luo and Robert Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, LFCS, Computer Science Dept., University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, May 1992. Updated version.
12. Connor McBride. Epigram: Practical programming with dependent types. In *Notes from the 5th International Summer School on Advanced Functional Programming*, August 2004. Available at: `http://www.dur.ac.uk/CARG/epigram/epigram-afpnotes.pdf` .
13. Bengt Nordstrom. The ALF proof editor, March 20 1996. `ftp://ftp.cs.chalmers.se/pub/users/ilya/FMC/alfintro.ps.gz`.
14. Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.
15. Simon Peyton Jones. Special issue: Haskell 98 language and libraries. *Journal of Functional Programming*, 13, January 2003.
16. Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *LNAI*, pages 202–206, Berlin, July 7–10, 1999. Springer-Verlag.

17. Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. *ACM SIGPLAN Notices*, 37(1):217–232, January 2002.
18. T. Sheard. Using MetaML: A staged programming language. *Lecture Notes in Computer Science*, 1608:207–239, 1999.
19. T. Sheard and S. Peyton-Jones. Template meta-programming for Haskell. In *Proc. of the workshop on Haskell*, pages 1–16. ACM, 2002.
20. Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–227, Boston, Massachusetts, January 19–21, 2000.
21. Aaron Stump. Imperative LF meta-programming. In *Logical Frameworks and Meta-Languages workshop*, July 2004. Available at:
`http://cs-www.cs.yale.edu/homes/carsten/lfm04/`.
22. Walid Taha and Tim Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2), 2000.
23. The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 7.4*. INRIA, 2003. `http://pauillac.inria.fr/coq/doc/main.html`.
24. Edwin Westbrook, Aaron Stump, and Ian Wehrman. A language-based approach to functionally correct inperative programming. Technical report, Washington University in St. Louis, 2005. Available at:
`http://cl.cse.wustl.edu/`.
25. Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1997.
26. Hongwei Xi. Applied type systems (extended abstract). In *In post-workshop proceedings of TYPES 2003*, pages 394 – 408, 2004. Springer-Verlag LNCS 3085.
27. Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. *ACM SIGPLAN Notices*, 33(5):249–257, May 1998.
28. Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, ACM SIGPLAN Notices, pages 214–227, New York, NY, USA, 1999. ACM Press.

## A Inductively Sequential Functions

We restrict the form of function definitions at the type level to be inductively sequential [1]. This ensures a sound and complete narrowing strategy for answering type-checking time questions. The class of inductively sequential functions is a large one, in fact every Haskell function has an inductively sequential definition. The inductively sequential restriction affects the form of the equations, and not the functions that can be expressed. Informally, a function definition is inductively sequential if all its clauses are non-overlapping. For example the definition of `zip1` is not-inductively sequential, but the equivalent program `zip2` is.

```
zip1 (x:xs) (y:ys) = (x,y): (zip1 xs ys)
zip1 xs ys = []

zip2 (x:xs) (y:ys) = (x,y): (zip2 xs ys)
zip2 (x:xs) []     = []
zip2 []     ys     = []
```

The definition for `zip1` is not inductively sequential, since its two clauses overlap. In general any non-inductively sequential definition can be turned into an inductively sequential definition by duplicating some of its clauses, instantiating variable patterns with constructor based patterns. This will make the new clauses non-overlapping. We do not think this burden is too much of a burden to pay, since it is applied only to functions at the type level, and it supports sound and complete narrowing strategies. In addition to the inductively sequential form required for type functions, $\Omega$mega assumes that each type function is a total terminating function. This assumption is not currently enforced, and it is up to the programmer to ensure that this is the case.