Programming in Ωmega.

Tim Sheard & Nathan Linger

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

1 Introduction

Ωmega is a language with an infinite hierarchy of computational levels: value, type, kind, sort, etc. Data, and functions manipulating data, can be introduced at any level. Data is introduced by declaring the type of constructors, and functions are introduced by writing (possibly recursive) pattern matching equations.

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. As discussed earlier, 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.

2 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 [21].

```
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` below.

```
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 an append 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`	$\longrightarrow$ `Seq a m` $\longrightarrow$	`Seq a {plus n m}`
equation	`app (Scons x xs)`	`ys`	`= Scons x (app xs ys)`
computed type	`Seq a (S b)`	$\longrightarrow$ `Seq a m` $\longrightarrow$	`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.

Exercise 1. Write an Ωmega function that defines the length function over sequences. `length:: Seq n a -> Int`. You will need to create a file, and paste the definition for `Seq` into the file, as well as write the length function. The `Nat` kind is predefined. You will need to include the function prototype, above, in your file (type inference is limited in Ωmega). How might we reflect the fact that the resulting `Int` should have size `n`? See Section 3.7.

Exercise 2. After you complete Exercise 1, create a table, as we did for `app` above, with expected type, equations, computed type, and equations to be discharged. How might we solve the equations produced?

3 Features of the Ωmega Language

Ωmega is modelled after the Haskell language. There several important difference between Ωmega and Haskell that give Ωmega its unique power of expression. These include.

- **Data Structures at All Levels.** Kinds are a type system for classifying types. Sorts are a type system for classifying kinds. There is

```
←  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
                     | Tp                            :: Shape               :: *1
                     | Nd                            :: Shape               :: *1
                     | Fk                            :: Shape               :: *1
                     | Tree                          :: Shape ~> *0 ~> *0   :: *1
Tip                :: Tree Tp a                     :: *0                  :: *1
Node               :: a -> Tree Nd a                :: *0                  :: *1
Fork               :: Tree x a -> Tree y a ->
                      Tree (Fk x y) a               :: *0                  :: *1
find               :: (a -> a -> Bool) -> a ->
                      Tree sh a -> [Path sh a]      :: *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.

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 tree- structured data 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 the form of such definitions to be inductively sequential (See Appendix B). 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.** Ωmega supports the run-time generation of code, along the lines of MetaML [24] and Template Haskell [25]. The meta-programming ability of code generation allows us to remove a layer of interpretation from our 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.

3.1 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), and Boolean:

```
data Shape :: *1 where
  Tp:: Shape
  Nd:: Shape
  Fk:: Shape ~> Shape ~> Shape
```

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

Like the kind Nat defined earlier, Shape and Boolean also define new kinds, and new types classified by these kinds. The new tree-like data at the type level are constructed by the type-constants (To, Nd, T, F), and type constructors (Fk and S). The kinds Shape and Boolean classify these structures, as shown explicitly in the declaration. For example T is classified by Boolean, and Fk is a constructor from Shape to Shape to Shape. Think of the operator ~> as an function arrow at the type level. Note that while To, Nd, T, and F 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 the types introduce by Nat, Shape, and Boolean, they are very useful. Instead of using them to classify values, we use them as indexes to value level data, i.e. types like Proof {even n} and Seq a (S Z). The indexes like {even n} 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.

Exercise 3. Write a data declaration introducing a new kind called Color with types Red and Black. Are there any values with type Red? Now write a data declaration introducing a new type Tree which is indexed by Color (this will be similar to the use of Nat in the declaration of Seq). There should be some values classified by the type (Tree Red), and others classified by the type (Tree Black).

3.2 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 indexing Seq. We may also 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 functions are strict total (terminating) functions at the type level.

Exercise 4. Write an Ωmega function `mult`, which is the multiplication function at the type level over natural numbers. It should be classified by the kind `mult:: Nat ~> Nat ~> Nat`.

Exercise 5. Write the `odd` function classified by the kind `Nat ~> Boolean`.

Exercise 6. Write the `or` and `not` functions, that are classified by the kinds `(Boolean ~> Boolean ~> Boolean)` and `(Boolean ~> Boolean)`. Which arguments of `or` should you pattern match over? Does it matter? Experiment, Ωmega won't allow some combinations. See Appendix B on inductively sequential definitions and narrowing for the reason why.

3.3 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 Ωmega 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 GADTs `Seq`, `Path` and `Tree`:

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

data Path:: Shape ~> *0 ~> *0 where
  None :: Path Tp a
  Here :: b -> Path Nd b
  Left :: Path x a -> Path (Fk x y) a
  Right:: Path y a -> Path (Fk x y) a

data Tree :: Shape ~> *0 ~> *0 where
  Tip:: Tree Tp a
  Node:: a -> Tree Nd a
  Fork::  Tree x a -> Tree y a -> Tree (Fk x y) a
```

Note, that instead of ranges like `(Seq a b)`, and `(Path a b)` where only type variables like `a`, and `b` can be used as parameters, the ranges contain sophisticated instantiations such as `(Seq a (S n))` and `(Path Nd)`. Note that the second index to `Seq` (the one of kind `Nat`) is used to describe an invariant about the length of the sequence, and the `Shape` index to `Path`, indicates the shape of a tree in which that path is legal. This is one of the many uses of GADTs – to enforce invariants about the structure of data. Notice how the shape of `tree1` appears in its type.

```
tree1 :: Tree (Fk  (Fk   Tp  Nd)       (Fk   Nd       Nd))        Int
tree1 =       Fork (Fork Tip (Node 4)) (Fork (Node 4) (Node 3))
```

We can write pattern matching functions over GADTs just as we can over algebraic datatypes. The only caveat is that we must specify the type of the function using a prototype. *Ω*mega does type checking of functions over GADTs rather than type inference.

Suppose we wanted to search a tree, returning all paths that lead to a particular element. It would be nice to know that every path returned was a legal path within the tree. For example `(Left (Here 2))` is not a legal path within the tree `Tip`. The `Shape` index allows us to specify that our searching function always obeys this invariant.

```
find:: (a -> a -> Bool) -> a -> Tree sh a -> [Path sh a]
find eq n Tip = []
find eq n (Node m) =
  if eq n m then [Here n] else []
find eq n (Fork x y) =
  map Left (find eq n x) ++
  map Right (find eq n y)
```

The type of `find` guarantees that every path returned is a legal path within the tree searched, because both the tree and every path in the list has the same `Shape`, namely `sh`.

Exercise 7. Write an Ωmega function with type `extract:: Path sh a -> Tree sh a -> a`, which extracts the value of type `a`, stored in the tree at the location pointed to by the path. This function will pattern match over two arguments simultaneously. Some combinations of patterns are not necessary. Why? See section 3.10 for how you can document this fact.

Exercise 8. Replicate the shape index pattern for lists. Write two Ωmega GADTs. One at the kind level which encodes the shape of lists, and one at the type level for lists indexed by their shape. Also, write a find function for your new types. `find:: (a -> a -> Bool) -> a -> List sh a -> Maybe(ListPath sh a)`, which returns the first path, if one exists.

Since every GADT is comprised of a sum of products, can you define a single shape kind, that could be used for all parametric datatypes?

Exercise 9. Consider the GADT below.

```
data Rep :: *0 ~> *0 where
   Int :: Rep Int
   Prod :: Rep a -> Rep b -> Rep (a,b)
   List :: Rep a -> Rep [a]
```

Construct a few terms. Do you note any thing interesting about this type? Write a function with the following type: `showR:: Rep a -> a -> String`, which given values of type `Rep a` and `a`, displays the second as a string. Extend this GADT with a few more constructors, then extend your `showR` function as well.

3.4 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 2) plays a qualitatively different role than type arguments in ordinary ADTs. Consider the declaration for a binary tree datatype in Haskell:

```
data HTree a = HFork (HTree a) (HTree a) | HNode a | HTip
```

In this declaration the type parameter `a` is used to indicate that there are sub components of `HTree`s that are of type `a`. In fact, `HTree`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 `HTree`'s type. Contrast this with the `n` in `(Seq a n)`, and the `sh` in `(Tree sh a)`. Instead, the parameter `n` is used to stand for an abstract property (the length of the list represented), and the parameter `sh` is used to stand for the shape of the tree. When we use a type parameters in this way we call it a type index [40, 43] 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 several new indexed types `Proof`, `Plus`, `LE` and `Even`. Note that these are ordinary data structures that exist at the value level, but describe properties at the type level.

```
data Proof:: Boolean ~> *0 where
  Triv:: Proof T

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

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

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

These declarations introduce value-level constants (`Triv`, `EvenZ`, `PlusZ`, and `LeZ`) and constructor functions (`EvenSS`, `PlusS`, and `LeS`). Let's examine the types of several values constructed with these constructors.

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

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

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

even2 :: Proof {even 2t}
even2 = Triv
```

To make it easier to handle types of kind `Nat`, in *Ω*mega, we have special syntactic sugar for entering and displaying them: `Z` = `0t`, `S Z` = `1t`, and `S(S Z)` = `2t`, etc. We may also write `(1+x)t` for `S x`, and `(2+x)t` for `S(S x)`, etc. We introduce this notation here (see Section 3.13 for more detail) 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)`, `(Even n)`, and `(Proof {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 1t)`. This is the essence of the Curry-Howard isomorphism. We can view `(EvenSS Even):: Even 2t` as either the statement that `(EvenSS EvenZ)` has type `Even 2t`, or that `(EvenSS EvenZ)` is a proof of the property `Even 2t`.

In the same fashion, the type system will reject ill-typed terms that witness false statements. For example, consider the response when we try to type the term `Triv` with the type `(Proof {even 1t})`

```
bad:: Proof {even 1t}
bad = Triv

While type checking in the scope of:
   bad
We need to prove:
   Equal {even 1t} T
From the truths:
And the rules:S,Z,
But, The equations: (F=T) =>  have no solution
```

All this follows directly from the introduction of new types as GADTs and the ability to define them, and to compute over them, at arbitrary levels.

Exercise 10. Construct terms with the types `(Plus 2t 3t 5t)`, `(Plus 2t 1t 3t)`, and `(Plus 2t 6t 8t)`. What did you discover?

Exercise 11. Write an *Ω*mega function with the following type: `summandLessThanSum:: Plus a b c -> LE a c`. Hint: it is a recursive function. Can you write a similar function with type `(Plus a b c -> LE b c)`?

3.5 Pattern: Witness vs. Type Function

The reader may have noticed that `(Proof {even n})` and `(Even n)` 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.

This is actually more general than it appears. The type function does not have to have `Boolean` as its range. Instead, for every n-ary function at the type level, we can build an $(n+1)$-ary witness type. We express the equality between a function call and its result: `{function a b} = c` as an relation: `{Relation a b c}`.

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)`.

Exercise 12. Use the pattern above to define a GADT (a type constructor with 2 arguments) that witnesses the `even` type function.

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.

3.6 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 [32, 23]. The cost associated with this avoidance is the possible duplication of data structures and functions at several levels.

3.7 Pattern: A pun: Nat'

We now define the type `Nat'`, which is identical structurally to the type `SNat`. As such, the type `Nat'` is also a singleton type representing the natural numbers, but it relies on an feature of the Ωmega type system. In Ωmega (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)`.

Exercise 13. Write the two Ωmega functions with types: `same:: Nat' n -> LE n n`, and `predLE:: Nat' n -> LE n (S n)`. Hint they are simple recursive functions.

Exercise 14. Write the Ωmega function which witnesses the transitivity of the less-than-or-equal to predicate. `trans:: LE a b -> LE b c -> LE a c`. Hint: it is a recursive function with pattern matching over both arguments. One of the cases is not reachable. Which one? Why? See Section 3.10 for how you can document this fact.

Exercise 15. In Exercise 11 we proposed writing a function with type `(Plus a b c -> LE b c)`. This turned out to be quite difficult. But, it is possible to write a function with type `(Nat' b -> Plus a b c -> LE b c)`. Write this function. What benefit does the first `Nat' b` argument provide? Hint: both the functions `trans` and `predLE` come in useful.

3.8 Pattern: Leibniz Equality

Terminating terms of type `(Equal lhs rhs)` are values witnessing the equality of `lhs` and `rhs`. The type constructor `Equal` is defined as:

```
data Equal :: a ~> a ~> *0 where
  Eq:: Equal x x
```

Note that `Equal` is a GADT, since in the type of the constructor `Eq` the two type indexes are the same, and not just polymorphic variables (i.e. the type of `Eq` is *not* `(Equal x y)` but is rather `(Equal x x)`). The single constructor (`Eq`) has a polymorphic type `(Equal x x)`. Ordinarily, if the two arguments of `Equal` are type-constructor terms, the two arguments must be the same (or they couldn't be equal). But, if we allow type functions as arguments (see Section 3.2), since many functions may compute the same result (even with different arguments), the two terms can be syntactically different (but semantically the same). For example `(Equal 2t {plus 1t 1t})` is a well formed equality type since 2 is semantically equal to 1+1. The `Equal` type allows the programmer to reify the type checkers notion of equality, and to pass this reified evidence around as a value. The `Equal` type plays a large role in the `theorem` declaration (see Section 6).

Exercise 16. Singleton types allow us to construct Equal objects at runtime. Because of the one-to-one relationship between singleton values and their types, knowing the shape of a value determines its type. In a similar manner knowing the type of a singleton determines its shape. Write the function in Ωmega that exploits this fact: `sameNat:: Nat' a -> Nat' b -> Maybe(Equal a b)`. I have written the first clause. You can finish it.

```
sameNat:: Nat' a -> Nat' b -> Maybe(Equal a b)
sameNat Z Z = Just Eq
```

If one wonders how this function is typed, it is very instructive to construct the typing box (as we did for `app` in Section 2) with expected types, equations, computed types, and generated equalities.

3.9 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 [6]. 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 Ωmega 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.

In a similar manner, given a proof that $a \leq b$ we can always find a c such that $a + c = b$.

```
smaller :: Proof {le (S a) (S b)} -> Proof {le a b}
smaller Triv = Triv

diff:: Proof {le a b} -> Nat' a -> Nat' b ->
       exists c .(Nat' c,Equal {plus a c} b)
diff Triv Z m = Ex (m,Eq)
diff Triv (S m) Z = unreachable
diff (q@Triv) (S x) (S y) =
  case diff (smaller q) x y of
   Ex (m,Eq) -> Ex (m,Eq)
```

Exercise 17. The filter function drops some elements from a list. Thus, the length of the resulting list cannot be known statically. But, we can compute the length of the resulting list along with the list. Write the Ωmega function with type:

```
filter :: (a->Bool) -> Seq a n -> exists m . (Nat' m,Seq a m)
```

Since filter never adds elements to the list, that weren't already in the list, the result-list is never shorter than the original list. We can compute a proof of this fact as well. Write the Ωmega function with type:

```
filter :: (a->Bool) -> Seq a n -> exists m . (LE m n,Nat' m,Seq a m)
```

Hint: You may find the functions `predLE` and `trans` from Exercises 13 and 14 useful.

3.10 Feature: Unreachable Clauses

The key word `unreachable` in the second clause of the definition for `diff` states that type considerations preclude the flow of control ever reaching the clause labled unreachable. This is because the type information in the function prototype for `diff` is propagated into the patterns of each clause. In the second clause the following information is propagated.

```
Triv  :: Proof {le a b}
```

```
(S m) :: Nat' a
Z     :: Nat' b ->
```

We compute the type of `(S m)` to be `(Nat' m)`, and we compute the type of `Z` to be `(Nat' Z)`, combining this with the propagated type information we see that `a = (S m)` and `b = Z`. Thus the type of `Triv` must be `Proof {le (S m) Z}`. The type function application `{le (S m) Z}` reduce to `F`, but the argument to `Proof` must be `T`. These sets of assumptions are inconsistent. So the clause in the scope of these patterns is unreachable. There are no well-typed arguments, to which we could apply `diff`, that would exercise the second clause. The keyword `unreachable` indicates to the compiler that we recognize this fact. The reachability of all unreachable clauses is tested. If they are in fact reachable, an error is raised. An unreachable clause, without the `unreachable` keyword also raises an error.

The point of the unreachable clause is to document that the author of the code know that this clause is unreachable, and to help document that the clauses exhaustively cover all possible cases. The function `extract` from exercise 7 and the function `trans` from exercise 14 could use an unreachable clause.

3.11 Feature: Staging.

*Ω*mega supports the staging annotations: brackets: (`[| _ |]`), escape (`$( _ )`), and the two staging functions: `lift::(forall a . a -> Code a)` and `run::(forall a . (Code a) -> a)` for building and manipulating code. *Ω*mega uses the Template Haskell [25] conventions for creating code. Brackets (`[| _ |]`) are a quasi-quotation mechanism, and escape (`$( _ )`) 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. Ωmega supports many of the features of MetaML [24, 36].

Exercise 18. The traditional staged function is the power function. The term `(power 3 x)` returns x to the third power. The unstaged power function can be written as:

```
power:: Int -> Int -> Int
power 0 x = 1
power n x = x * power (n-1) x
```

Write a staged power function: `pow:: Int -> Code Int -> Code Int` such that `(pow 3 [|99|])` evaluates to `[| 99 * 99 * 99 * 99 * 1 |]`. This can be written simply by placing staging annotations in the unstaged version.

3.12 Feature: Level Polymorphism

Some times we wish to use the same structure at both the value and type level. One way to do this is to build isomorphic, but different, data structures at different levels. In Ωmega, we can define a structure to live at many levels. We call this level polymorphism. For example a `Tree` type that lives at all levels can be defined by:

```
data Tree :: level n . *n ~> *n where
  Tip :: a ~> Tree a
  Fork :: Tree a ~> Tree a ~> Tree a
```

Levels are *not* types. A level variable can only be used as an argument to the `*` operator. Level abstraction can only be introduced in the kind part of a `data` declaration, but level polymorphic functions can be inferred from their use of constructor functions introduced in level polymorphic `data` declarations.

In the example above, Ωmega adds the type constructor `Tree` at all type levels, and the constructors `Tip` and `Fork` at the value level as well at all type levels. We can illustrate this by evaluating a tree at the value level, and by asking Ωmega for the kind of a similar term at the type level.

```
prompt> Fork (Tip 3) (Tip 1)
(Fork (Tip 3) (Tip 1)) : Tree Int

prompt> :k Tip Int
Tip Int :: Tree *0
```

Another useful pattern is to define normal (`*0`) datatypes indexable by types at all levels. For example consider the kind of the type constructor `Equal` and the type of its constructor `Eq` from Section 3.8. Its type can be more verbosely expressed as follows where the level polymorphism is explicit (rather than inferred, as it is in Section 3.8).

```
Equal :: level b . forall (a:*(1+b)).a ~> a ~> *0

Eq :: level b . forall (a:*(1+b)) (c:a:*(1+b)).Equal c c
```

Without level polymorphism, the `Equal` type constructor could only witness equality between types a single level. I.e. types classified by `*0` but not `*1`. So `(Equal Int Bool)` is well formed but `(Equal Nat Tag)` would not be, since both `Nat` and `Tag` are classified by `*1`. For a useful example, the type of `labelEq` could not be expressed using a level-monomorphic `Equal` datatype.

```
labelEq:: forall (a:Tag) (b:Tag). Label a -> Label b -> Maybe (Equal a b)
```

This is because the `a` and `b` are classified by `Tag`, and are not classified by `*0`.

Exercise 19. A row is a list like structure that associates a pair of objects. In *Ω*mega we write `{‘a=Int,‘z=Bool}r` for the row classified by `(Row Tag *0)`, which associates the `Tag` `‘a` with `Int`, and `‘z` with `Bool`. In general we'd like not to restrict rows to any single level. Level polymorphism comes in handy here. Write a GADT that defines a level polymorphic row type at level 1, but which is indexed by a pair of types from any level. I.e. `Row` should be classified as follows:

```
Row :: level d b . forall (a:*(2+b)) (c:*(2+d)). a ~> c ~> *1
```

3.13 Feature: Syntactic Extension

Many languages supply syntactic sugar for constructing homogeneous sequences and heterogeneous tuples. For example in Haskell lists are often written with bracketed syntax, `[1,2,3]`, rather than a constructor function syntax, `(Cons 1 (Cons 2 (Cons 3 Nil)))`, and tuples are often written as `(5,"abc")` and `(2,True,[])` rather than `(Pair 5 "abc")` and `(Triple 2 True [])`. In *Ωmega* we supply special syntax for four different kinds of data, and allow users to use this syntax for data they define themselves. *Ωmega* has special syntax for list-like, natural-number-like, pair-like, and record-like types. Some examples in the supported syntax are: `[4,5]i`, `(2+n)j` `(4,True)k`, and `{"a"=5, "b"=6}h` In general, the syntax starts with list-like, natural-number-like, record-like, or pair-like syntax, and is terminated by a tag. A user may specify that a user defined type should be displayed using the special syntax with a given tag. Each tag is associated with a set of functions (a different set for list-like, natural-number-like, record-like, and pair-like types). Each term expands into a call of the functions specified by the tag in the special syntax.

The list-like syntax associates two functions with each tag. These functions play the role of `Nil` and `Cons`. For example if the tag "`i`" is associated with the functions `(C,N)`, then the expansion is as follows.

```
[]i          ---> N
[x,y,z]i     ---> C x(C y (C z N))
[x;xs]i      ---> (C x xs)
[x,y ; zs]i ---> C x (C y zs)
```

The semi colon may only appear before the last element in the square brackets. In this case, the last element stands for the tail of the resulting list.

The natural-number-like syntax associates two functions with each tag. These functions play the role of `Zero` and `Succ`. For example if the tag "`i`" is associated with the functions `(Z,S)`, then the expansion is as follows.

```
4i      ---> S(S(S(S Z)))
0i      ---> Z
(2+x)i ---> S(S x)
```

For backward compatibility reasons, to represent the built in types `Nat` and `Nat'`, the syntax `4t` is equivalent to either `4t` (`S(S(S(S Z)))`) in the type name space, and `4v` (`S(S(S(S Z)))`) in the value name space.

The pair-like syntax associates one function with each tag. This function plays the role of a binary constructor. For example if the tag "`i`" is associated with the function `(P)`, then the expansion is as follows.

```
(a,b)i       ---> P a b
(a,b,c)i     ---> P a (P b c)
(a,b,c,d)i  ---> P a (P b (P c d))
```

The record-like syntax associates two functions with each tag. These functions play the role of the constant `RowNil` and the ternary function `RowCons`. For example if the tag "`i`" is associated with the functions `(RC,RN)`, then the expansion is as follows.

```
{}i            ---> RN
{a=x,b=y}i    ---> RC a x (RC b y RN)
{a=x;xs}i     ---> (RC a x xs)
{a=x,b=y ; zs}i ---> RC a x (RC b y zs)
```

Syntactic extension can be applied to any GADT, at either the value or type level. The new syntax can be used by the programmer for terms, types, or patterns. *Ω*mega uses the new syntax to display such terms. The constructor based mechanism can also still be used. The tags are specified using a deriving clause in a GADT. See Section 5.9 for an example use of this feature.

Exercise 20. Consider the GADt with syntactic extension "`i`".

```
data Nsum:: *0 ~> *0 where
  SumZ:: Nsum Int
  SumS:: Nsum x -> Nsum (Int -> x)
 deriving Nat(i)
```

What is the type of the terms `0i`, `1i`, and `2i`. Can you write a function with type: `add:: Nsum i -> i`. Such a function sums n integers given n`i` as input. For example: `add 3i 1 2 3` $\longrightarrow$ `6`.

3.14 Feature: Tags and Labels

Many object languages have a notion of name. To make representing names in the type system easy we introduce the notion of Tags and Labels. As a *first approximation*, consider the finite kind `Tag` and its singleton type `Label`:

```
data Tag:: *1 where
  A:: Tag
  B:: Tag
  C:: Tag

data Label:: Tag ~> *0 where
  A:: Label A
  B:: Label B
  C:: Label C
```

Here, we again deliberately use the value-name space, type-name space overloading. The names `A`, `B`, and `C` name different, but related, objects at both the value and type level. At the value level, every `Label` has a type index that reflects its value. I.e. `A::Label A`, and `B::Label B`, and `C::Label C`. Now consider a countably infinite set of tags and labels. We can't define this explicitly, but we can build such a type as a primitive inside of Ωmega. At the type level, every legal identifier whose name is preceded by a back-tick (`‘`) is a type classified by the kind `Tag`. For example the type `‘abc` is classified by `Tag`. At the value level, every such symbol `‘x` is classified by the type `(Label ‘x)`.

There are several functions that operate on labels. The first is `labelEq` which compares two labels for equality. Since labels are singletons, a simple true or false answer would be useless. Instead `labelEq` returns a Leibniz proof of equality (see Section 3.8) that the `Tag` indexes of identical labels are themselves equal.

```
labelEq :: forall (a:Tag) (b:Tag).Label a -> Label b -> Maybe (Equal a b)

prompt> labelEq ‘w ‘w
(Just Eq) : Maybe (Equal ‘w ‘w)

prompt> labelEq ‘w ‘s
Nothing : Maybe (Equal ‘w ‘s)
```

Fresh labels can be generated by the function `freshLabel`. Since the `Tag` index for such a label is unknown, the generator must return a structure where the `Tag` indexing the label is existentially quantified. Since every call to `freshLabel` generates a different label, the `freshLabel` operation must be an action in the `IO` monad. The function `newLabel` coerces a string into a label. It too, must existentially hide the Tag indexing the returned label. But, because it always returns the same label when given the same input it can be a pure function.

```
freshLabel :: IO HiddenLabel
newLabel:: String -> HiddenLabel

data HiddenLabel :: *0 where
 Hidden:: Label t -> HiddenLabel
```

We illustrate this at the top-level loop. The Ωmega top-level loop executes `IO` actions, and evaluates and prints out the value of expressions with other types.

```
prompt> freshLabel
Executing IO action              -- An IO action
(Hidden ‘#cbp) : IO HiddenLabel
```

```
prompt> temp <- freshLabel        -- An IO action
Executing IO action
(Hidden ‘#sbq) : HiddenLabel
prompt> temp
(Hidden ‘#sbq) : HiddenLabel

prompt> newLabel "a"              -- A pure value
(Hidden ‘a) : HiddenLabel
```

Exercise 21. A common use of labels is to name variables in a data structure used to represent some object language as data. Consider the GADT and an evaluation function over that object type.

```
data Expr:: *0 where
  VarExpr :: Label t -> Expr
  PlusExpr:: Expr -> Expr -> Expr

valueOf:: Expr -> [exists t .(Label t,Int)] -> Int
valueOf (VarExpr v) env = lookup v env
valueOf (PlusExpr x y) env = valueOf x env + valueOf y env
```

Write the function: `lookup:: Label v -> [exists t .(Label t,Int)] -> Int`.

4 Maintaining Structural Invariants of Data

Both `Seq` and `Tree` use kinds as indexes (`Nat` for `Seq`, and `Shape` for `Tree`) to maintain an invariant about the shape of the data. This is quite common. In this section we illustrate this in more detail by examining the world of balanced trees.

4.1 AVL Trees

Binary search trees are a classic data structure for implementing finite maps or sets in a purely functional way. To guarantee efficient operations, we want our trees to be somewhat balanced. There are several ways to define what it means for a tree to be balanced, each leading to different data structures such as Red-Black trees, AVL trees, B-trees, etc. In this section we implement AVL trees in such a way that Ωmega's type system guarantees compliance with the balancing invariant.

Types Expressing Invariants. The balancing invariant for AVL trees is simple: any internal node in the tree has children whose heights differ by no more than one. In this section, we define types that express this invariant. Here is our core data structure for AVL trees (indexed by tree height).

```
data Avl :: Nat ~> *0 where
  Leaf :: Avl Z
  Node :: Balance hL hR hMax -> Avl hL -> Int -> Avl hR -> Avl (S hMax)
```

A binary tree has two constructors – one for (empty) leaves and one for internal nodes carrying data. An auxiliary type captures the balancing constraints.

```
data Balance:: Nat ~> Nat ~> Nat ~> *0 where
  Less :: Balance h     (S h) (S h)
  Same :: Balance h     h     h
  More :: Balance (S h) h     (S h)
```

Think of the type `Balance hL hR hMax` as a predicate stating (1) that `hL` and `hR` differ by at most one, and (2) that `hMax` is the maximum of `hL` and `hR`. For any given internal node, there are only three possibilities for the relative heights of its subtrees:

$$1 + \mathtt{hL} = \mathtt{hR} \quad \text{or} \quad \mathtt{hL} = \mathtt{hR} \quad \text{or} \quad \mathtt{hL} = \mathtt{hR} + 1$$

These three possibilities correspond to the three constructors of the datatype `Balance`. Under this interpretation of `Balance`, we see that the `h` in `(Avl h)` really does capture the height of the tree (leaves have height zero and the height of an internal node is the successor of the maximum of the heights of its children).

Finally, we would like to protect users of our library from having to deal with height indices in their own code. To this end, we define a wrapper type that hides away the height index.

```
data AVL:: *0 where
  AVL:: (Avl h) -> AVL
```

In this type the `h` is existentially quantified. This is the type that users will see.

That is all the code we ever need write to guarantee that every AVL tree in our library is well-balanced. Because these type declarations express the balancing invariants, the problem of deciding whether our implementation respects those invariants reduces to the problem of deciding type-correctness, which the Ωmega type-checker does for us automatically.

Basic operations. The two most basic operations are constructing an empty tree and testing an element for membership in the tree.

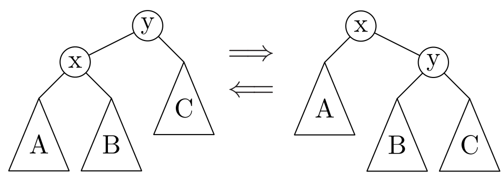
```
empty :: AVL
empty = AVL Leaf
```

```
element :: Int -> AVL -> Bool
element x (AVL t) = elem x t

elem :: Int -> Avl h -> Bool
elem x Leaf = False
elem x (Node _ l y r)
  | x == y  = True
  | x < y   = elem x l
  | x > y   = elem x r
```

The remaining operations of insertion and deletion are much more interesting.

Balancing Constructors. The algorithms for insertion and deletion each follow the same basic pattern: First do the insertion (or deletion) as you would for any other binary search tree. Then re-balance any subtree that became unbalanced in the process. The tool used for re-balancing is tree rotation, which is best described visually.



The transformation of the tree on the left to the tree on the right is *right rotation* and the opposite transformation is called *left rotation*. This operation preserves the BST invariant. However, they do *not* preserve the balancing invariant, which is precisely why they are useful for rebalancing.

It turns out that we can package up all necessary rotations into a couple of *smart constructors*, `rotr` and `rotl`. Think of `rotr lc x rc` as a smart version of `Node ? lc x rc` where

1. We don't have to say (or know) how the resulting tree is balanced, and
2. The subtrees, `lc` and `rc`, don't quite balance out because `height(lc)` = `height(rc) + 2` and therefore we must do some rightward rebalancing rotation(s).

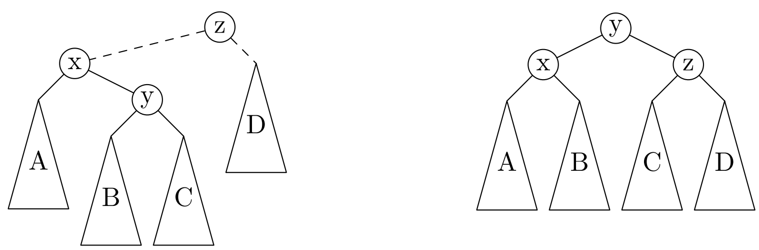
The only wrinkle in the "smart constructor" story is that the height of the resulting tree depends on what rotations were performed. However, the result height ranges over merely two values, so we just return a value of a sum type[1]. Here is the code:

[1] In Ωmega the value constructors `L :: a -> (a+b)` and `R :: b -> (a+b)` are used to construct sums.

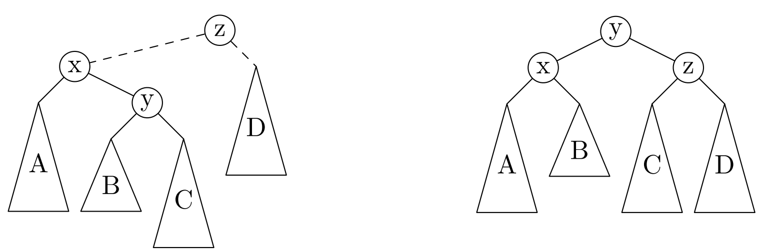
```
rotr (Node Same a x b) y c      = R(Node Less a x (Node More b y c))
```


```
rotr (Node More a x b) y c      = L(Node Same a x (Node Same b y c))
```


```
rotr (Node Less a x                = L(Node Same (Node Same a x b) y
           (Node Same b y c)) z d                 (Node Same c z d) )
```




```
rotr (Node Less a x                = L(Node Same (Node More a x b) y
           (Node Less b y c)) z d                 (Node Same c z d) )
```




```
rotr (Node Less a x                = L(Node Same (Node Same a x b) y
           (Node More b y c)) z d                 (Node Less c z d) )
```

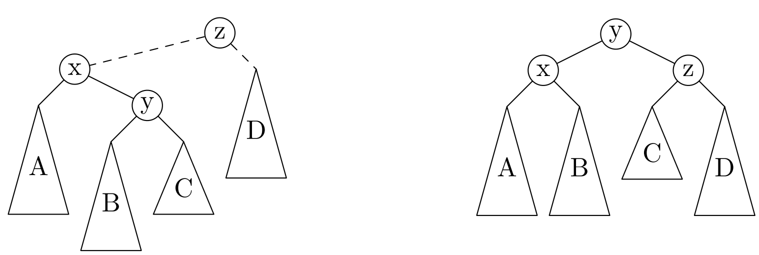


Fig. 2. Each substantive case in the definition of `rotr`.

```
rotr :: Avl (2+n)t -> Int -> Avl n -> ( Avl(2+n)t + Avl (3+n)t )
rotr Leaf x a = unreachable
rotr (Node Less a x Leaf) y b = unreachable
-- single rotations
rotr (Node Same a x b) y c = R(Node Less a x (Node More b y c))
rotr (Node More a x b) y c = L(Node Same a x (Node Same b y c))
-- double rotations
rotr (Node Less a x (Node Same b y c)) z d =
   L(Node Same (Node Same a x b) y (Node Same c z d))
rotr (Node Less a x (Node Less b y c)) z d =
   L(Node Same (Node More a x b) y (Node Same c z d))
rotr (Node Less a x (Node More b y c)) z d =
   L(Node Same (Node Same a x b) y (Node Less c z d))
```

Figure 2 depicts the rotation for each substantive case in the definition of `rotr`. The algorithm for `rotl` is perfectly symmetric to that for `rotr`.

```
rotl :: Avl n -> Int -> Avl (2+n)t -> ( Avl (2+n)t + Avl (3+n)t )
rotl a x Leaf = unreachable
rotl a x (Node More Leaf y b) = unreachable
-- single rotations
rotl a u (Node Same b v c) = R(Node More (Node Less a u b) v c)
rotl a u (Node Less b v c) = L(Node Same (Node Same a u b) v c)
-- double rotations
rotl a u (Node More (Node Same x m y) v c) =
   L(Node Same (Node Same a u x) m (Node Same y v c))
rotl a u (Node More (Node Less x m y) v c) =
   L(Node Same (Node More a u x) m (Node Same y v c))
rotl a u (Node More (Node More x m y) v c) =
   L(Node Same (Node Same a u x) m (Node Less y v c))
```

As these functions are both self-contained and non-recursive, we see that they operate in constant time.

Insertion When we insert an element into an AVL tree, the height of the tree either remains the same or increases by at most one. We therefore arrive at the following type for insertion:

```
ins :: Int -> Avl n -> (Avl n + Avl (S n))
```

The code for `ins` is an elaborate case analysis. The first decision to make is whether we're at the right spot for insertion. If so, then do the insertion (or not, depending on whether the value already exists in the tree), and then return. If not, make the appropriate recursive call and then rebalance. Most of the work goes into determining how to rebuild a balanced tree by choosing the correct `Balance` value or rebalancing constructor.

```
ins :: Int -> Avl n -> (Avl n + Avl (S n))
ins x Leaf = R(Node Same Leaf x Leaf)
```

```
ins x (Node bal a y b)
  | x == y = L(Node bal a y b)
  | x < y  = case ins x a of
               L a -> L(Node bal a y b)
               R a ->
                 case bal of
                   Less -> L(Node Same a y b)
                   Same -> R(Node More a y b)
                   More -> rotr a y b -- rebalance!
  | x > y  = case ins x b of
               L b -> L(Node bal a y b)
               R b ->
                 case bal of
                   Less -> rotl a y b -- rebalance!
                   Same -> R(Node Less a y b)
                   More -> L(Node Same a y b)
```

Figure 3 depicts each case in the `x < y` branch. Now we wrap this function up to work on user-level AVL trees.

```
insert :: Int -> AVL -> AVL
insert x (AVL t) = case ins x t of L t -> AVL t; R t -> AVL t
```

Deletion Whereas insertion always places an element in the fringe of a tree, deletion may find the targeted element somewhere in the interior. For this reason, deletion is a more complex operation. The strategy for deleting the value x at an interior node is to first replace its value with that of the minimum value z of its right child (or the maximum value of its left child, depending on the policy). Then delete z (which is always at a leaf) from the right child.

We will calculate the minimum value in a tree and delete it in a single pass. The operation only works on trees of height ≥ 1 (which are therefore non-empty). The returned tree might have decreased in size by one.

```
delMin :: Avl (S n) -> (Int, (Avl n + Avl (S n)))
delMin Leaf = unreachable
delMin (Node Less Leaf x r) = (x,L r)
delMin (Node Same Leaf x r) = (x,L r)
delMin (Node More Leaf x r) = unreachable
delMin (Node bal (l@(Node _ _ _ _)) x r) =
      case delMin l of
        (y,R l) -> (y,R(Node bal l x r))
        (y,L l) ->
          case bal of
            Same -> (y,R(Node Less l x r))
            More -> (y,L(Node Same l x r))
            Less -> (y,rotl l x r) -- rebalance!
```

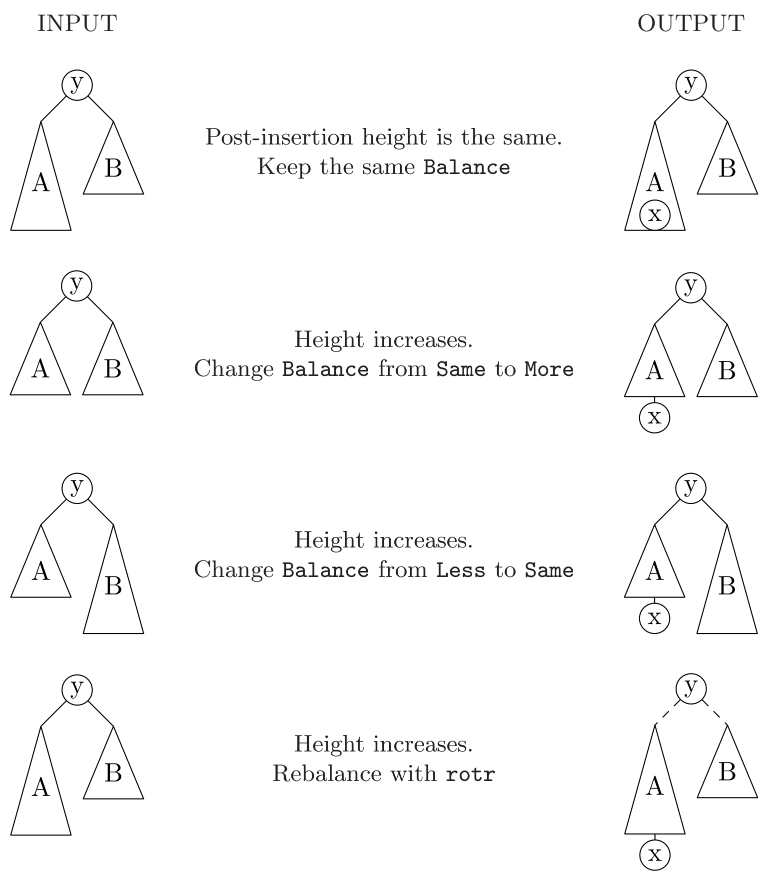


Fig. 3. Rebalancing after insertion into the left child.

Deletion of the minimum element requires rebalancing operations on the way up, just as in insertion.

When we delete an element into an AVL tree, the height of the tree either remains the same or decreases by at most one. We therefore arrive at the following type for deletion:

```
del :: Int -> Avl n -> (Avl n + Avl (S n))
```

The code for `del` is an elaborate case analysis. The first decision to make is whether we're at the right spot for deletion. If so, then do the deletion (or not, depending on whether the value exists in the tree) and return. If not, make the appropriate recursive call and then rebalance. Most of the work goes into determining how to rebuild a balanced tree by choosing the correct `Balance` value or rebalancing constructor.

```
del :: Int -> Avl n -> (Avl n + exists m .(Equal (S m) n,Avl m))
del y Leaf = L Leaf
del y (Node bal l x r)
  | y == x = case r of
              Leaf ->
                case bal of
                  Same -> R(Ex(Eq,l))
                  More -> R(Ex(Eq,l))
                  Less -> unreachable
              Node _ _ _ _ ->
                case (bal,delMin r) of
                  (_,z,R r) -> L(Node bal l z r)
                  (Same,z,L r) -> L(Node More l z r)
                  (Less,z,L r) -> R(Ex(Eq,Node Same l z r))
                  (More,z,L r) ->
                     case rotr l z r of -- rebalance!
                       L t -> R(Ex(Eq,t))
                       R t -> L t
  | y < x = case del y l of
             L l -> L(Node bal l x r)
             R(Ex(Eq,l)) ->
               case bal of
                 Same -> L(Node Less l x r)
                 More -> R(Ex(Eq,Node Same l x r))
                 Less ->
                    case rotl l x r of -- rebalance!
                      L t -> R(Ex(Eq,t))
                      R t -> L t
  | y > x = case del y r of
             L r -> L(Node bal l x r)
             R(Ex(Eq,r)) ->
               case bal of
                 Same -> L(Node More l x r)
                 Less -> R(Ex(Eq,Node Same l x r))
```

```
                More ->
                   case rotr l x r of -- rebalance!
                     L t -> R(Ex(Eq,t))
                     R t -> L t
```

Now we wrap this function up to work on user-level AVL trees.

```
delete :: Int -> AVL -> AVL
delete x (AVL t) = case del x t of L t -> AVL t; R t -> AVL t
```

Exercise 22. **Red Black Trees** A red-black tree is a binary search tree with the following additional invariants:

1. Each node is colored either red or black
2. The root is black
3. The leaves are black
4. Each Red node has Black children
5. For all internal nodes, each path from that node to a descendant leaf contains the same number of black nodes.

We can encode these invariants by thinking of each internal node as having two attributes: a color and a black-height. We will use a GADT, we call `SubTree`, with two indexes, one of them a `Nat` (for the black-height) and the other a `Color`.

```
data Color:: *1 where
  Red:: Color
  Black:: Color

data SubTree:: Color ~> Nat ~> *0 where
 Leaf:: SubTree Black Z
 RNode:: SubTree Black n -> Int -> SubTree Black n -> SubTree Red n
 BNode:: SubTree cL m    -> Int -> SubTree cR m    -> SubTree Black (S m)

data RBTree:: *0 where
 Root:: SubTree Black n -> RBTree
```

Note how the black height increases only on black nodes. The type `RBTree` encodes a "full" Red-Black tree, forcing the root to be black, but placing no restriction on the black-height. Write an insertion function for Red-Black trees. A solution to this exercise is found in Appendix A.

5 *Ω*mega as a Meta Language

It has become common practice when designing a new language to study the relationship between a static semantics (a type system) and a dynamic semantics (a meaning function). This process is often exploratory.

The designer has an idea, the approach is analyzed, and hopefully the consequences of the approach are quickly discovered. Automated aid in this process would be a great boon.

The ultimate goal of this exploratory process is a type system, a semantics, and a proof. The proof witnesses the fact that *well-typed programs do not go wrong*[16] for the language under consideration. The most common way to perform such a proof is by a subject reduction proof in the style of Wright and Felleisen[39] on a small step semantics, though there other approaches as well[16, 10]. Such proofs require an amazing amount of detail and are most often carried out by hand, and are thus subject to all the foils of human endeavors.

Ωmega is my attempt at developing a generic meta-language that could be used for exploring the static and dynamic semantics for new object-languages[19, 26] that could aid in the generation of such proofs. This section describes how Ωmega can be used as a meta-language. We show that:

- Much of the work of exploring the nuances of a type system for a new language can be assisted by using mechanized tools – a generic meta-language.
- Such tools need not be much more complicated than your favorite functional language (Haskell), and are thus within the reach most language researchers.
- The automation helps language designers visualize the consequences of their design choices quickly, and thus helps speed the design process.
- The artifacts created by this exploration are machine checked proofs, and are hence less subject to error than proofs constructed by the more traditional approach.

5.1 Object Languages

In a meta-programming system meta-programs manipulate object-programs. Meta-programs may construct object-programs, combine object-program fragments into larger object-programs, observe the structure and other properties of object-programs, and execute object-programs to obtain their values.

There are several important kind of meta-programming scenarios: program generators, and program analyses. Each of these scenarios has a number of distinguishing characteristics

A program generator (a meta-program) solves a particular problem by constructing another program (an object-program) that solves the problem at hand. Usually the generated (object) program is "specialized" for the particular problem and uses less resources than a general purpose, non-generator solution.

A program analysis (a meta-program) observes the structure and environment of an object-program and computes some value as a result. Results can be data- or control-flow graphs, or even another object-program with properties based on the properties of the source object-program. Examples of these kind of meta-systems are: program transformers, optimizers, and partial evaluation systems.

A language model (a meta-program) gives meaning to, and points out properties of an object-language. Examples of these include type systems, type judgments, denotational and operational semantics, and small-step semantics.

5.2 Representing Object Programs

Meta-programs must represent object-programs as data. Object program representations usually fall into one of three categories. (1) Strings, (2) Algebraic datatypes, or (3) Quasi-quote systems. Other representations (as graphs for example) are possible, but not widespread.

With the string encoding, we represent the code fragment f (x,y) simply as `"f(x,y)"`. While constructing and combining fragments represented by strings can be done concisely, deconstructing them is quite verbose, and in essence degenerates into a parsing problem. More seriously, there is no automatically verifiable guarantee that programs thusly constructed are syntactically correct. For example, `"f(,y)"` can have the static type string, but this clearly does not imply that this string represents a syntactically correct program.

5.3 Object-programs as Algebraic Datatypes

With the Algebraic datatype encoding, we can address the syntactic correctness problem. A datatype encoding is essentially the same as what is called abstract syntax or parse trees. The encoding of the fragment `"plus(x,y)"` in an Ωmega datatype might be:
`Apply Plus (Tuple [Variable "x" ,Variable "y"])`
using a datatype declared as follows:

```
data Exp:: *0 where
  Variable:: String  -> Exp    -- x
  Constant:: Int -> Exp        -- 5
  Plus:: Exp                   -- plus
```

```
Less:: Exp                    -- less
Apply:: Exp -> Exp -> Exp     -- Apply Plus (x,y)
Tuple:: [Exp] -> Exp          -- (x,y)
```

Using a datatype encoding has an immediate benefit: correct typing for the meta-program ensures correct syntax for all object-programs. Because Ωmega(like most functional languages) supports pattern matching over datatypes, deconstructing programs becomes easier than with the string representation. However, constructing programs is now more verbose because we must use the cumbersome constructors like Variable, Apply, and Tuple.

5.4 Representing Programs using Quasi-quotes

The Quasi-quote representation is an attempt to represent object-programs without cumbersome constructor functions. Here the actual representation of object-code is hidden from the user by the means of a quotation mechanism. Object code is constructed by placing "quotation" annotations around normal code fragments. The quasi-quotation approach is the approach used in MetaML, Template Haskell, and the staged fragment of Ωmega.

In the staged fragment of Ωmega (see Section 3.11) the quasi-quotations are called staging annotations, and include Brackets `[| |]` and Escape `$`. An expression `[| e |]` is a quotation, and it builds the code representation of `e` (a data structure); `$(e)` is an anti-quotation, and splices the code obtained by evaluating `e` into the body of a surrounding bracketed expression (embedding one data structure into another). The quotation and anti-quotation mechanism abstracts the actual data-type representing code.

In a quasi-quoted system, the meta-language may now enforce the type-correctness of the object language as well as the meta-language, and avoid the problems associated with a constructor based approach. The major disadvantages of quasi-quoted systems are

- Their is usually only a single object-language, and it must be built into the meta-language.
- The quasi-quote mechanism is great for constructing code, but less useful for taking code apart, especially code with binding constructs.
- The type system of the meta-language must be aware of the type system of the object language. Usually this is accomplished by making the meta-language and the object-language the same language. Heterogeneous quasi-quote systems are rare because of this.

In the remainder of this section, we eschew the quasi-quote mechanism in favor of using GADTs in an effort to address these disadvantages.

5.5 Interpreters in a Typed Meta-language

Often one would like to build an interpreter or evaluation function for an object-language. In a typed meta-language, it is necessary to define a `Value` domain, that is a labeled sum of all the possible result types of evaluating an expression. In the `Exp` type above this would include both integers and booleans (as these are the types of the ranges of the functions `Plus` and `Less`), as well as functions and tuples.

```
data Value :: *0 where
  IntV:: Int -> Value
  BoolV:: Bool -> Value
  FunV:: (Value -> Value) -> Value
  TupleV :: [Value] -> Value
```

The evaluation function is then a case analysis over the structure of terms, recursively evaluating sub-terms into values, and then combining the sub-values into answer values.

```
eval:: (String -> Value) -> Exp -> Value
eval env (Variable s) = env s
eval env (Constant n) = IntV n
eval env Plus = FunV plus
  where plus (TupleV[IntV n ,IntV m]) = IntV(n+m)
eval env Less = FunV plus
  where plus (TupleV[IntV n ,IntV m]) = BoolV(n < m)
eval env (Apply f x) =
  case eval env f of
    FunV g -> g (eval env x)
eval env (Tuple xs) = TupleV(map (eval env) xs)
```

The key observation is – There is considerable overhead in such a function. It must first interpret the structure of the expressions, and it must perform quite a bit of tagging and un-tagging by applying the `Value` constructors (`IntV`, `BoolV`, `FunV`, and `TupleV`), and deconstructing them when appropriate.

5.6 Staging an Interpreter

We may remove the interpretive overhead by using staging. Like the evaluation function, the staged evaluation function is a case analysis over the structure of terms, recursively evaluating sub-terms into code values, and then splicing the smaller code values into larger code values.

```
stagedEval:: (String -> Code Value) -> Exp -> Code Value
stagedEval env (Variable s) = env s
stagedEval env (Constant n) = lift(IntV n)
stagedEval env Plus = [| FunV plus |]
  where plus (TupleV[IntV n ,IntV m]) = IntV(n+m)
stagedEval env Less = [| FunV less |]
  where less (TupleV[IntV n ,IntV m]) = BoolV(n < m)
stagedEval env (Apply f x) = [| apply $(stagedEval env f) $(stagedEval env x) |]
  where apply (FunV g) x = g x
stagedEval env (Tuple xs) = [| TupleV $(mapLift (stagedEval env) xs) |]
  where mapLift f [] = lift []
        mapLift f (x:xs) = [| $(f x) : $(mapLift f xs) |]
```

We may observe the result of staging by applying `stagedEval` to an actual `Exp`.

```
exp1 = Apply Plus (Tuple [Variable "x" ,Variable "y"]) -- (+)(x,y)

ans = stagedEval f exp1
  where f "x" = lift(IntV 3)
        f "y" = lift(IntV 4)


ans = [| %apply (%FunV %plus) (%TupleV [IntV 3,IntV 4]) |] : Code Value
```

We have removed the interpretive overhead, but the tagging and untagging overhead remains. This overhead is a by caused by using a disjoint sum as the range of the evaluator, which is necessary in a typed meta-language. This not the only problem when using algebraic datatypes to encode object-languages in a strongly typed meta-language like Haskell. The algebraic datatype approach to encoding object-languages does not track the type correctness of the object-program. We will fix both these problems by representing object-programs using GADTs rather than Algebraic datatypes.

5.7 Typed Object-languages using GADTS

GADTs allow us to build datatypes indexed by another type. We can use the GADT to represent object programs (just as we use algebraic datatype to represent object programs), but we may also use the type index to represent the type of the object-language program being represented. A simple typed object-language example is:

```
data Term:: *0 ~> *0 where
  Const :: Int -> Term Int              -- 5
  Add:: Term ((Int,Int) -> Int)         -- (+)
```

```
  LT:: Term ((Int,Int) -> Bool)          -- (<)
  Ap:: Term(a -> b) -> Term a -> Term b  -- (+) (x,y)
  Pair:: Term a -> Term b -> Term(a,b)   -- (x,y)
```

Above we introduced the new type constructor `Term`, which is a representation of a simple object-language of constants, pairs, and numeric operators. `Term`s are a typed object-language representation, i.e. a data structure that represents terms in some object-language. The meta-level type of the representation, i.e. the `a` in (`Term a`), indicates the type of the object-level term. This is made possible by the flexibility of the GADT mechanism. Using typed object-level terms, it is impossible to construct ill-typed term representations, because the meta-language type system enforces this constraint.

```
ex1 :: Term Int
ex1 = Ap Add (Pair (Const 3) (Const 5))

ex2 :: Term (Int,Int)
ex2 = Pair ex1 (Const 1)
```

Attempting to construct an ill-typed object term, like `(Ap (Const 3) (Const 5))`, causes a meta-level (Ωmega) type error. Another advantage of using GADTs rather than ADTs is that it is now possible to construct a tagless[19, 35, 34] interpreter directly:

```
evalTerm :: Term a -> a
evalTerm (Const x) = x
evalTerm Add = \ (x,y) -> x+y
evalTerm LT = \ (x,y) -> x<y
evalTerm (Ap f x) = evalTerm f (evalTerm x)
evalTerm (Pair x y) = (evalTerm x,evalTerm y)
```

In a language without GADTs, as we illustrated in Section 5.7, we would need to employ universal value domain like `Value`. See [18] for a detailed discussion of this phenomena. Such a tagless interpreter has the structure of a large step (or operational) semantics. If the `eval` function is total and well-typed at the meta-level, it implies that the object-level semantics (defined by `eval`) is also well-typed. Every well-typed object level term evaluates to a well-formed value.

Exercise 23. In the object-languages we have seen so far, there are no variables. One way to add variables to a typed object language is to add a variable constructor tagged by a name and a type. A singleton type representing all the possible types of a program term is necessary. For example we may add a `Var` constructor as follows (where the `Rep` is similar to the `Rep` type from Exercise 9).

```
data Term:: *0 ~> *0 where
  Var:: String -> Rep t -> Term t        -- x
  Const :: Int -> Term Int               -- 5
  . . .
```

Write a GADT for `Rep`. Now the evaluation function for `Term` needs an environment that can store many different types. One possibility is use existentially quantified types in the environment as we did in Exercise 21. Something like:

```
type Env = [exists t . (String,Rep t,t)]

eval:: Term t -> Env -> t
```

Write the evaluation function for the `Term` type extended with variables. You will need a function akin to `sameNat` from Exercise 16, except it will have type: `sameRep:: Rep a -> Rep b -> Maybe(Equal a b)`

Exercise 24. Another way to add variables to a typed object language is to reflect the name and type of variables in the meta-level types of the terms in which they occur. Consider the GADTs:

```
data VNum:: Tag ~> *0 ~> Row Tag *0 ~> *0 where
  Zv:: VNum l t (RCons l t row)
  Sv:: VNum l t (RCons a b row) -> VNum l t (RCons x y (RCons a b row))
 deriving Nat(u)

data Exp2:: Row Tag *0 ~> *0 ~> *0 where
  Var:: Label v -> VNum v t e -> Exp2 e t
  Less:: Exp2 e Int -> Exp2 e Int -> Exp2 e Bool
  Add:: Exp2 e Int -> Exp2 e Int -> Exp2 e Int
  If:: Exp2 e Bool -> Exp2 e t -> Exp2 e t -> Exp2 e t
```

What are the types of the terms `(Var ‘x 0u)`, `(Var ‘x 1u)`, and `(Var ‘x 2u)`. Now the evaluation function for `Exp2` needs an environment that stores both integers and booleans. Write a datatype declaration for the environment, and then write the evaluation function. One way to approach this is to use existentially quantified types in the environment as we did in Exercises 21 and 23. Better mechanisms exist. Can you think of one?

5.8 Taggless Staged Interpreters

By staging a object-level type indexed GADT we can remove both the interpretive and tagging overhead.

```
stagedEvalTerm :: Term a -> Code a
stagedEvalTerm (Const x) = lift x
stagedEvalTerm Add = [| add |]
  where add (x,y) = x+y
stagedEvalTerm LT = [| less |]
  where less (x,y) = x < y
stagedEvalTerm (Ap f x) = [| $(stagedEvalTerm f) $(stagedEvalTerm x) |]
```

```
stagedEvalTerm (Pair x y) = [|($(stagedEvalTerm x),$(stagedEvalTerm y))|]

ex2 = (Pair (Ap Add (Pair (Const 3) (Const 5))) (Const 1))
```

We can stage a program like `ex2` by applying `stagedEvalTerm` to produce some code. For `ex2` we get: `[| (add (3, 5), 1) |]`. Note that both the interpretive overhead, and the tagging overhead, have been completely removed.

Exercise 25. A staged evaluator is a simple compiler. Many compilers have an optimization phase. Consider the term language with variables from Exercise 23.

```
data Term:: *0 ~> *0 where
  Var:: String -> Rep t -> Term t
  Const :: Int -> Term Int               -- 5
  Add:: Term ((Int,Int) -> Int)          -- (+)
  LT:: Term ((Int,Int) -> Bool)          -- (<)
  Ap:: Term(a -> b) -> Term a -> Term b  -- (+) (x,y)
  Pair:: Term a -> Term b -> Term(a,b)   -- (x,y)
```

Can you write a well-typed staged evaluator the performs optimizations like constant folding, and applies laws like $(x + 0) = x$ before generating code?

5.9 A Typed-Object Language with Binding.

Object languages with variables and binding structures are harder to represent in a way that reflects the type of the object-language term in the type of its meta-language representation.

This is because if we change the type of the object-level variables, the type of the whole object-level term may also change. The key to this dilemma is to represent the type of the free variables in a term, as well as the type of the term, in the type of its meta-level representation. We do this by indexing terms by two indexes: first, the terms object-level type, and second, a type level structure encoding the environment (i.e. a mapping from variables to their types) in which the term has that type.

If we represent variables by labels (see Section 3.14), we can represent the environment by a row. A `Row` is nothing more than a list like structure (storing pairs of elements at each "cons" node) at the type level (see Exercise 19).

```
data Row :: a ~> b ~> *1 where
   RNil :: Row x y
   RCons :: x ~> y ~> Row x y ~> Row x y
 deriving Record(r)
```

For example the type: `(RCons 3t Int RNil)` is classified by `(Row Nat *0)`. Note that we have defined a syntactic extension for rows tagged by `r`. Thus `(RCons 3t Int RNil)` will display as `{3t=Int}r`. An environment is just a type classified by `(Row Tag *0)`. We define a new value level type, `Lam`, indexed by environments (represented by `(Row Tag *0)`) and types (represented by *0).

```
data Lam:: Row Tag *0 ~> *0 ~> *0  where
  Var   :: Label s -> Lam (RCons s t env) t
  Shift :: Lam env t -> Lam (RCons s q env) t
  Abs   :: Label a -> Lam (RCons a s env) t -> Lam env (s -> t)
  App   :: Lam env (s -> t) -> Lam env s -> Lam env t
```

The first index to `Lam`, is a Row tracking its variables, and the second index, tracks the object-level type of the term. For example a term with variables `x` and `y` might have type `Lam {‘x:Int, ‘y:Bool; u}r Int`.

The key to this approach is the typing of the constructor functions for variables (`Var`) and lambda expressions (`Abs`). Consider the `Var` constructor function. To construct a variable we simply apply `Var` to a label, and its type reflects this. For example here is the output from a short interactive session with the *Ω*mega interpreter.

```
prompt> Var ‘name
(Var ‘name) : forall a (b:Row Tag *0).Lam {‘name=a; b}r a

prompt> Var ‘age
(Var ‘age) : forall a (b:Row Tag *0).Lam {‘age=a; b}r a
```

Variables are really De Bruijn-like in their behavior. Variables created with `Var` all have index level 0. The two examples have different names in the same index position, and they would clash if they were both used in the same lambda term. To shift the position of variable to a different index, we use the constructor `Shift:: Lam a b -> Lam {c=d; a}r b` (see Exercise 23 for an alternative mechanism to distinguish variables). To define two variables `x` and `y` for use in the same environment we shift one of them into a different index. We type a few examples at the *Ω*mega top-level loop to illustrate the this.

```
prompt> Var ‘x
(Var ‘x) : Lam {‘x=a; b}r a

prompt> Shift(Var ‘y)
(Shift (Var ‘y)) : Lam {a=b,‘y=c; d}r c

prompt> Shift (Shift (Var ‘z))
(Shift (Shift (Var ‘z))) : Lam {a=b,c=d,‘z=e; f}r e
```

A `Lam` term represented by `(Var ‘x)` has the tag `‘x` appearing as the first element in the environment row. By applying `Shift` once, the tag `‘x` is pushed into the second element of the row, a second Shift pushes it into the third element, etc.

The Abstract constructor, binds the first tag in the first element of the row, removing the tag and its associated type from the environment, and shifting the others towards the front of the environment.

```
prompt> App (Var `a) (Shift (Var `b))
(App (Var `a) (Shift (Var `b))) : Lam {`a=a -> b,`b=a; c}r b

prompt> Abs `f (Abs `x (App (Shift (Var `f)) (Var `x)))
(Abs `f (Abs `x (App (Shift (Var `f)) (Var `x)))) : Lam a ((b -> c) -> b -> c)
```

Note how terms with free variables, have non-trivial environment indexes which mention their free variables. To build an evaluator for an object-level typed term, we will need values for each free variable in the term. We can package up a set of these values using a record.

A Record structure is a labeled tuple. We can define such a value, and type it using the Row defined above.

```
data Record :: Row Tag *0 ~> *0 where
    RecNil :: Record RNil
    RecCons :: Label a -> b -> Record r -> Record (RCons a b r)
  deriving Record()
```

Note that we have defined a syntactic extension for records tagged by the empty tag. Thus we may use the record syntax (with no tag) to build records.

```
prompt> {`a=34,`b="abc"}
{`a=34,`b="abc"} : Record {`a=Int,`b=[Char]}r
```

5.10 A Tagless Interpreter for a Language with Variables

The typed-object language Lam can be supplied with a typed evaluation function. The key is to supply a record that supplies exactly the values necessary for the free variables in the term being evaluated. The type system ensures that the record and the free variables coincide.

```
evalLam:: Record r -> Lam r t -> t
evalLam (RecCons _ v r) (Var _)        = v
evalLam RecNil          (Var _)        = unreachable
evalLam (RecCons _ _ r) (Shift e)      = evalLam r e
evalLam RecNil          (Shift _)      = unreachable
evalLam env             (Abs lab body) = \ x -> evalLam (RecCons lab x env) body
evalLam env             (App f x)      = (evalLam env f) (evalLam env x)
```

Exercise 26. Instead of using `Var` and `Shift`, fold the ideas from Exercise 24 into the `Lam` datatype, and then write the evaluation function for this GADT.

5.11 A Staged Interpreter for a Language with Variables

It is even possible to stage such an interpreter. One complication is that the record does not contain values but code. To enable this we define the staged record.

```
data StaticRecord:: Row Tag *0 ~> *0 where
  StNil :: StaticRecord RNil
  StCons:: Label t -> Code x -> StaticRecord r -> StaticRecord (RCons t x r)

stageLam:: StaticRecord r -> Lam r t -> Code t
stageLam (StCons _ code r) (Var _)        = code
stageLam StNil             (Var _)        = unreachable
stageLam (StCons _ _ r)    (Shift e)      = stageLam r e
stageLam StNil             (Shift _)      = unreachable
stageLam env               (App f x)      =
   [| $(stageLam env f) $(stageLam env x) |]
stageLam env               (Abs lab body) =
   [| \ x -> $(stageLam (StCons lab [|x|] env) body) |]
```

5.12 Small step semantics.

The datatype declarations for representing well-typed terms in the previous sections bear a striking similarity to the typing judgments for the those languages. For example consider:

$$\frac{}{\Gamma, x{:}\tau \vdash x : \tau}\textsc{Var} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma, x{:}\sigma \vdash e : \tau}\textsc{Shift} \qquad \frac{\Gamma, x{:}\tau \vdash e : \sigma}{\Gamma \vdash \lambda.e : \tau \to \sigma}\textsc{Abs}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \sigma \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \sigma}\textsc{Abs}$$

The similarity justifies a slight change in perspective. We have been thinking of `Lam` as representing a piece of abstract syntax, but we may also think of it as representing a typing derivation.

The latter perspective supports an interesting approach to studying the meta-theory of object languages. A typing derivation is a *proof* that a given term has a given type in a given context. So total functions that transform proofs into other proofs can be considered as constructive proofs of results in the meta-theory of our object language. This is the approach taken in Twelf, where the meta-language is a Prolog-style logic language. In Ωmega, we can write our meta-programs in a functional programming style.

In the remainder of this section, we build up to a proof of type soundness for our little language. Our proof is has the following basic structure (due to Wright and Felleisen)[39].

1. All terms are categorized syntactically as either values or non-values.
2. A reduction relation $e \rightarrow e'$ comprising a small-step operational semantics is given.
3. Any non-value term that cannot be reduced any further is considered to exhibit a run-time error.
4. **Progress.** Any well-typed term e is either a value or can step to another well-typed term e' ($e \rightarrow e'$).
5. **Preservation.** The reduction relation preserves types: If e has type τ and $e \rightarrow e'$, then e' has type τ.
6. Therefore if a term is well-typed, and we reduce it until no more reduction steps are possible, then the resulting term must be a value (rather than a term exhibiting a run-time error).

Values versus computations. The first step to proving type-soundness in Ωmega by this method is to distinguish between values and non-values. We accomplish this by adding another index to the abstract syntax (or typing derivation) datatype.

```
data Mode:: *1 where
 Exp:: Mode
 Val:: Mode

data E:: Mode ~> Row Tag ObjType ~> ObjType ~> *0  where
  Const:: Rel a b -> b -> E Val env a
  Var  :: Label s -> E Val (RCons s t env) t
  Shift:: E m env t -> E m (RCons s q env) t
  Lam  :: Label a -> E m (RCons a s env) t -> E Val env (ArrT s t)
  App  :: E m1 env (ArrT s t) -> E m2 env s -> E Exp env t
```

Note, we now view the new GADT, `E`, as both a typing derivation, and a representation of an object-level term. It is indexed by 3 kinds.

- `Mode`. The mode of the term. Either a `Val`, a term in normal form, or an `Exp`, a term with redex.
- `Row Tag ObjType`. The environment which indicates the position and type of the free variables in the term.

– `ObjType`. The object-level type of the term.

Note how terms in normal form have types where `Val` is the first index, and ones with redexes have types where `Exp` is the first index. Consider the short Ωmega session:

```
prompt> Const IntR 3
(Const IntR 3) : E Val a IntT

prompt> Lam ‘x (Var ‘x)
(Lam ‘x (Var ‘x)) : E Val a (ArrT b b)

prompt> App (Lam ‘x (Var ‘x)) (Const IntR 3)
(App (Lam ‘x (Var ‘x)) (Const IntR 3)) : E Exp a IntT
```

In `E`, we no longer use types of kind `*0` as object-level types. We do this because we wish to lift some, but not all, meta-level values into constants in the object-language. In this example we wish to lift integer constants, and n-ary functions over integers (the so-called δ-reductions). To accomplish this we define a new kind to represent object-level types.

```
data ObjType:: *1 where
 ArrT:: ObjType ~> ObjType ~> ObjType
 IntT:: ObjType
```

This new kind appears as the third index of `E`, and also as an index to the `Row` comprising the environment, the second index of `E`. The constructor `Const` lifts only those values classified by types that are related to some `ObjType` by the witness relation `Rel`.

```
data Rel:: ObjType ~> *0 ~> *0 where
 IntR:: Rel IntT Int
 IntTo:: Rel b s -> Rel (ArrT IntT b) (Int -> s)
   -- First order functions only as constants
```

The structure of `Rel` relates only integers and first-order, n-ary functions over integers to the type `Obj-Type`. Consider the short Ωmega session:

```
prompt> IntR
IntR : Rel IntT Int

prompt> IntTo IntR
(IntTo IntR) : Rel (ArrT IntT IntT) (Int -> Int)

prompt> IntTo (IntTo IntR)
(IntTo (IntTo IntR)) : Rel (ArrT IntT (ArrT IntT IntT)) (Int -> Int -> Int)
```

Finally, on occasion we will need to observe the structure of an object-level term, and compute whether it is a value in normal form, or a term with a redex. We do this by defining a singleton type reflecting the kind `Mode` into the value world, and by writing a total function that computes a safe approximation of the mode of any expression. By safe, we mean that no term is ever indexed by `Exp` if it is a value, though some terms might be indexed by `Exp` even though they do not contain a redex. Such terms generally have the form `(App (Var 'x) _)`, i.e. an application with a variable in the function part).

```
data Mode':: Mode ~> *0 where
  Exp':: Mode' Exp
  Val':: Mode' Val

mode :: E m e t -> Mode' m
mode (Lam v body) = Val'
mode (Var v) = Val'
mode (Const r v) = Val'
mode (Shift e) = mode e
mode (App _ _) = Exp'
```

There are to kinds of redexes in a term. β-redexes (an explicit λ-expression in the function position of an application) and δ-redexes (a higher-order constant in the function position of an application). We give meaning to β-redexes by the use of substitution. Thus we need a well-typed version of substitution over object-level terms represented by `E`.

Substitution lemma. The key lemma behind the preservation proof is type-soundness is called the *substitution lemma*. The lemma says that if a term e has type σ under the assumption that some variable x has type τ, then substituting any term e' of type τ for x in e yields $e[e'/x]$ of type σ. In our version of the preservation proof, the lemma exhibits itself as a total well-typed function that performs substitution.

We choose to represent substitutions as data structures. This provides another example of object language syntax because our syntax is similar to explicit substitutions [5]. In this approach a substitution of type `(Sub e1 e2)` is a mapping from one environment (of kind `e1`) to another (of kind `e2`).

```
data Sub:: Row Tag ObjType ~> Row Tag ObjType ~> *0 where
  Id:: Sub r r
  Bind:: Label t -> E m r2 x -> Sub r r2 -> Sub (RCons t x r) r2
  Push:: Sub r1 r2 -> Sub (RCons a b r1) (RCons a b r2)

subst:: E m1 r t -> Sub r s -> exists m2 . E m2 s t
subst t           Id          = Ex t
```

```
subst (Const r c) sub          = Ex (Const r c)
subst (Var v)     (Bind u e r) = Ex e
subst (Var v)     (Push sub)   = Ex (Var v)
subst (Shift e)   (Bind _ _ r) = subst e r
subst (Shift e)   (Push sub)   = case subst e sub of {Ex a -> Ex(Shift a)}
subst (App f x)   sub          = case (subst f sub,subst x sub) of
                                   (Ex g,Ex y) -> Ex(App g y)
subst (Lam v x)   sub          = case subst x (Push sub) of
                                   (Ex body) -> Ex(Lam v body)
```

Preservation. In our proof, we perform steps 2 (define the one-step evaluation relation), 4 (prove type preservation), and 5 (prove type preservation) at once by defining a total single-step operation that operates on well-typed non-value closed terms. Its type is given by
`onestep :: E m Closed t -> (E Exp Closed t + E Val Closed t)`.
Read logically this type says that every closed term (regardless of whether it is a value or an expression with a redex) can be transformed into another closed term with the same type, or is already a value.

```
type Closed = RNil

onestep :: E m Closed t -> (E Exp Closed t + E Val Closed t)
onestep (Var v)      = unreachable
onestep (Shift e)    = unreachable
onestep (Lam v body) = R (Lam v body)
onestep (Const r v)  = R(Const r v)
onestep (App e1 e2)  =
  case (mode e1,mode e2) of
    (Exp',_) ->
      case onestep e1 of
        L e -> L(App e e2)
        R v -> L(App v e2)
    (Val',Exp') ->
      case onestep e2 of
        L e -> L(App e1 e)
        R v -> L(App e1 v)
    (Val',Val') -> rule e1 e2
```

This function is a non-recursive case analysis. The `Var` and `Shift` cases are unreachable (they cannot be closed terms). The `Lam` and `Const` cases are already values. Observing the mode of the two parts of an application we have four choices. If the function is an expression with a possible redex, we take one step in the function part, and then rebuild the term. If function part is a value, we must apply one of the β- or δ-rules. Note that the function part is always a closed term with an `(ArrT _ _)` object-level type.

```
rule::  E Val Closed (ArrT a b) ->
```

```
        E Val Closed a ->
        (E Exp Closed b + E Val Closed b)
rule (Var _)    _ = unreachable
rule (Shift _) _ = unreachable
rule (App _ _) _ = unreachable
-- The beta-rule
rule (Lam x body) v =
  let (Ex term) = subst body (Bind x v Id)
  in case mode term of
       Exp' -> L term
       Val' -> R term
rule (Const IntR _)      _                    = unreachable
rule (Const (IntTo b) _) (Var _)              = unreachable
rule (Const (IntTo b) _) (Shift _)            = unreachable
rule (Const (IntTo b) _) (App _ _)            = unreachable
rule (Const (IntTo b) f) (Lam x body)         = unreachable
rule (Const (IntTo b) f) (Const (IntTo _) x) = unreachable
-- The delta-rule
rule (Const (IntTo b) f) (Const IntR x)      = R(Const b (f x))
```

There are twelve cases. Nine of which are unreachable from type considerations (i.e. the inputs are not values, are not closed, or the first argument does not have an arrow type). We have structured our function body to make it explicit that we have covered every case. This allows us to prove (by a meta-level argument) that `rule` is total. In other systems (i.e. Twelf, Coq, etc.) this argument can be enforced by the type-system of the meta-language. In these systems all functions are total (or they are not accepted). In Ωmega, we aspire to this level of automated assistance, but as we think of Ωmega as a programming language (not a proof system) we must support both total and partial functions. We hope to separate total and partial functions by using the type system sometime in the near future.

The function `onestep` makes progress. By inspecting the code we see all values are immediately returned, and all non-values actually take one step forward.

5.13 Example: Constructing Typing Derivations at Runtime

At first glance, using GADTs to represent object-languages solves many problems. But, further introspection reveals a subtle problem. We can build typed object-level terms by typing constructed terms into our program using the constructors of the GADT, but how do we build such terms algorithmically. I.e. how do we write a parser, for example, that builds a well-typed object-level term. What would the type of the parser be?

The type `(parse:: String -> E m e t)` is clearly not sufficient. Not every string can be parsed. But the type `(parse:: String -> Maybe(E m e t))` is also not sufficient. What mode, environment, and object level type should constrain the meta-level type variables `m`, `e`, and `t`? The type `(parse:: String -> exists m e t . Maybe(E m e t))` is closer to the mark, but this is also too unconstrained. We expect some properties to be true of these type variables. One solution is to build runtime representations that represent the constraints we envision, and runtime tests for these constraints, that we can execute at runtime.

We do this by building a singleton type to reflect the object-level types as meta-level runtime values (Section 3.6 and Exercise 9), and a runtime test for equality of these object-level type indexes (Exercises 16 and 23).

```
data Rep:: ObjType ~> *0 where
 I:: Rep IntT
 Ar:: Rep a -> Rep b -> Rep (ArrT a b)

compare:: Rep a -> Rep b -> (String + Equal a b)
compare I I = R Eq
compare (Ar x y) (Ar s t) =
  do { Eq <- compare x s
     ; Eq <- compare y t
     ; R Eq}
compare I (Ar x y) = L "I /= (Ar _ _)"
compare (Ar x y) I = L "(Ar _ _) /= I"
```

Here, because we want our runtime tests to report interesting error messages, the comparison returns a sum type, were the left injection (a failure) is an error message, and the right injection (a success) is an equality proof.

We will break our parsing problem into two parts. First, parsing a string into an untyped object-language representation (not shown in this report, as this is the ordinary parsing problem). Second, transforming this untyped representation into a well-typed GADT representing a typed object-language term (or typing derivation, depending upon your perspective). In this report, we assume every the untyped representation suggests a type for every variable, and that our algorithm checks that this suggestion is correct. The inference problem is much harder, and not shown here. Our untyped representation follows:

```
data Term:: *0 where
  C:: Int -> Term
  Ab:: String -> Rep a -> Term -> Term
  Ap:: Term -> Term -> Term
  V:: String -> Term
```

We will check each term with respect to a given environment which maps every variable to an object-level type. It will also store the string used to name the variable in the untyped representation, and the label used to represent the variable in the typed-representation. Such an environment is indexed by `(Row Tag ObjType)` in the same manner as terms `E` and substitutions `Sub`.

```
data Env:: Row Tag ObjType ~> *0 where
   Enil:: Env RNil
   Econs:: Label t -> (String,Rep x) -> Env e -> Env (RCons t x e)
 deriving Record(e)
```

A key component of our algorithm, to produce a well-typed representation from an untyped representation, is to lookup the type of a variable.

```
lookup:: String -> Env e -> (String + exists t m .(E m e t,Rep t))
lookup name Enil = fail ("Name not found: "++name)
lookup name {l=(s,t);rs}e | eqStr name s = R(Ex(Var l,t))
lookup name {l=(s,t);rs}e =
  do { Ex(v,t) <- lookup name rs
     ; R(Ex(Shift v,t)) }
```

If successful, both a representation of a type, and a term with that type are returned. Now we need put all this machinery together. The type checker is a program with the following type:
`tc:: Term -> Env e -> (String + exists t m . (E m e t,Rep t))`
Read logically, for every untyped term, and every environment with types for variables reflected in the row `e`, we can either report a type-checking error, or return a typed-term. In this typed term, its actual type, and its mode, are existentially quantified, but the actual object-level type is reflected in a runtime singleton object.

```
tc:: Term -> Env e -> (String + exists t m . (E m e t,Rep t))
tc (V s) env = lookup s env
tc (Ap f x) env =
  do { Ex(f',ft) <- tc f env
     ; Ex(x',xt) <- tc x env
     ; case ft of
        (Ar a b) ->
           do { Eq <- compare a xt
              ; R(Ex(App f' x',b)) }
        _ -> fail "Non fun in Ap" }
tc (Ab s t body) env =
  do { let (Hidden l) = newLabel s
     ; Ex(body',et) <- tc body {l=(s,t); env}e
     ; R(Ex(Lam l body',Ar t et)) }
tc (C n) env = R(Ex(Const IntR n,I))
```

The application case is the most interesting. First, recursively type check the function and argument, returning typed terms `f'` and `x'`, and reflected types `ft` and `xt`. If either of these fail, the monad syntax causes the whole function to fail. Test that the function argument is really a function, and then compare the domain with the type of the argument. Only if this succeeds, and we have a proof that the two types are equal, can the whole case succeed.

5.14 The bottom line.

The ability to define type indexed GADTs, and the ability to define new kinds, creates a rich playground for those wishing to explore the design of new languages. These features, along with the use of rank-N polymorphism (which is beyond the scope of this report) make Ωmega a better meta-language than Haskell. In order to explore the design of a new language one can proceed as follows:

– First, represent the object-language as a type indexed GADT. The indexes correspond to static properties of the program.
– The indexes can have arbitrary structure, because they are introduced as the type constructors of new kinds.
– The typed constructor functions of the object-language GADT define a static semantics for the object language.
– Meta programs written in Ωmega, manipulate object-language represented as data, and check and maintain the properties captured in the type indexes by using the meta-language type system. This lets us build and test type systems interactively.
– A dynamic semantics for the language can be defined by (1) writing either a large step semantics in the form of an interpreter or evaluation function, or by (2) writing a small step semantics in terms of substitution over the term language. In either case, the type system of the meta-language guarantees that these meta-level programs maintain object level type-safety.
– Normal operations such as pretty-printing and parsing functions can also be constructed, albeit with a little more cleverness that is ordinarily required.

6 Using Terms as Theorems

We can use a value of type `(Nat' n)` as a proof that `n` is a natural number. In Ωmega, ordinary datatypes can be used as constraints over types. A constraint can be discharged by exhibiting a non-divergent term with that type. RThe classic datatype used in this fashion is the equality type from Section 3.8. Recall:

```
data Equal :: a ~> a ~> *0 where
  Eq:: Equal x x
```

The `Equal` constraint can be applied to all types of the same kind because it is level polymorphic (see section 3.12). Thus `(Equal #2 #3)` and `(Equal Int Bool)` are both well formed, but neither is inhabited (i.e. there are no non-divergent values with these types since, (`Int` $\neq$ `Bool`) and ((`S(S Z))` $\neq$ `(S(S(S Z)))`)). The normal mode of use is to construct terms with types like (`Equal` x y) where x and `y` are type level functions applications. For example consider the function `plusZ`.

```
plusZ :: Nat' n -> Equal {plus n Z} n
plusZ Z = Eq
plusZ (S m) = Eq
  where theorem indHyp = plusZ m
```

This function is a proof by induction that for all natural numbers `n` : `{plus n #0}` = `n`. The definition exhibits a well-typed, total function with this type.

To see that `plusZ` 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 two clauses of the definition of `plusZ`.

1.

expected type	`Nat' n` $\longrightarrow$ `Equal {plus n Z} n`
equation	`plusZ Z` = `Eq`
computed type	`Nat' Z` $\longrightarrow$ `Equal a a`
equalities	`n = Z` $\Rightarrow$ `(a = n, a= {plus n Z})`

In the first case, the left-hand-side equalities let us assume `n` = `Z`. The right-hand-side equalities, require us to establish that `a` = `{plus n Z}` and `a` = `n`. This can be established *iff* `n` = `{plus n Z}`. Using the assumption that `n` = `Z`, we are left with the requirement that `Z` = `{plus Z Z}`, which is easy to prove using the definition of `plus`.

2.

expected type	`Nat' n` $\longrightarrow$ `Equal {plus n Z} n`
equation	`plusZ (S m)` $=$ `Eq`
computed type	`Nat' (S b)` $\longrightarrow$ `Equal a a`
equalities	`n = (S b)` $\Rightarrow$ `(a = n, a= {plus n Z})`

In the second case, the left-hand-side assumptions are `n` = `(S b)` (where the pattern introduced variable `m` has type `(Nat' b)`). The right-hand-side equalities, require us to establish that `a` = `{plus n Z}` and `a` = `n`. Again, this can only be established if `n` = `{plus n Z}`. Using the assumption that `n` = `(S b)`, we are left with the requirement that `(S b)` = `{plus (S b) Z}`. Using the definition of `plus`, this reduces to `(S b)` = `(S{plus b Z})`. To establish this fact, we use the inductive hypothesis. Since the argument `(S m)` is finitely constructed, and the function `plusZ` is total, the term, `(plusZ m)` exhibits a proof that `(Equal {plus b Z} b)`.

The declaration, `where theorem indHyp = plusZ m`, instructs the type checker to use the type of the term `(plusZ m)` as a reasoning rule. Thus we may assume its type: `(Equal {plus b #0} b)` while discharging `(Equal (S{plus b Z}) (S b))`. Other interesting facts, that are established in the same way, but omitted for brevity, include:

```
plusS :: Nat' n -> Equal {plus n (S m)} (S{plus n m})
plusCommutes :: Nat' n -> Nat' m -> Equal {plus n m} {plus m n}
plusAssoc :: Nat' n -> Equal {plus {plus n b} c} {plus n {plus b c}}
plusNorm :: Nat' x -> Equal {plus x {plus y z}} {plus y {plus x z}}
```

Exercise 27. Write an Ωmega function body for each of the prototypes above. The function bodies for `plusS` and `plusAssoc` are very similar to `plusZ`. The other two require appealing to theorems in addition to an induction hypotheses. In fact, `plusCommutes` requires both `plusZ` and `plusS` in addition to an induction hypothesis. I leave it to you to figure out what theorem is required for `plusNorm`.

6.1 Self Describing Combinatorial Circuits

Our next example is the description of combinatorial circuits. We will use types to ensure that our descriptions describe what they implement. We first describe the `Bit` type.

```
data Bit:: Nat ~> *0 where
  One :: Bit (S Z)
  Zero :: Bit Z
```

Like `Nat'`, `Bit` is a singleton type (see Section 3.6), there is only one value for each type. Note how the type of a bit, carries the value of the bit as a natural number as its type index. I.e. `(One :: Bit #1)` and `(Zero :: Bit #0)`. We exploit this to define a data structure representing a base-2 number as a sequence of bits. The idea is for a value of type `(Binary bit w v)` to represent a binary number represented by `w` bits, with width `w` and value `v`.

```
data Binary:: (Nat ~> *0) ~> Nat ~> Nat ~> *0 where
  Nil :: Binary bit Z Z
  Cons:: bit i -> Binary bit w n -> Binary bit (S w) {plus {plus n n} i}
```

A value with type `(Binary Bit 2t 3t)` is a sequence of `(Bit j)` values. The individual `j`'s are combined to represent a binary number with value `3t`. Binary numbers are stored least significant bit first. Prefixing a new bit, shifts the previous bits into the next significant position, so the value of the new number is the value of the new bit plus twice the value of the old bits. Thus the type expression `{plus {plus n n} i}` in the type of `Cons` which prefixes a new bit. For example consider the term: `(Cons Zero (Cons One (Cons Zero (Cons Zero Nil))))` that has type `(Binary Bit 4t 2t)`. I.e. "0100" (where the least significant bit is leftmost) has value 2 and width 4.

Note, that the type of the elements in the sequence has been abstracted to be any type constructor classified by the kind `(Nat ~> *0)`. Because in the above examples, we constructed a list of (*Bit* `i`) we got a `(Binary` *Bit* `len value)` as a result. In the sequel we will build binary numbers from other representations of bits.

If we add three one bit numbers, we always get a two bit result. We can write this function as follows.

```
add3Bits:: (Bit i) -> (Bit j) -> (Bit k) -> Binary Bit #2 {plus {plus j k} i}
add3Bits Zero Zero Zero = Cons Zero (Cons Zero Nil)
add3Bits Zero Zero One  = Cons One  (Cons Zero Nil)
add3Bits Zero One  Zero = Cons One  (Cons Zero Nil)
add3Bits Zero One  One  = Cons Zero (Cons One  Nil)
add3Bits One  Zero Zero = Cons One  (Cons Zero Nil)
add3Bits One  Zero One  = Cons Zero (Cons One  Nil)
add3Bits One  One  Zero = Cons Zero (Cons One  Nil)
add3Bits One  One  One  = Cons One  (Cons One  Nil)
```

This function is an exhaustive case analysis of all 8 possible combination of bits. It is exhaustive and total. Consider type checking one case,

expected type	`Bit i -> Bit j -> Bit k` → `Binary Bit 2t {plus {plus j k} i}`
equation	`add3Bits Zero One One` = `Cons Zero (Cons One Nil)`

computed type	`Bit 0t -> Bit 1t -> Bit 1t` → `Binary Bit 2t`
	`{plus {plus {plus {plus 0t 0t} 1t}`
	`{plus {plus 0t 0t} 1t} }`
	`0t }`

equalities	`(i = 0t,j = 1t,k = 1t)` ⇒ `{plus {plus j k} i} =`
	`{plus {plus {plus {plus 0t 0t} 1t}`
	`{plus {plus 0t 0t} 1t} }`
	`0t }`

Under the assumptions, both parts of the equality in the requirements for the right-hand-side reduce to `(Binary Bit #2 #2)`, so the clause is well typed. Iterating `add3Bits`, we can construct a ripple carry adder, whose type states that it is really an addition function!

```
add :: Bit c ->
       Binary Bit n i ->
       Binary Bit n j -> Binary Bit (S n) {plus {plus i j} c}
add c Nil Nil = Cons c Nil
add c (Cons x xs) (Cons y ys) =
  case add3Bits c x y of
    (Cons bit (Cons c2 Nil)) -> Cons bit (add c2 xs ys)
       where theorem plusCommutes, plusAssoc, plusNorm
```

The function `add` is type checked in the same manner as we illustrated with `plusZ` and `add3Bits`. In `add`, the type checker relies on the three theorems `plusCommutes`, `plusAssoc`, `plusNorm`.

```
plusCommutes :: Equal {plus n m}         {plus m n}
plusAssoc    :: Equal {plus {plus n b} c} {plus n {plus b c}}
plusNorm     :: Equal {plus x {plus y z}} {plus y {plus x z}}
```

When used in conjunction, as a set of left-to-right rewriting rules, they have a strong normalizing effect. This effect occurs because the theorems `plusCommutes` and `plusNorm` are only applied if the rewritten term is lexigraphically smaller than the original term. For example, while type checking `add` the type checker uses them to repeatedly rewrite the term:

`{plus {plus {plus {plus x3 x3} x2} {plus {plus x5 x5} x4}} x1}`

to the term:

`{plus x1 {plus x2 {plus x3 {plus x3 {plus x4 {plus x5 x5}}}}}}`

Exercise 28. Re-do the progression from defining the GADT `Binary` through defining the function `add`, but this time make `Binary` store most-significant bits on the left.

6.2 Symbolically Combining Bits

While we have shown how to use types to describe properties of programs, our adder is not a very effective hardware description. We need a data structure that can represent not only the constant bits, `One` and `Zero`, but also operations on bits. This motivates `BitX` (for eXtended bit).

```
data BitX:: Nat ~> *0 where
  OneX :: BitX (S Z)
  ZeroX :: BitX Z
  And:: BitX i -> BitX j -> BitX {and i j}
  Or:: BitX i -> BitX j -> BitX {or i j}
  Xor:: BitX i -> BitX j -> BitX {xor i j}
```

In order to track the result of *and*ing (*or*ing, *xor*ing) two bits, we need the `and` (`or`, `xor`) functions at the type level. These functions take any two natural numbers as input, but always return `0t` or `1t` as a result.

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

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

Exercise 29. Write the Ωmega type-level function classified by the kind: `xor :: Nat ~> Nat ~> Nat`, that implements the exclusive-or function.

We can prove a number of interesting theorems about these functions by exhibiting terms with logical types. As we did with `add3Bits`, these functions are basically an exhaustive analysis of the cases. Here we prove that `and` is associative.

```
andAs :: Bit a -> Bit b -> Bit c ->
         Equal {and {and a b} c} {and a {and b c}}
andAs Zero Zero Zero = Eq
andAs Zero Zero One  = Eq
andAs Zero One  Zero = Eq
andAs Zero One  One  = Eq
andAs One  Zero Zero = Eq
andAs One  Zero One  = Eq
andAs One  One  Zero = Eq
andAs One  One  One  = Eq
```

Note, that this is a theorem about `Bit a`, `Bit b`, and `Bit c`, not about natural numbers `a`, `b`, and `c`. I.e.

```
(Bit a -> Bit b -> Bit c -> Equal {and {and a b} c} {and a {and b c}})
```

is a theorem but

```
(Nat' a -> Nat' b -> Nat' c -> Equal {and {and a b} c} {and a {and b c}}}
```

is not. A number of other useful theorems are proved in a similar manner.

```
andZ1:: Bit a -> Equal {and a Z} Z
andZ2:: Bit a -> Equal {and Z a} Z
andOne2:: Bit a -> Equal {and a (S Z)} a
andOne1:: Bit a -> Equal {and (S Z) a} a
```

Exercise 30. Following the pattern of `AndAs`, write function definitions for the above prototypes.

Every `(BitX i)` can be evaluated into a `(Bit i)`, by applying the definitions of the operations `and`, `or` and `xor`. This is the purpose of the function `fromX`. Since the operations are functions are at the type level, and we need operations on bits (which live at the value level) we define the functions `and'`, `or'` and `xor'`.

```
fromX :: BitX n -> Bit n
fromX OneX = One
fromX ZeroX = Zero
fromX (Or x y) = or' (fromX x) (fromX y)
fromX (And x y) = and' (fromX x) (fromX y)
fromX (Xor x y) = xor' (fromX x) (fromX y)
fromX (And3 x y z) =
      and' (fromX x) (and' (fromX y) (fromX z))
```

```
and' :: Bit i -> Bit j -> Bit {and i j}
and' Zero Zero = Zero
and' Zero One = Zero
and' One Zero = Zero
and' One One = One

or' :: Bit i -> Bit j -> Bit {or i j}
xor' :: Bit i -> Bit j -> Bit {xor i j}
```

Exercise 31. Write Ωmega function bodies for the omitted functions `or'` and `xor'`.

Because every `(BitX i)` can be evaluated into a `(Bit i)`, we can lift theorems about `Bit` to theorems about `BitX`. For example:

```
andAs:: Bit a -> Bit b -> Bit c -> Equal {and {and a b} c} {and a {and b c}}

andAssoc:: BitX a -> BitX b -> BitX c -> Equal {and {and a b} c} {and a {and b c}}
andAssoc a b c = andAs (fromX a) (fromX b) (fromX c)
```

So unlike `andAs`, where we could not lift a theorem about `Bit` to a theorem about `Nat`, every theorem about `Nat` can be lifted to a theorem about `Bit`. With these tools, we can build a ripple carry adder that performs addition by applying the bit operations. For example, to add three one bit numbers to obtain a two bit result, we need to construct a logical formula that captures the following table.

```
inputs     sum

i j k      high bit  low bit
------
0 0 0      0         0
0 0 1      0         1
0 1 0      0         1          low bit  = (Xor i (Xor j k))
0 1 1      1         0          high bit = (Or (And i j)
1 0 0      0         1                         (Or (And i k)
1 0 1      1         0                             (And j k)))
1 1 0      1         0
1 1 1      1         1
```

To implement this is Ωmega, we introduce a 2-bit number `Pair` (most significant bit on the left), and the function `addthree`.

```
data Pair:: Nat ~> *0 where
 Pair:: BitX hi -> BitX lo -> Pair {plus {plus hi hi} lo}

addthree :: BitX i -> BitX j -> BitX k -> Pair {plus j {plus k i}}
addthree i j k = Pair (Or (And i j) (Or (And i k) (And j k)))
                          (Xor i (Xor j k))
  where theorem lemma = logic3 (fromX i) (fromX j) (fromX k)
```

Unlike the function `add3Bits`, we cannot type check `addthree` by exhaustively enumerating all possible inputs because there are an infinite number of possible terms of type `(BitX i)` for each natural number `i`. But we can prove a lemma about `Bit` (which we can prove by exhaustive analysis) and then lift it to a theorem about `BitX`. This is the role of the term `(logic3 (fromX i) (fromX j) (fromX k))` in the `theorem` clause in `addthree`.

```
logic3 :: Bit i -> Bit j -> Bit k ->
          (Equal {plus {plus {or {and i j}
                                 {or {and i k} {and j k}}}
                             {or {and i j}
                                 {or {and i k} {and j k}}}}
                       {xor i {xor j k}}}
                 {plus j {plus k i}})
logic3 Zero Zero Zero = Eq
```

```
logic3 Zero Zero One  = Eq
logic3 Zero One  Zero = Eq
logic3 Zero One  One  = Eq
logic3 One  Zero Zero = Eq
logic3 One  Zero One  = Eq
logic3 One  One  Zero = Eq
logic3 One  One  One  = Eq
```

We can now re-implement our ripple carry adder, but this time by symbolically combining the input bits, to compute the output bits as a logical function of the inputs. This function has a similar type, the same structure, and uses the same theorems as the function `add`.

```
addBits :: BitX c -> Binary BitX n i -> Binary BitX n j ->
           Binary BitX (S n) {plus {plus i j} c}
addBits c Nil Nil = Cons c Nil
addBits c (Cons x xs) (Cons y ys) =
  case addthree c x y of
    (Pair c2 bit) -> Cons bit (addBits c2 xs ys)
       where theorem plusCommutes, plusAssoc, plusNorm
```

To actually compute a circuit we need to have some symbolic inputs. We do this by extending the type `BitX` with a constructor to represent variables.. We can then construct some inputs, and compute the description of an adder. Our function works on inputs of any size.

```
data BitX:: Nat ~> *0 where
  . . .
  X:: Int -> BitX a

xs :: Binary BitX #2 {plus {plus a a} b}
xs = Cons (X 1) (Cons (X 2) Nil)

ys :: Binary BitX #2 {plus {plus a a} b}
ys = Cons (X 3) (Cons (X 4) Nil)
carry = (X 5)

ans = addBits carry xs ys
```

Here `xs` and `ys` are two bit symbolic inputs, and `carry` is a symbolic input carry. Calling `addBits` we construct an output which is a `(Binary Bit)` list with three elements, each of which is a combinatorial function of the input bits, whose value is guaranteed by the types to be the sum of the inputs! Below, we display the output with a pretty printer that displays `(X n)` as "xn", and indents the display to emphasize its structure.

```
(Cons (Xor x5
```

```
          (Xor x1 x3))
(Cons (Xor (Or (And x5 x1)
               (Or (And x5 x3)
                   (And x1 x3)))
           (Xor x2 x4))
(Cons (Or (And (Or (And x5 x1)
                   (Or (And x5 x3)
                       (And x1 x3)))
               x2)
          (Or (And (Or (And x5 x1)
                       (Or (And x5 x3)
                           (And x1 x3)))
                   x4)
             (And x2 x4)))
Nil)))
```

The key property here is that the type of this structure guarantees that it implements an addition function.

Exercise 32. There are many equivalencies between boolean expressions. Any a function with the type: `(BitX n -> Maybe (BitX n))` can be thought of as a meaning preserving transformation. Given a value typed `v:: BitX n`, a meaning preserving transformation returns `(Just u)` or `Nothing`. If it returns `(Just u)` then `u` is semantically equivalent to `v`. If it returns `Nothing` we interpret this to mean the transformation did not apply to `v`. Choose a few boolean laws and implement them as meaning preserving transformations as discussed above.

Exercise 33. Transformations can be combined by placing them in a list, and applying them using transformation combinators. Consider functions with the types below:

```
first:: [BitX n -> Maybe(BitX n)] -> BitX n -> Maybe(BitX n)
all:: [BitX n -> Maybe(BitX n)] -> BitX n -> [BitX n]
```

The combinator `first` lifts a list of transformations to a single transformation, applying the first applicable transformation in the list. The combinator `all` finds all applicable transformations and returns a list of all possible results, including the untransformed term as well. Define these two functions in Ωmega.

The combinator `retry` continually re-applies a meaning preserving transformation until the term reaches a fixed-point. What is the type of `retry`? Write an Ωmega function body for `retry`. What other combinators can you think of?

6.3 A Caveat

The addition of the variable `BitX` constructor `X` was necessary if we want to use our functions to build hardware descriptions. Without it, we can only build constant combinatorial circuits! Unfortunately, it breaks the soundness of our descriptions. The lack of soundness flows from the fact that our function `fromX` is no longer total. How do we turn a variable into a `Bit`? Thus, we can longer lift facts about the functions `and`, `or`, and `xor` and the type `Bit` to facts about the type `BitX`. To overcome this limitation we would need to track the variables in the type of `BitX` objects. For example we may write `(BitX Bit env width value)` as the type of a binary number whose free variables are described by `env`. Now, we must recast our theorems in terms of `(BitX Bit env width value)` and well formed environments `env`. This is sufficient, because a well formed environment means every variable will eventually be replaced by a bit, and in this new formulation the lifting of theorems hold.

Exercise 34. Using the patterns discussed in Section 5.9 for object- languages with binding structures, re-do the progression from the GADT `BitX` to the function `addBits`, but this time track the variables in the types of `BitX`. Recast the theorems about `BitX` so that they hold for all environments.

7 Conclusion

7.1 Relation to other systems

In order to make Ωmega 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. Ωmega 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 Ωmega a logic, in which our reasoning would be sound. This is the basis of our decision to make Ωmega 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 [9, 12], theorem provers (Coq [37], Isabelle [20]), logical frameworks (Twelf [22], LEGO [14]), and proof assistants (ALF [17], Agda [8]). 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 [15], Stump's Rogue-Sigma-Pi [33, 38], Xi and Pfenning's Dependent ML [42, 11], and Xi's Applied Type Systems [41, 7]. 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 *Ωmega* to program generic programs, since logical soundness is not an issue in this domain.

8 Acknowledgments

This report was originally prepared for the Central-European Functional Programming School held in Cluj, Romania, between 25-30 June, 2007. The text, code, and examples come from many sources. Here is quick (though, not by any means exhaustive) list.

1. Section 2 comes from the paper *Generic Programming in Ωmega*[31]. This paper has many additional examples, not covered in this report.
2. Section 3 also comes from the same paper, as well as the *Ωmega Users' Guide*[29], both of which have their roots in the paper *Putting Curry-Howard to Work*[28].
3. The shaped tree example with paths (Section 3.1) comes from discussions with James L. Caldwell from the University of Wyoming.
4. The exercise about Red-Black trees comes from some examples posted by Hong-Wei Xi for the ATS system.
   `http://www.cs.bu.edu/~hwxi/ATS/ATS.html`

5. The Leibniz equality GADT has its roots in two papers[13, 4] which introduce similar types with slight variants.
6. The material on using Ωmega as a meta-language comes from an unpublished paper called *Playing with Types*[27], though many of the examples used here are original to this report.
7. The combinatorial circuit examples come from a paper *Types and Hardware Description Languages*[30] prepared for the *Hardware design and Functional Languages* workshop held March 24-25 2007, in Braga, Portugal.
8. This material is based upon work supported by the National Science Foundation under Grant No. 0613969.

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. Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In S. Peyton Jones, editor, *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 157–166. ACM Press, 2002.
5. Zine-El-Abidine Benaissa, Daniel Briaud, Pierre Lescanne, and Jocelyne Rouyer-Degli. lambda-nu, A calculus of explicit substitutions which preserves strong normalisation. *J. Funct. Program*, 6(5):699–722, 1996.
6. Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.
7. Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *ICFP 2005*, 2005. `http://www.cs.bu.edu/~hwxi/`.
8. Catarina Coquand. Agda is a system for incrementally developing proofs and programs. Web page describing AGDA: `http://www.cs.chalmers.se/~catarina/agda/` .
9. T. Coquand and P. Dybjer. Inductive definitions and type theory: an introduction. (preliminary version). *Lecture Notes in Computer Science*, 880:60–76, 1994.
10. L. Damas. *Type assignment in programming languages.* PhD thesis, Edinburgh University CST-33-85, 1985.
11. 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.
12. P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. *Lecture Notes in Computer Science*, 1581:129–146, 1999.
13. Ralf Hinze and James Cheney. A lightweight implementation of generics and dynamics. In Manuel Chakravarty, editor, *Proceedings of the ACM SIGPLAN 2002 Haskell Workshop*, pages 90–104. ACM SIGPLAN, October 2002.
14. 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.

15. 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` .
16. Robin Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Science*, 17(3):348–375, 1978.
17. Bengt Nordstrom. The ALF proof editor, March 20 1996.
`ftp://ftp.cs.chalmers.se/pub/users/ilya/FMC/alfintro.ps.gz`.
18. Emir Pasalic and Nathan Linger. Meta-programming with typed object-language representations. In *Generative Programming and Component Engineering (GPCE'04)*, pages 136 – 167, October 2004. LNCS volume 3286.
19. Emir Pasalic, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP-02).*, pages 218–229, Pittsburgh, PA., October 4–6 2002. ACM Press.
20. Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.
21. Simon Peyton Jones. Special issue: Haskell 98 language and libraries. *Journal of Functional Programming*, 13, January 2003.
22. 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.
23. Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. *ACM SIGPLAN Notices*, 37(1):217–232, January 2002.
24. T. Sheard. Using MetaML: A staged programming language. *Lecture Notes in Computer Science*, 1608:207–239, 1999.
25. T. Sheard and S. Peyton-Jones. Template meta-programming for Haskell. In *Proc. of the workshop on Haskell*, pages 1–16. ACM, 2002.
26. Tim Sheard. Accomplishments and research challenges in meta-programming. In Walid Taha, editor, *Proceedings of the Workshop on Semantics, Applications and Implementation of Program Generation (SAIG'01)*, volume 2196 of *LNCS*, pages 2–44, Berlin, September 2001. Springer Verlag. Invited talk.
27. Tim Sheard. Playing with types. Technical report, Portland State University, 2005. `http://www.cs.pdx.edu/~sheard`.
28. Tim Sheard. Putting Curry-Howard to work. In *Proceedings of the ACM SIGPLAN 2005 Haskell Workshop*, pages 74–85, 2005.
29. Tim Sheard. Omega users' gude. Technical report, Portland Stage University, 2005-2007. Available at:
`http://web.cecs.pdx.edu/~sheard/Omega/index.html`.
30. Tim Sheard. Types and hardware description languages. In *Proceedings of the Hardware design and Functional Languages workshop*, March 2007. Available online at `http://web.cecs.pdx.edu/~sheard/`.
31. Tim Sheard. Generic programming in omega. *To appear in Lecture Notes in Computer Science*, ??, Notes from the Spring School on Datatype-Generic Programming 2006.
32. 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.

33. 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/`.
34. Walid Taha. Tag elimination - or - type specialisation is a type-indexed effect. In *APPSEM Workshop on Subtyping & Dependent Types in Programming. Ponte de Lima Portugal.*, July 2000. Available online at http://www-sop.inria.fr/oasis/DTP00/Proceedings/proceedings.html.
35. Walid Taha, Henning Makholm, and John Hughes. Tag elimination and jones-optimality. *Lecture Notes in Computer Science*, 2053:257–??, 2001.
36. Walid Taha and Tim Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2), 2000.
37. The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 7.4.* INRIA, 2003. `http://pauillac.inria.fr/coq/doc/main.html`.
38. 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/`.
39. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.
40. Hongwei Xi. *Dependent Types in Practical Programming.* PhD thesis, Carnegie Mellon University, 1997.
41. Hongwei Xi. Applied type systems (extended abstract). In *In post-workshop proceedings of TYPES 2003*, pages 394 – 408, 2004. Springer-Verlag LNCS 3085.
42. Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. *ACM SIGPLAN Notices*, 33(5):249–257, May 1998.
43. 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 Red-Black Tree Insertion

```
---------------------------------------------------------------------------
-- Introduce a new kind to represent colors

kind Color  = Red | Black

---------------------------------------------------------------------------
-- Top-level type that hides both
-- color of the node and tree height

data RBTree:: *0 where
 Root:: SubTree Black n -> RBTree

---------------------------------------------------------------------------
-- GADT that captures invariants

data SubTree:: Color ~> Nat ~> *0 where
 Leaf:: SubTree Black Z
```

```
 RNode:: SubTree Black n ->
         Int ->
         SubTree Black n ->
         SubTree Red n
 BNode:: SubTree cL m ->
         Int ->
         SubTree cR m ->
         SubTree Black (S m)

------------------------------------------------------------------------
-- A Ctxt records where we've been as we descend
-- down into a tree as we search for a value

data Dir = LeftD | RightD

data Ctxt:: Color ~> Nat ~> *0 where
  Nil:: Ctxt Black n
  RCons:: Int -> Dir ->
          SubTree Black n ->
          Ctxt Red n ->
          Ctxt Black n
  BCons:: Int -> Dir ->
          SubTree c1 n ->
          Ctxt Black (S n) ->
          Ctxt c n

------------------------------------------------------------------------
-- Turn a Red tree into a black tree. Always
-- possible, since Black nodes do not restrict
-- the color of their sub-trees.

blacken :: SubTree Red n -> SubTree Black (S n)
blacken (RNode l e r) = (BNode l e r)


------------------------------------------------------------------------
-- A singleton type representing Color at
-- the value level.

data CRep :: Color ~> *0 where
  Red   :: CRep Red
  Black :: CRep Black

color :: SubTree c n -> CRep c
color Leaf = Black
color (RNode _ _ _) = Red
color (BNode _ _ _) = Black

------------------------------------------------------------------------
```

```
-- fill a context with a subtree to regain the original
-- RBTree, works if the colors and black depth match up

fill :: Ctxt c n -> SubTree c n -> RBTree
fill Nil t = Root t
fill (RCons e LeftD  uncle c) tree = fill c (RNode uncle e tree)
fill (RCons e RightD uncle c) tree = fill c (RNode tree  e uncle)
fill (BCons e LeftD  uncle c) tree = fill c (BNode uncle e tree)
fill (BCons e RightD uncle c) tree = fill c (BNode tree  e uncle)

insert :: Int -> RBTree -> RBTree
insert e (Root t) = insert_ e t Nil

------------------------------------------------------------------------
-- as we walk down the tree, keep track of everywhere
-- we've been in the Ctxt input.

insert_ :: Int -> SubTree c n -> Ctxt c n -> RBTree
insert_ e (RNode l e' r) ctxt
        | e < e'        = insert_ e l (RCons e' RightD r ctxt)
        | True          = insert_ e r (RCons e' LeftD  l ctxt)
insert_ e (BNode l e' r) ctxt
        | e < e'        = insert_ e l (BCons e' RightD r ctxt)
        | True          = insert_ e r (BCons e' LeftD  l ctxt)
-- once we get to the bottom we "insert" the node as a Red node.
-- since this may break invariant, we may need do some patch work
insert_ e Leaf ctxt = repair (RNode Leaf e Leaf) ctxt

------------------------------------------------------------------------
-- Repair a tree if its out of balance. The Ctxt holds
-- crucial information about colors of parent and
-- grand-parent nodes.

repair :: SubTree Red n -> Ctxt c n -> RBTree
repair t (Nil)                 = Root (blacken t)
repair t (BCons e LeftD  sib c) = fill c (BNode sib e t)
repair t (BCons e RightD sib c) = fill c (BNode t e sib)
-- these are the tricky cases
repair t (RCons e dir sib (BCons e' dir' uncle ctxt)) =
  case color uncle of
    Red   -> repair (recolor dir e sib dir' e' (blacken uncle) t) ctxt
    Black -> fill ctxt (rotate dir e sib dir' e' uncle t)
repair t (RCons e dir sib (RCons e' dir' uncle ctxt)) = unreachable

recolor :: Dir -> Int -> SubTree Black n ->
           Dir -> Int -> SubTree Black (S n) ->
           SubTree Red n -> SubTree Red (S n)
recolor LeftD  pE sib RightD gE uncle t = RNode (BNode sib pE t) gE uncle
recolor RightD pE sib RightD gE uncle t = RNode (BNode t pE sib) gE uncle
```

```
recolor LeftD  pE sib LeftD  gE uncle t = RNode uncle gE (BNode sib pE t)
recolor RightD pE sib LeftD  gE uncle t = RNode uncle gE (BNode t pE sib)

rotate :: Dir -> Int -> SubTree Black n ->
          Dir -> Int -> SubTree Black n ->
          SubTree Red n -> SubTree Black (S n)
rotate RightD pE sib RightD gE uncle (RNode x e y) =
   BNode (RNode x e y) pE (RNode sib gE uncle)
rotate LeftD  pE sib RightD gE uncle (RNode x e y) =
   BNode (RNode sib pE x) e (RNode y gE uncle)
rotate LeftD  pE sib LeftD  gE uncle (RNode x e y) =
   BNode (RNode uncle gE sib) pE (RNode x e y)
rotate RightD pE sib LeftD  gE uncle (RNode x e y) =
   BNode (RNode uncle gE x) e (RNode y pE sib)
```

B 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, Ωmega 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.