Ωmega Users' Guide

Omega Interpreter: version 1.4.4
Sun Mar 28 16:05:41 Pacific Daylight Time 2010

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

Contents

This manual corresponds to the Ωmega implementation with the following version information.

Omega Interpreter: version 1.4.4
Sun Mar 28 16:05:41 Pacific Daylight Time 2010

Ωmega Users' Guide

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

March 28, 2010

1 Introduction

The Ωmega interpreter is styled after the Hugs Haskell Interpreter. In fact, users of Hugs will find many similarities, and this is intentional. Like Hugs, the Ωmega interpreter is a read-typecheck-eval-print loop. It can load whole files, and the top-level loop allows the user to experiment by typing expressions to be evaluated, and by querying the program environment.

The Ωmega syntax is based upon the syntax of Haskell. If you're unsure of what syntax to use, a best first approximation is to use Haskell syntax. It works most of the time. While clearly descended from Haskell, Ωmega has several important syntactic and semantic differences. The purpose of this manual is to identify the differences and give examples of the differences so users can quickly learn them. These differences include:

- Ωmega supports the introduction of Generalized Algebraic Datatypes. It does this by generalizing the `data` declaration to include explicit types for every constructor (Section 7).
- Ωmega adds the ability to introduce new types, and new kinds to classify these types (Section 11).
- Ωmega allows users to write functions at the type level, and to use these functions in the type of functions at the value level (Section 18).
- Ωmega is strict while Haskell is lazy. But Ωmega has an experimental explicit laziness annotation (Section 17).
- Ωmega does not support Haskell's class system. Because of this Ωmega supports the `do` syntax using another mechanism (Section 6).
- Ωmega has only a fixed set of infix operators, which cannot be extended (Section 3).
- Ωmega only supports fully applied type synonyms (Section 5).
- Ωmega has a very primitive module system (Section 4).
- Ωmega suports a notion of anonymous existential types (Section 10).

- Ωmega has an experimental implementation of Pitts and Gabbay's freshness mechanism (Section 25).

```
Type :: Kind

Monad :: (*0 ~> *0) ~> *0
Record :: Row Tag *0 ~> *0
RCons ::
   e ~> f ~> Row e f ~> Row e f
RNil ::
   Row e f
Row :: a ~> c ~> *1
HiddenLabel :: *0
Equal :: a ~> a ~> *0
Nat' :: Nat ~> *0
S :: Nat ~> Nat
Z :: Nat
Nat :: *1
and :: Prop ~> Prop ~> Prop
Success :: Prop
Prop :: *1
Maybe :: *0 ~> *0
Ordering :: *0
Bool :: *0
```

```
DiffLabel :: Tag ~> Tag ~> *0
Label :: Tag ~> *0
Tag :: *1
Parser :: *0 ~> *0
Ptr :: *0 ~> *0
IO :: *0 ~> *0
Symbol :: *0
Code :: *0 ~> *0
ChrSeq :: *0
Float :: *0
Char :: *0
Int :: *0
[] :: *0 ~> *0
(,) :: *0 ~> *0 ~> *0
() :: *0
(->) :: *0 ~> *0 ~> *0
Atom :: a ~> *0
(+) :: *0 ~> *0 ~> *0
(!=) :: a ~> a ~> *0
DiffLabel :: Tag ~> Tag ~> *0
String :: *0
(==) :: a ~> a ~> Prop
```

Figure 1: Predefined types and their kinds.

```
Value :: Type

(+) :: Int -> Int -> Int
(*) :: Int -> Int -> Int
(-) :: Int -> Int -> Int
div :: Int -> Int -> Int
rem :: Int -> Int -> Int
mod :: Int -> Int -> Int
negate :: Int -> Int
(<) :: Int -> Int -> Bool
(<=) :: Int -> Int -> Bool
(>) :: Int -> Int -> Bool
(>=) :: Int -> Int -> Bool
(==) :: Int -> Int -> Bool
(/=) :: Int -> Int -> Bool
(#+) :: Float -> Float -> Float
(#*) :: Float -> Float -> Float
(#-) :: Float -> Float -> Float
(#/) :: Float -> Float -> Float
negateFloat :: Float -> Float
(#<) :: Float -> Float -> Bool
(#<=) :: Float -> Float -> Bool
(#>) :: Float -> Float -> Bool
(#>=) :: Float -> Float -> Bool
(#==) :: Float -> Float -> Bool
(#/=) :: Float -> Float -> Bool
intToFloat :: Int -> Float
round :: Float -> Int
truncate :: Float -> Int
eqStr :: [Char] -> [Char] -> Bool
chr :: Int -> Char
ord :: Char -> Int
putStr :: [Char] -> IO ()
getStr :: IO [Char]
```

```
(&&) :: Bool -> Bool -> Bool
(||) :: Bool -> Bool -> Bool
not :: Bool -> Bool
True :: Bool
False :: Bool
(:) :: a -> [a] -> [a]
null :: [a] -> Bool
[] :: [a]
(++) :: [a] -> [a] -> [a]
(,) :: a -> b -> (a,b)
Nothing :: Maybe a
Just :: a -> Maybe a
show :: a -> [Char]
mimic :: (a -> b) -> a -> b
strict :: a -> a
trace :: [Char] -> a -> a
compare :: a -> a -> Ordering
error :: [Char] -> a
fresh :: Char -> Symbol
swap :: Symbol -> Symbol -> a -> a
symbolEq :: Symbol -> Symbol -> Bool
sameLabel :: Label a -> Label b -> ((Equal a b)+(DiffLabel a
freshLabel :: IO HiddenLabel
newLabel :: [Char] -> HiddenLabel
LabelNotEq :: Ordering -> DiffLabel a b
freshen :: a -> (a,[(Symbol,Symbol)])
run :: Code a -> a
lift :: a -> Code a
returnIO :: a -> IO a
bindIO :: IO a -> (a -> IO b) -> IO b
failIO :: [Char] -> IO a
handleIO :: IO a -> ([Char] -> IO a) -> IO a
($) :: (a -> b) -> a -> b
(.) ::
   (a -> b) -> (c -> a) -> c -> b
```

Figure 2: Predefined functions and values.

2 Primitive and Predefined Types and Values

Ωmega supports a number of built in types. In Figure 1 they are listed along with their kinds. Ωmega also implements a number of primitive functions and values over the predefined types. In Figure 2 they are listed along with their types. These figures are generated when a distribution (including this manual) is built, and should be up-to date. Compare the version and build date of your Ωmega interpreter with the information on the license page of this manual to see if this manual is up-to-date. In a future version of Ωmega we hope to have a more comprehensive set of primitives.

The following types are predefined. All other types listed in Figure 1 are primitive, abstract, builtin types. The predefined types behave as if they were defined as follows:

```
data Int = primitive
data Char = primitive
data Float = primitive
data ChrSeq = primitive
data Code (x::*0) = primitive
data Symbol = primitive
data IO (x::*0) = primitive
data Ptr (x::*0) = primitive
data Parser (x::*0) = primitive
kind Tag = primitive
data Label (t :: Tag) = primitive
prop DiffLabel (t :: Tag) (t :: Tag) = primitive
data Bool:: *0 where
  True:: Bool
  False:: Bool
data Ordering:: *0 where
  EQ:: Ordering
  LT:: Ordering
  GT:: Ordering
data Maybe:: *0 ~> *0 where
  Just :: a -> Maybe a
  Nothing :: Maybe a
data Prop :: *1 where
  Success :: Prop
and:: Prop ~> Prop ~> Prop
{and Success x } = x
data Nat :: *1 where
  Z :: Nat
  S :: Nat ~> Nat
 deriving Nat(t)
prop Nat' :: Nat ~> *0 where
  Z:: Nat' Z
  S:: forall (a:: Nat) . Nat' a -> Nat' (S a)
 deriving Nat(v)
data Equal:: a ~> a ~> *0 where
  Eq:: Equal x x
data HiddenLabel :: *0 where
  HideLabel:: Label t -> HiddenLabel
```

```
Operator :: Type

(*) :: Int -> Int -> Int
(#*) :: Float -> Float -> Float
(#/) :: Float -> Float -> Float
(+) :: Int -> Int -> Int
(-) :: Int -> Int -> Int
(#+) :: Float -> Float -> Float
(#-) :: Float -> Float -> Float
(:) :: a -> [a] -> [a]
(++) :: [a] -> [a] -> [a]
(==) :: Int -> Int -> Bool
(/=) :: Int -> Int -> Bool
(<) :: Int -> Int -> Bool
(<=) :: Int -> Int -> Bool
(>) :: Int -> Int -> Bool
(>=) :: Int -> Int -> Bool
(#==) :: Float -> Float -> Bool
(#/=) :: Float -> Float -> Bool
(#<) :: Float -> Float -> Bool
(#<=) :: Float -> Float -> Bool
(#>) :: Float -> Float -> Bool
(#>=) :: Float -> Float -> Bool
(&&) :: Bool -> Bool -> Bool
(||) :: Bool -> Bool -> Bool
($) :: (a -> b) -> a -> b
(.) ::
   (a -> b) -> (c -> a) -> c -> b
```

Figure 3: The fixed set of infix operators and their types.

```
data Row :: a ~> b ~> *1 where
  RNil :: Row x y
  RCons :: x ~> y ~> Row x y ~> Row x y
 deriving Record(r)
data Record :: Row Tag *0 ~> *0 where
   RecNil :: Record RNil
   RecCons :: Label a -> b -> Record r -> Record (RCons a b r)
 deriving Record()
data Monad :: (*0 ~> *0) ~> *0 where
  Monad :: forall (m:: *0 ~> *0) .
                  ((forall a . a -> m a)) ->
                  ((forall a b . (m a) -> (a -> m b) -> m b)) ->
                  ((forall a . String -> m a)) ->
                  Monad m

data [] a = [] | (:) a [a]
```

3 Infix Operators

In Ωmega there are a fixed number of infix operators. The current implementation precludes dynamically adding new infix operators. The infix operators can be found in Figure 3.

4 Simple Module System

The module system in Ωmega is very primitive. It works by specifying a string which is the name of an Ωmega file. It is followed by an optional parenthesized list of import items. The file is loaded, and if the import items are present in the file, those import items are imported into the current context. An import item is either a name or a syntax item. A syntax item names one of the extensible syntactical extensions (see Section 22). A syntax item has the form:

```
"syntax" ("Num" | "List" | "Record" | "Pair") "(" id ")"
```

Some examples are illustrated below.

```
import "LangPrelude.prg"
  (head,tail,lookup,member,fst,snd,Monad,maybeM)
```

imports only the names listed from the `LangPrelude.prg` file.

If the parenthesized import list is not present, all the names and syntax extensions defined in the file are imported. An Ωmega program can contain multiple imports.

```
import "fileAll"

import "generic.prg" (name, syntax List(i), syntax Num(n))
```

Like Haskell, Ωmega has two name spaces. One for values and another for types and kinds. One drawback of the module system is that if a name is listed in the import list, then that name is imported into both name spaces if it exists. There is currently no way to be selective.

5 Type Synonyms

Starting with release 1.1 Ωmega supports both parameterized and non-parameterized type synomyms. For example:

```
type String = [Char]
type Env t = [(String,t)]
```

Every use of a type synonym must be fully applied to all its type arguments. At type checking time a type synonym is fully expanded.

6 Monads and the `do` Syntax

Since Ωmega does not have a class system, it uses an alternate mechanism to support the `do` syntax for monads. Typing a `do` expression involves computing the type of the variables `return`, `fail`, and `bind` in the scope where the `do` expression occurs. The typing rule is:

$$\frac{\begin{array}{l}\Gamma(x,d) \vdash \mathtt{e_2} :: m\ \ c \\ \Gamma \vdash \mathtt{e_1} :: m\ \ d \\ \Gamma \vdash \mathtt{fail} :: \forall a\,.\,String \rightarrow m\ \ a \\ \Gamma \vdash \mathtt{bind} :: \forall a\,b\,.\,m\ \ a \rightarrow (a \rightarrow m\ \ b) \rightarrow m\ \ b \\ \Gamma \vdash \mathtt{return} :: \forall a\,.\,a \rightarrow m\ \ a\end{array}}{\Gamma \vdash\ \mathtt{do}\ \{x \leftarrow e_1;\ e_2\} :: m\ \ c}$$

A Monad in Ωmega is just an ordinary algebraic datatype with constructor `Monad` and three polymorphic functions as components, `return`, `bind`, and `fail`. Its definition can be found in Section 2 as one of the predefined types. Some sample monad declarations are:

```
maybeM =  (Monad Just bind fail)
  where return x = Just x
        fail s = Nothing
        bind Nothing g = Nothing
        bind (Just x) g = g x

listM =  (Monad unit bind fail)
  where unit x = [x]
        fail s = []
        bind [] f = []
        bind (x:xs) f = f x ++ bind xs f

ioM = Monad returnIO bindIO failIO
```

```
data St st a = St (st -> (a,st))

stateM =  (Monad unit bind fail)
  where unit a = St(\ st -> (a,st))
        bind (St f) g = St h
          where h st = case f st of
                         (a,st') ->
                            case g a of
                             (St j) -> j st'
        fail s = error ("Fail in state monad: "++s)

runstate n (St f) = f n
getstate = St f  where f s = (s,s)
setstate n = St f   where f s = ((),n)
```

Given any expression (m::`Monad m`), the monad declaration (`monad` m) is syntactic sugar for the pattern based declaration `(Monad return bind fail) =` m. Which introduces bindings for the monad functions (`return`, `bind`, and `fail`) into the local scope. To use the `do` syntax import the names for `return`, `bind`, and `fail` into the current context either by defining them or using the `monad` declaration.

```
d1 = do { x <- Just 4; return(x+1) }
  where monad maybeM

d2 = runstate 0 (do { setstate 4; x <- getstate; return(x+1) })
  where (Monad return bind fail) = stateM

d3 = runstate 5 (do { a <- getstate; setstate 3; x <- getstate; return(x+a) })
  where monad stateM
```

To import a monad into the scope of a multi-clause function definition without polluting the global scope, define the function using a `where` clause. Place a helper function and the `monad` declaration in the same `where` clause, and equate the function (you really want to define) to the helper function.

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

test = help where
  help :: Rep a -> Rep b -> Maybe(Equal a b)
  help Int Int = Just Eq
  help Char Char = Just Eq
  help (Prod a b) (Prod m n) =
    do { Eq <- help a m
```

```
      ; Eq <- help b n
      ; return Eq }
  help (Arr a b) (Arr m n) =
    do { Eq <- help a m
      ; Eq <- help b n
      ; return Eq }
  help (List a) (List b) =
    do { Eq <- help a b; return Eq }
  help _ _ = Nothing
  monad maybeM
```

7 Generalized Algebraic Datatypes

We assume the reader has a certain familiarity with Algebraic Datatypes (ADTs). In particular that values of ADTs are constructed by *constructors* and that they are taken apart by the use of *pattern matching*. Consider

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

This declaration defines the polymorphic `Tree` type constructor. Example tree types include `(Tree Int)` and `(Tree Bool)`. Note how the constructor functions (`Fork`, `Node`) and constants (`Tip`) are given polymorphic types.

```
Fork ::  forall a .  Tree a -> Tree a -> Tree a
Node ::  forall a .  a -> Tree a
Tip ::  forall a .  Tree a
```

When we define a parameterized algebraic datatype, the formation rules enforce the following restriction. The range of every constructor function, and the type of every constructor constant must be a polymorphic instance of the new type constructor being defined. Notice how the constructors for `Tree` all have range (`Tree` a) with a polymorphic type variable a. Generalized Algebraic Datatypes (GADTs) remove this restriction. Since the range of the constructors of an ADT are only implicitly given (as the type to the left of the equal sign in the ADT definition), a new syntax is necessary to remove the range restriction. In Ωmega a new form of declaring new datatypes is supported. We call this form the GADT (or explicit) form.

1. An explicit form of a `data` definition is supported in which the type being defined is given an explicit kind, and every constructor is given an explicit type.

We illustrate the new form below:

```
-- A GADT declaration using explicit classification. Note the range
-- of the constructor 'Pair' is not 'Term' applied to a type variable.
data Term :: *0 ~> *0 where
```

```
  Const :: a -> Term a
  Pair :: Term a -> Term b -> Term (a,b)
  App :: Term(a -> b) -> Term a -> Term b

-- The ADT, Tree, using the explicit GADT form (allowed but not necessary).
data Tree:: *0 ~> *0 where
  Fork:: Tree a -> Tree a -> Tree a
  Node:: a -> Tree a
  Tip:: Tree a
```

Like an ADT style declaration, the GADT style declaration introduces a new type constructor (`Term`) which is classified by a kind (`*0 ~> *0`). Here the kind is made explicit. This means that `Term` takes types to types. In the GADT form no restriction is placed on the types of the constructor functions except that the range of each constructor must be a fully applied instance of the type being defined, and that the type of the constructor as a whole must be classified by $*n$, for some n. See the rest of this manual, and many of the papers about Ωmega listed in Section 27, for many more examples.

Note that all Algebraic Data Types, like `Tree`, can be expressed using the explicit GADT form. We retain the ADT form for backward compatibility.

7.1 The Off-side Rule in GADT Declaration

Note also that the ADT form separates the declaration of constructor functions by using the vertical bar (`|`), but the GADT form uses the offside rule. The declarations of all the constructor functions should begin in the same column.

7.2 Equality-constrained Types are Deprecated

Some early versions of Ωmega supported an additional form of declaration using equality constraints in `where` clauses. For example:

```
-- Old style "equality-constraint" style.
data Term a
  = Const a
  | exists x y . Pair (Term x)(Term y) where a =(x,y)
  | exists b . App (Term(b -> a)) (Term b)
```

This form has been deprecated starting with Ωmega version 1.2.3. It is no longer allowed. In order to help users convert old Ωmega programs, the system prints an error message and suggests a GADT style definition. For example, given the old-style declaration above the system suggests:

```
The implicit form of data decls (using where clauses) has been deprecated.
Please use the explicit form instead. Rather than writing:

data Term a = Const a
            | forall x y . Pair (Term x) (Term y)
                             where a = (x, y)
```

```
            | forall b . App (Term (b -> a)) (Term b)

One should write something like:
(One may need to adjust the kind of 'Term')

data Term:: (*? ~> *0) where
   Const:: forall a . a -> Term a
   Pair:: forall x y a . Term x -> Term y -> Term (x, y)
   App:: forall b a . Term (b -> a) -> Term b -> Term a
```

Sometimes the classification of the type constructor is not correct (See the `"*?"` above), but other than this, it can be cut and pasted into the file to replace the old-style declaration.

8 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 18), then since many functions may compute the same result (even with 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 checker's 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 19).

9 Polymorphism and Data Declarations

Ωmega supports both existentially and universally quantified components in `data` declarations. An example of existential quantification is:

```
data Closure:: *0 ~> *0 ~> *0 where
   Close:: env -> (env -> a -> b) -> Closure a b
```

In the type of a constructor function, *all* type variables not appearing in the range of the constructor function (here, the variable `env`) are implicitly existentially quantified. The type of `Close` is polymorphic since it can be applied to any kind of environment, but special typing rules are employed when pattern matching against a pattern like `(Close e f)` which ensure that the *type* of the pattern variable `e` does not escape its scope.

Polymorphic (or universally quantified) components can be declared by placing a universal quantifer in the arg position of a constructor. An example of this are the types of the `return`, `bind`, and `fail` components of the `Monad` declaration (defined in Section 2) and used in Section 6. Another example would be the type that can only store the identity function,

```
data Id = Id (forall a . a -> a)
```

It is also possible to give prototypes with rank-N types, and the type checker will check that the defined function is called properly. The Ωmega implementation is based upon Simon Peyton Jones work on implementing type-checking for rank-N polymorphism[JS03]

10 Anonymous Existential Types

Existential types can be encoded in Ωmega inside of `data` declarations by using the implict existential quantification of type variables not occurring in the range-type of constructor functions as shown above in the declaration for `Close`.

Sometimes it is convenient to have existential quantification without defining a `data` type to encode it. This is possible in Ωmega by using the anonymous existential constructor `Ex`. Because `Ex` encodes anonymous existential types, the user is required to use function prototypes that propagate the intended existential type into the use of `Ex`. For example:

```
existsA :: exists t . (t,t->String)
existsA = Ex (5,\ x -> show(x+1))

testf :: (exists t . (t,t-> String)) -> String
testf (Ex (a,f)) = f a
```

The keyword `Ex` is the "pack" operator of Cardelli and Wegner [CW85]. 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.

11 New Types with Different Kinds

Kinds are similar to types in that, while types classify values, kinds classify types. We indicate this by the overloaded *classifies* relation (`::`). For example: `5::Int`, and `Int::*0` . We say `5` is classified by `Int`, and `Int` is classified by `*0` (star-zero). The kind `*0` classifies all types that classify values (things we actually can compute). A `kind` declaration introduces new types and their associated kinds (just as a `data` declaration introduces new values (the constructors) and their associated types). Types introduced by a `kind` declaration have kinds other than `*0`. For example, the `Nat` declaration introduces two new type constructors `Z` and `S` which encode the natural numbers at the type level:

```
kind Nat = Z | S Nat
```

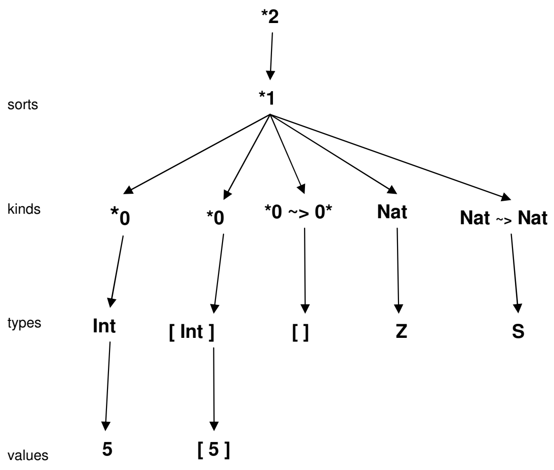

Figure 4: The classification hierarchy. An arrow from a to `b` means $b{::}a$. Note how only values are classified by types that are classified by `*0`, and how type constructors (like `[]` and `S`) have higher order kinds.

The type `Z` has kind `Nat`, and `S` has kind `Nat ~> Nat`. The type `S` is a type constructor, so it has a higher-order kind. We indicate this using the classifies relation as follows:

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

The classification `Nat::*1` indicates that `Nat` is at the same "level" as `*0` — they are both classified by `*1`. Just as every ADT can be expressed as a GADT, every `kind` declaration can also be expressed as an GADT. For example we could write the declaration of `Nat` as a GADT as follows:

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

The system infers that this is a *kind* declarartion by observing that `Nat` is kinded by `*1` (GADTs kinded by `*0` would be types, and those kinded by `*2` would be sorts). There is an infinite hierarchy of classifications, and users can populate each level of the classification using GADT style declarations. Each level is associated with one of the kinds `*`n. For example, at the kind level `*0` and `Nat` are classified by `*1`, `*1` is classified by `*2`, etc. We call this hierarchy the *strata*. In fact this infinite hierarchy is why we chose the name Ωmega. The first few strata are: values and expressions that are classified by types, types that are classified by kinds, and kinds that are classified by sorts, etc. We illustrate the relationship between the values, types, kinds, and sorts in Figure 4.

In general, a GADT declaration mediates between two levels of the hierarchy. Introducing the type constructor at level $n+1$, and constructor functions at level n. The ADT `data` declaration is a special case of the GADT introducing a type constructor at the type level, and constructor functions at the value level (as it does in Haskell). The `kind` declaration is another special case introducing a type constructor at the kind 2, and constructor functions at the type level.

Kinds (Sorts, etc.) are useful because we can use them to index new types. For example the type constructor `List` is indexed by `Nat`, and the `Nat` valued index is a static indication of the length of the list.

```
data List:: Nat ~> *0 ~> *0 where
  Nil:: List Z a
  Cons:: a -> List m a -> List (S m) a
```

We can also use kinds to define other interesting types. For example we can define our own kind of "tuples" using the `Prod` kind.

```
data Prod:: *1 ~> *1 where
  PNil:: Prod a
  PCons:: k ~> Prod k ~> Prod k

data Tuple:: Prod *0 ~> *0 where
   Tnil:: Tuple PNil
   Tcons:: t -> (Tuple r) -> Tuple (PCons t r)
```

The type `(PCons Int PNil)` is classified by `(Prod *0)`, but the type `(PCons Z (PCons (S Z) PNil))` is classified by `(Prod Nat)`. The value level term `(Tcons 5 (Tcons True Tnil)))` is classified by the type `(Tuple (Rcons Int (Rcons Bool Rnil)))`.

11.1 Polymorphic kinds

In Ωmega it is possible to give a type contructor introduced using the GADT form a polymorphic classification. This allows users to specify kinds for type constructors that cannot be inferred. In the example below we specify a polymorphic kind for the type `TRep`

```
data TRep:: forall (k :: *1) . (k ~> *0) where
  Int:: TRep Int
  Char:: TRep Char
  Unit:: TRep ()
  Prod:: TRep (,)
  Sum:: TRep (+)
  Arr:: TRep (->)
  Ap:: TRep f -> TRep x -> TRep (f x)
```

This allows a single GADT to represent both type constructors like `(->)` and `(,)`, as well as types like `Int` and `Char`. The term `(Ap(Ap Prod Int) Unit)` is classified by the type `(TRep (Int,()))`.

12 Separate Value and Type Name Spaces

In Ωmega, as in Haskell, the name-space for the value level is separate from the name-space for the type level. In addition, in Ωmega, the type name-space includes all levels above the value level. This includes types, kinds, sorts etc. This allows a sort of punning, where objects at the value level can have the same name as different objects at the type level and above. This punning becomes particularly useful when the objects at the two levels are different, but related. The classic pun is the singleton type `Nat'`

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

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

The value constructors `(Z:: Nat' Z)` and `(S:: Nat' n -> Nat' (S n))` are ordinary values whose types mention the type constructors they pun. In `Nat'`, the singleton relationship between a `Nat'` value and its type is emphasized strongly, as witnessed by the example `three'`. Here the shape of the value, and the type index appear isomorphic. The table below illustrates the sdeparate name spaces and the levels within the type hierarchy where each constructor resides.

⟵ *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 Ωmega, we further exploit this pun, by providing syntactic sugar for writing natural numbers at both the value and the type level. We write `0t` (for `Z`), `1t` (for `S Z`), `2t` (for `S(S Z)`), etc. for `Nat` terms at the type level. We write `0v` (for `Z`), `1v` (for `S Z`), `2v`, (for `S(S Z)`) etc. for `Nat'` terms at the value level. This syntactic sugar is an example of syntactic extension. See Section 22 for further examples. For backward compatibility reasons one may also write `#0`, `#1`, `#2`, etc.

13 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 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 (see Section 15), 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
```

14 Staging

Ωmega includes two staging annotations and brackets: (`[| _ |]`), escape (`$( _ )`), and two functions: `lift::(forall a . a -> Code a)` and `run::(forall a . (Code a) -> a)` for building and manipulating code. Ωmega uses the Template Haskell[SPJ02] conventions for creating code. 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 |]
```

The purpose of the staging mechanism is to have finer control over evaluation order. Ωmega supports many of the features of MetaML[She99, TS00]. The staging in Ωmega is experimental and may change in future releases.

An example staged term is:

```
c18 = let h 0 z = z
          h n z = [| let y = n in $(h (n-1) [| $z + y |]) |]
      in h 3 [| 99 |]
```

This term evaluates to the piece of code below:

```
[| let y297 = 3
       y299 = 2
       y301 = 1
   in 99 + y297 + y299 + y301
|] : Code Int
```

15 Command Level Prompts

Once inside the Ωmega interpreter the user interacts with Ωmega using command-level prompts. Most commands begin with the letter `':'` In Figure 5, the command level prompts are defined and their usage is explained. Commands are modelled after the Hugs command level prompts. There are far fewer commands in Ωmega than in Hugs. The `:set` and `:clear` command are discussed in much more detail in Section 24 where we discuss user controlled tracing inside the type-checker.

```
let v = e    bind 'v' to 'e' in interactive loop
v <- e       evaluate IO expression 'e' and bind to 'v'
exp          read-typecheck-eval-print 'exp'
:quit        quit
:type        display type of expression
:env n       display info for first 'n' items
:env str     display info for items with 'str' in their names
:reload file reload file into system
:version     display version info
:kind type   display kind of type expression
:load file   load file into system. Start with fresh, empty environment
:also file   load file into system, extending current environment
:set         list all the Mode variables and their settings
:set mode    set the Mode variable X (where 'mode' is a prefix of X) to True
:clear mode  clear the Mode variable X (where 'mode' is a prefix of X) to False
:init        reset system to initial configuration (flush definitions)
:rules name  display rules for 'name' (if name is omitted, displays all rules)
:pre         display declarations for all predefined types
:narrow type narrow type expression
:norm type   normalize type expression (use function definitions and rewrites)
:bounds X n  set the resource bound X to n
:sources     list the source files currently loaded
:?           display list of legal commands (this message)
```

Figure 5: Legal inputs to the command line interpreter.

15.1 The command line editor

Starting with Ωmega version 1.2, the read eval print loop comes equipped with a command line editor. The editor is implemented by the *GNU Readline Library.* This is the same library used to implement the command line editor in many Linux distributions, so it should be familiar to many users. For a short introduction one can consult any of the online introductions to the library. One that I used was `http://www.ugcs.caltech.edu/info/readline/rlman_1.html`.

Amongst many other features, the command line editor allows users to cycle through previous commands by using the up (↑) and down (↓) arrow keys, to move the cursor forward and backward using the right (→) and left (←) arrow keys, and to edit text using backspace and delete. The command editor also implements tab completion by looking up prefixes in the Ωmega symbol table. I'd like to thank Nils Anders Danielsson for the initial implementation.

The command line editor is also installed in the Read-typecheck-print loop described in the next section.

16 The Type Checking Read-Typecheck-Print Loop

Ωmega allows the user to investigate the types of expressions while type checking them. By placing the language construct "`check _` " before a term the type checker enters a read-typecheck-print loop in the scope where the `check` occurs. The check construct has the lowest parsing precedence, so its scope (like the lambda expression), extends to as far to the right as possible. One can delimit its scope by using parenthesis. For example consider:

```
data LE:: Nat ~> Nat ~> *0 where
   LeBase:: LE n n
   LeStep:: LE n m -> LE n (S m)

compare :: Nat' a -> Nat' b -> ((LE a b)+(LE b a))
compare Z Z = L LeBase
compare Z (S x) =
  case compare Z x of L w -> L(LeStep w)
compare (S x) Z =
  case compare Z x of L w -> R(LeStep w)
compare (S x) (S y) = check mapP g g (compare x y)
  where mapP f g (L x) = L(f x)
        mapP f g (R x) = R(g x)
        g :: LE x y -> LE (S x) (S y)
        g LeBase = LeBase
        g (LeStep x) = LeStep(g x)
```

In the 4th clause of `compare`, the `check` construct extends to the end of the line, encompassing the whole `mapP` application. The purpose of the `check` is to allow the user to investigate the type of some expressions in a rather large complex function definition. The `check` in the scope of the fourth clause of `compare` directs the type checker to pause and enter an interactive loop. The following is a script of one such interaction. It places the user in the scope of the fourth clause to `compare` above, where the `check` keyword appears. Note that, in the transcript below, the user enters the code after the `check>` prompt.

```
*** Checking: mapP g g (compare x y)
*** expected type: ((LE (1+_c)t (1+_d)t)+(LE (1+_d)t (1+_c)t))
***    refinement: {_a=(1+_c)t, _b=(1+_d)t}
***   assumptions:
***      theorems:
check> x
x :: Nat' _c

check> y
y :: Nat' _d

check> mapP
mapP :: (e -> f) -> (g -> h) -> (e+g) -> (f+h)

check> g
g :: LE i j -> LE (1+i)t (1+j)t

check> :h
```

```
Hint = ((LE (1+_c)t (1+_d)t)+(LE (1+_d)t (1+_c)t))

check> compare x y
compare x y :: ((LE _c _d)+(LE _d _c))

check> mapP g g
mapP g g :: ((LE k l)+(LE m n)) -> ((LE (1+k)t (1+l)t)+(LE (1+m)t (1+n)t))

check> mapP g g (compare x y)
mapP g g (compare x y) :: ((LE (1+_c)t (1+_d)t)+(LE (1+_d)t (1+_c)t))
```

Inside the read-typecheck-print loop the user may enter expressions whose types are then printed, or they may use one of the following commands.

`:q`	Quit checking and let the type checker continue.
`:e`	Show the assumptions list in the current environment.
`:k t`	Print the kind of type `t`.
`:h`	Show the *hint* the type the type checker expects the checked expression to have.
`:t f`	Print the type scheme for the variable `f`.
exp	Type check and print the type of the expression *exp*.
`:set m`	Set the mode m, the same modes are available here as in the interactive loop.
`:try exp`	Compare the type of `exp` with the expected type, and display generated constraints.

17 Explicit Laziness

Ωmega is a strict, but pure, language. All side-effects are captured in the `IO` monad. We have included an experimental feature in Ωmega. That feature is the introduction of explicit laziness. It is best to read the paper *A Pure Language with Default Strict Evaluation Order and Explicit Laziness*[She03] for a complete description. We list the interface to this feature here for completeness.

```
lazy :: e -> e        -- lazy is a language construct not a function
strict :: e -> e
mimic :: (f -> g) -> f -> g
```

We can use this interface to build infinite streams

```
(twos,junk) = (2:(lazy twos),lazy(head twos))
(ms,ns) = (1:(lazy ns),2:(lazy ms))
```

Expressions labelled with `lazy` are not evaluated until they are pulled on. Printing in the read-eval-print loop, does not print `lazy` thunks, but prints them as `...`.

```
prompt> twos
[2 ...] : [Int]
prompt> take 5 twos
[2,2,2,2,2] : [Int]
```

A classic example is the infinite list of fibonacci numbers.

```
-- fibonacci
zipWith f (x:xs) (y:ys) =
  f x y : mimic (mimic (zipWith f) xs) ys

fibs = 0 : 1 : (lazy (zipWith (+) fibs (tail fibs)))
```

We can observe a finite prefix of fibs by defining a take function

```
take 0 xs = []
take n [] = []
take n (x:xs) = x :(take (n-1) xs)

prompt> take 10 fibs
[0,1,1,2,3,5,8,13,21,34] : [Int]
```

18 Type Functions

Ωmega allows programmers to write arbitrary type functions as first order equations, which are interpreted as left to right rewrite rules. In a future release we expect to limit such definitions to confluent and terminating sets of rewrite rules. A type function "computes" over types of a particular kind, and can be used in a prototype declaration of a function that mentions a type of that kind. We can illustrate this by the Ωmega program

```
data List:: Nat ~> *0 ~> *0 where
  Nil:: List Z a
  Cons:: a -> List m a -> List (S m) a

plus :: Nat ~> Nat ~> Nat
{plus Z y} = y
{plus (S x) y} = S{plus x y}

app:: List n a -> List m a -> List {plus n m} a
app Nil ys = ys
app (Cons x xs) ys = Cons x (app xs ys)
```

The code introduces the type of sequences, `Seq`, with static length (a `Nat`), the new *type-function*, `plus` (over `Nat`), and the definition of `app` the append function over sequences. Note how the length of the result is a function of the length of the input sequences. Since the length of a sequence is a `Nat` which is a type, a function over types is required to give a type to `app`.

Type-functions are functions at the type level. We define them by writing a set of equations. We distinguish type-function application from type-constructor (i.e. `Tree` or `Term`) by enclosing them in squiggly brackets. Type functions must be preceded by prototype declaration stating their kind. They consist of a set of exhaustive (over their kind) and confluent rewriting rules. This is not currently enforced, but failure to do this may cause the Ωmega type checker to diverge.

Type checking `app`, generates and propagates equalities, and requires solving the equations like `(S{plus t m} = {plus n m})` and `(n = S t)`. As of version 1.2.2, the Ωmega typechecker solves such equations using a form of narrowing.

18.1 Solving Type-Checking Equations Using Narrowing

Ωmega uses a number of techniques to solve equations which arise from type checking. One of these is Narrowing. Narrowing is a well studied computational mechanism, and is the primary means of computation in the functional logic language Curry [He06, HS99]. Narrowing combines the power of reduction and unification. Narrowing can be used to compare two terms (each containing function calls and variables) and decide if they are equal. Unlike unification, the terms being compared can contain functions, and unlike reduction, the terms being simplified can contain variables. Narrowing finds bindings for some free variables in a term, such that the term reduces to a normal form. Narrowing is a special purpose constraint solving system, that can answer some of the equivalence questions asked when there are functions at the type level. In order to implement narrowing with a sound and complete strategy, we require type functions to be written in inductively sequential form (see Section 18.2.1). In earlier versions of Ωmega, narrowing played a larger role in solving type-checking equations. With the advent of the `theorem` declaration (Section 19), this role has diminished, but it is still used to some degree.

18.1.1 How Narrowing Works

Narrowing finds bindings for some free variables in the term being narrowed, once instantiated, these bindings allow the term to reduce to a normal form.

If a term contains constructors, function symbols, and variables, it often cannot be reduced. Usually, because function calls within the term do not match any left-hand side of any of their definitions. The failure to match is caused by either variables or other function calls in positions where the function definitions have only constructor patterns. Consider narrowing `(plus a Z == Z)` (checking whether `{plus a Z}` is equal to `Z`).

This cannot be reduced because `plus` inducts over its first argument with the patterns `Z` and `(S n)`. But in `(plus a Z == Z)`, the first argument position is a variable (`a`). Narrowing proceeds by guessing instantiations for the variable `a` – either `{a -> Z}` or `{a -> (S m)}`, and following both paths.

`plus:: Nat ~> Nat ~> Nat`
`{plus Z m} = m`
`{plus (S n) m}= S {plus n m}`
	guess `{a -> Z}`
	
	`({plus Z Z} == Z)`
	`(Z == Z)`
	
	Success !
		guess `{a -> (S m)}`
		
		`({plus (S m) Z} == Z)`
		`(S {plus m Z} == Z)`
		
		Failure.

The returned solutions are the bindings obtained in every successful path. In the example above we get a list of one solution: `[{a -> Z}]`. Some problems have no solutions, some have multiple, and some even have infinite solutions. Consider `{plus x 2t}`, we get `2t` when `{x -> 0t}`, and `3t` when `{x -> 1t}`, and `4t` when `{x -> 2t}` etc.

Narrowing works best when we have a problem with many constraints. The constraints prune the search path resulting in few solutions. If we're type checking with narrowing we hope there is exactly one solution. Consider narrowing `({plus x 3t} == 5t)`, Guessing `{x -> Z}` and `{x -> S z1}` we get the two paths:

```
1)   { 3t == 5t }
2)   { (1+{plus z1 3t})t == 5t }
```

The first path fails, on the second path we take a single reduction step leaving:

```
{ {plus z1 3t} == 4t }
```

Guessing `{z1 -> 0}` and `{z1 -> S z2}` we get the two paths:

```
1)   { 3t == 4t }
2)   { (1+{plus z2 3t})t == 4t }
```

The first path fails, on the second path we again take a single reduction step leaving:

```
{ {plus z2 3t} == 3t }
```

Guessing `{z2 -> 0}` and `{z2 -> S z3}` we get the two paths:

```
1)   { 3t == 3t }
2)   { (1+{plus z3 3t})t == 3t }
```

The first succeeds, and the second eventually fails, leaving us with only one solution `{ x -> 2t }`. We have found narrowing to be a efficient and understandable mechanism for directing computation at the type level.

18.2 Narrowing and Type-checking

Narrowing plays a role in type checking value-level functions that mention type functions in their type. For example consider:

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

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

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

The left-hand-side equalities let us assume `n` = `(S b)`. The right-hand-side equalities, require us to establish that `{plus n m}` = `(S{plus b m})`. Using the assumption that `n` = `(S b)`, we are left with the requirement that `{plus (S b) m}` = `(S{plus b m})`, which is easy to prove using the definition of `plus`. Narrowing is one mechanism that could solve this kind of equation. We are currently using narrowing, but are evaluating its effectiveness for future releases of Ωmega.

18.2.1 Narrowing Strategies

While narrowing is non-deterministic, it is both sound and complete with an appropriate strategy[Ant05]. All answers found are real answers, and if there exists an answer, good strategies will find it. When a question has an infinite number of answers, a good implementation will produce these answers lazily. In our type-checking context, finding 2 or more answers is a sign that a program being type checked has an ambiguous type and needs to be adjusted. In the rare occurrence that narrowing appears to diverge on a particular question, we can safely put resource bounds on the narrowing process, declaring failure if the resource bounds are exceeded. The consequence of such a declaration, is the possibility of declaring a well-typed function ill-typed. In our experience this rarely happens.

We restrict the form of function definitions at the type level to be inductively sequential[Ant92]. 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 to 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.

We pay for the generality of narrowing over unification and reduction by a modest increase in overhead. Narrowing uses a general purpose search algorithm rather than a special purpose unification or reduction engine. Narrowing is Turing complete, so we can solve any problem that can be solved by reduction, and many more.

19 The `theorem` Declaration

The theorem declaration is a primary way of directing the type checker. It uses ordinary terms, with types that can be read logically, as specifcations for new type checking strategies. There are currently three types of theorems. Each is used to direct the type checker to do things it might not ordinarily do. This is best seen by example.

19.1 `Equal` Types as Rewrite Rules

Narrowing alone is quite weak. Suppose we had declared the type of `app` as:

```
app:: List n a -> List m a -> List {plus m n} a
```

I.e. we switched the order of `m` and `n` in the range of `app`, writing `(List {plus m n} a)`, rather than `(List {plus n m} a)`. Semantically, this is a valid type for `app`, it is just too hard for the current mechanism to check its validity. The type checker has to solve `(S{plus m t} = {plus m (S t)})`, which matches none of the rewrite rules defining `plus`, and which leads to an infinite set of solutions if narrowing is used.

The solution is to augment the narrowing system with a set of semantically valid rewrite rules, and to apply these rules at the proper time. Rewrite rules are validated by exhibiting a terminating term with type: `(Equal lhs rhs)` and using this type as a left to right rewrite rule.

We use `Equal` terms (see Section 8) in the `theorem` declaration. The declaration:

```
theorem name = term
```

checks that `term` is terminating, and its type is of the form: `conditions -> (Equal lhs rhs)`. If this is so, and the conditions are met, the rewrite-rule `lhs --> rhs` is enabled in the scope of the declaration. The declaration has no run-time meaning. The term is not evaluated at runtime, and only its type is used in the scope of the declaration, and only at type-checking time. An example use of the `theorem` declaration is in the definition of a term with type: `(Nat' x -> Equal {plus x (S y)} (S{plus x y}))`

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

Because this term is recursive, but terminating, we can consider it a proof by induction of the theorem embodied in its type. To illustrate how this knowledge is used, consider the information in the type-checking debugger break-point, caused by the `check` clause in the definition of `plusS` above:

```
*** Checking: Eq
*** expected type: Equal {plus (1+_a)t (1+_m)t} (1+{plus (1+_a)t _m})t
***    refinement: {_n=(1+_a)t}
***   assumptions: Nat' _a
***      theorems: Rewrite indHyp: [] => {plus _a (1+'b)t} --> (1+{plus _a 'b})t
```

Note we need to show that `{plus (1+_a)t (1+_m)t}` is equal to `(1+{plus (1+_a)t _m})t`. Reduction leads to the equality `(1+{plus _a (1+_m)t})t == (2+{plus a _m})t`. Applying the rewrite we get the expected result: `(2+{plus _a _m})t == (2+{plus a _m})t`.

20 Types as Propositions

A `data`type can be declared to be a proposition by the use of the `prop` declaration. A `prop` declaration is identical in form to a `data` declaration. It introduces a new type constructor and it's associated (value) constructor functions. But it also informs the compiler that this type can be used as a static level proposition which can be used as a constraint. For example consider the proposition `Le` (which is a slight variant of the `LE` witness from section 16).

```
prop Le :: Nat ~> Nat ~> *0 where
  Base:: Le Z a
  Step:: Le a b -> Le (S a) (S b)
```

The type `LE` is introduced as a witness type with constructor functions `Base` and `Step`. These construct ordinary values. But the type `LE` can now also be used as static constraint. For example, we might define statically ordered lists as follows:

```
data SSeq:: Nat ~> *0 where
  Snil:: SSeq Z
  Scons:: Le b a => Nat' a -> SSeq b -> SSeq a
```

Note that the type of `Scons` contains an `Le` proposition as a constraint:

```
Scons :: forall (a:Nat) (b:Nat) . Le b a => Nat' a -> SSeq b
```

These constraints are propagated like equality constraints (or like class constraints in Haskell). For example, the term: `\ x y z -> Scons x (Scons y z)` is assigned the constrained type:

```
(LE a b,LE b c) => Nat' c -> Nat' b -> SSeq a -> SSeq c
```

The compiler uses the type of the constructor functions for `Le` to build the following constraint solving rules.

```
Base: Le Z a -->

Step: Le (S a) (S b) --> Le a b
```

These rules can be used to satisfy obligations introduced by propositional types.

20.1 Types as Back-chaining Rules

The user can introduce new propositional facts by writing functions over types introduced by the `prop` declaration. For example we can show that `LE` is transitive by exhibiting the total function `trans` with the following type.

```
trans :: Le a b -> Le b c -> Le a c
trans Base Base = Base
trans (Step z) Base = unreachable
trans Base (Step z) = Base
trans (Step x) (Step y) = (Step(trans x y))
```

Since this function is total, its type becomes a new rule that can be used to satisfy propositional obligations. We can activate this rule in some scope by using the `theorem` declaration. For example:

```
f x = body
  where theorem trans
```

Observes that the type of `trans` can be used as a back-chaining predicate solver and adds the following rule in the scope of the `where` clause. The rule:

```
trans: (exists d) [Le b d, Le d c] => Le b c --> []
```

says that we can satisfy `(Le b c)` if we can find a concrete `d` such that (both) `(Le b d)` and `(Le d c)` are satisfied.

21 Unreachable Clauses

Type indexes to GADTs allow the user to make finer distinctions than when using ordinary algebraic datatypes. Sometimes such distinctions cause a clause in a function definition to become unreachable. For example consider the second clause in the definition of `transP` below:

```
trans :: Le a b -> Le b c -> Le a c
trans Base Base = Base
trans (Step z) Base = unreachable
trans Base (Step z) = Base
trans (Step x) (Step y) = (Step(trans x y))
```

The pattern `(Step z)` has type `(Le (S i) (S j))` when we know `a = (S i)` and `b = (S j)`. The pattern `Base` has type `(Le Z k)` when we know `b = Z` and `c = k`. These sets of assumptions are inconsistent, since `b` can't simultaneously be equal to `Z` and `(S i)`. So the clause in the scope of these patterns is unreachable. There are no well-typed arguments, to which we could apply `trans`, 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 knows that this clause is unreachable, and to help document that the clauses exhaustively cover all possible cases.

21.1 Types as Refinement Lemmas

Consider the function `half` defined below. Given a natural number whose size is expressed as the sum of a number with itself, it returns another number with size exactly half of the original. The type tells us the function is undefined on numbers whose size is not expressible as the sum of a number with itself.

```
plus :: Nat ~> Nat ~> Nat
{plus Z y} = y
{plus (S x) y} = S{plus x y}

half:: Nat' {plus n n} -> Nat' n
half Z = check Z
half (S Z) = unreachable
half (S (S x)) = S(half x)
```

The first and third equations generate the following type checking equations.

expected type	`Nat' {plus n n}` → Nat' n
equation	half Z = Z
computed type	Nat' Z → Nat' Z
equalities	(Equal {plus n n} Z) ⇒ (Equal n Z)

expected type	`Nat' {plus n n}` → Nat' n
equation	half (S (S x)) = (S(half x))
computed type	Nat' (S(S a)) → Nat' (S c)
equalities	(Equal {plus n n} (S(S a))) ⇒ (Equal n (S c),Equal a {plus c c})

The current system cannot solve the given constraints. The hypothesis are in terms of the type-function call `{plus n n}`. What we need is to direct the type checker to take these facts and discover facts about `n`, instead of facts about `{plus n n}`. Such facts are exhibited in the types of the following two functions.

```
nPlusN2:: Nat' n -> Equal (S (S m)) {plus n n} ->
                    exists k . (Equal n (S k),Equal m {plus k k})
nPlusN2 Z Eq = unreachable
nPlusN2 (S y) Eq = Ex(Eq,Eq)
  where theorem plusS

nPlusN0:: Nat' n -> Equal Z {plus n n} -> Equal n Z
nPlusN0 Z Eq = Eq
nPlusN0 (S y) Eq = unreachable
```

These functions are typechecked by a case analysis. In each function one of the clauses is unreachable, because the expected type is inconsistent with the type of the pattern in the definition. By introducing these terms as theorems in the definition of `half` we gain extra information. Consider

```
half:: Nat' {plus n n} -> Nat' n
half Z = Z                     where theorem nPlusN0  -- introduces (Equal n Z)
half (S Z) = unreachable
half (S (S x)) = S(half x)     where theorem nPlusN2
                                     -- introduces (Equal n (S c),Equal a {plus c c})
```

21.2 General Use of the `theorem` Declaration

Theorems are added by the `theorem` declaration. The general form of a `theorem` declaration is the keyword `theorem` followed by 1 or more (comma separated) theorems. Each theorem is either an *identifier* or an *identifier* = *term*. For example one may write:

```
f:: Nat' n -> Int
f m = 5
  where theorem trans, indHyp = plusS m, nPlusN0
```

Here, three theorems are introduced. They are named `trans`, `indHyp`, and `nPlusN0`. The body of the theorem is taken from the types of the terms `trans`, `(plusS m)` and `S`. If a theorem does not have an associated term (as is the case for `trans`), the variable in scope with the same name as the theorem is used. In the scope of the body of `f` the theorems are:

```
BackChain trans: (exists 'd) [Le 'b 'd, Le 'd 'c] => Le 'b 'c --> []
Rewrite indHyp: [] => {plus _n (1+'e)t} --> (1+{plus _n 'e})t
Refinement nPlusN0: [Nat' 'a] => Equal 0t {plus 'a 'a} --> [Equal 'a 0t]
```

Note how the body of the theorem derives from the type of the term in the `theorem` declaration. As discussed in Sections 19.1, 20.1, and 21.1, there are currently three uses for theorems:

1. As left to right rewrite rules. The type has the form (`Equal` *lhs rhs*) the *lhs* must be a Type constructor call, or a type function call. The *rhs* is used to replace a term matching the *lhs* when narrowing.

2. As a backchaining rule: `P x -> Q y -> S x y` where `P`, `Q`, and `S` are propositions. The intended semantics of a back chaining theorem is if one is trying to establish `(S x y)` as a predicate one may establish the set of predicates `{ (P x), (Q y) }` instead.

3. As a refinement lemma: `cond -> Equal f g -> (Equal x t,Equal y s)` where `f`, `g`, `s`, and `t` are arbitrary terms, but `x` and `y`, are variables. The intended semantics of a refinement lemma is to add additional facts to the set of assumed truths. These new facts are simpler in form (they equate variables to terms) than the old fact they are derived from.

22 Syntax Extensions

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 five 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, record-like, and unary-increment types. Some examples in the supported syntax are: `[4,5]i`, `(2+n)j` `(4,True)k`, `{"a"=5, "b"=6}h`, and `(x'3)w`. In general, the syntax starts with list-like, natural-number-like, record-like, pair-like, or unary-increment 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, pair-like, and unary-increment types). Each type of syntax, has an asociated tag-table, so the same tag can be used once for each kind of syntax. 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 semicolon 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)
```

The unary-increment syntax associates one function with each tag. This function plays the role of an increment function `Tick`. For example if the tag "`w`" is associated with the function `Tick`, then the expansion is as follows.

```
(x`0)w     ---> x
(x`1)w     ---> Tick x
(x`2)w     ---> Tick(Tick x)
(x`3)w     ---> Tick(Tick(Tick x))
```

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. Ωmega 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. As of the March 2010 release there are two style of syntax derivations: old-style, and new-style. Both are supported. The new-style is more expressive, and all old-style derivations can be replaced by equivalent new-style ones. Eventually the old-style will be deprecated and removed from the system.

22.1 Old-style derivations

Old-style derivations specify a single tag to be associated with a single syntax extension for a particular dataype. Using the old-style, each datatype can can be associated with a sungle syntax extension, and the number of constructor functions of the datatype must agree with the number of functions associated with that extension. For example to use the List extension the dataype must have two constructors, and to use the Pair extension the datatype must have one constructor.

22.1.1 Old-style List Example: Tuples are Lists that Reflect the Type of Their Components

An example where the list-like extension is used both at the value and type level follows. It generalizes the `Prod-Tuple` example from Section 11.

```
data Prod ::  *1 ~> *1 where
   Nil  :: Prod a
   Cons :: a ~> Prod a ~> Prod a
 deriving List(a)   -- the List tag "a" is associated with (Nil,Cons) of Prod

data Tuple :: forall (a:: *1). Nat ~> Prod a ~> *0 where
   NIL  :: Tuple Z Nil
   CONS :: a -> Tuple n l -> Tuple (S n) (Cons a l)
 deriving List(b)   -- the List tag "b" is associated with (NIL,CONS) of Tuple
```

We can construct value-level `Tuple` and type-level `Prod` using the special list-like syntax and the tags “`a`” and “`b`”. A `Tuple` is a heterogenous list with indexed length. The type of the elements in the list are reflected in the type of the list as a `Prod` at the type level. For example we evaluate a value level-list, and ask the system to compute the kind of a type-level list.

```
prompt> [3,True,3.4]b
[3,True,3.4]b : Tuple 3t [Int,Bool,Float]a

prompt> :k [Int,Bool]a
[Int,Bool]a :: Prod *0
```

Note how the lists are both entered and displayed with the list-syntax with the appropriate tags. We can even use the list-sytax as a pattern in a function definition.

```
testfun :: Tuple (n+2)t [Int,Bool; w]a -> (Int,Bool,Tuple n w)
testfun [x,y;zs]b = (x+1,not y,zs)
```

22.1.2 Old-style Nat example: A n-ary Summing Function

An example of a type that makes good use of the natural-number like syntax is the type `SumSpec`.

```
data SumSpec :: *0 ~> *0  where
   Zero :: SumSpec Int
   Succ :: SumSpec a -> SumSpec (Int -> a)
 deriving Nat(s)   -- associates the Nat tag "s" with (Zero,Succ)
```

The idea behind `SumSpec` is that types of the terms in the series `Zero`, `(Succ Zero)`, `(Succ(Succ Zero))`, is each of the form `SumSpec i`, i.e. `SumSpec Int`, `SumSpec(Int -> Int)`, `SumSpec(Int -> Int -> Int)`, etc. Each `i` in the series is the type of the corresponding function in the series of functions below.

```
0
\ x -> x
\ x -> \ y -> x+y
\ x -> \ y -> \ z -> x+y+z
```

This series of terms are the functions that sum 0, 1, 2, 3, etc. integers. We can can observe the relationship between the types of the terms and the type indexes of `SumSpec`, by typing a series of `SumSpec` examples into the Ωmega interactive loop.

```
prompt> 0s
0s : SumSpec Int
prompt> 1s
1s : SumSpec (Int -> Int)
prompt> 2s
2s : SumSpec (Int -> Int -> Int)
prompt> 3s
3s : SumSpec (Int -> Int -> Int -> Int)
```

We can now write a generic function that sums any number of integers.

```
sum:: SumSpec a -> a
sum x = sumhelp x 0
  where sumhelp:: SumSpec a -> Int -> a
        sumhelp Zero n = n
        sumhelp (Succ x) n = \ m -> sumhelp x (n+m)
```

By using the natural-number like syntax we get a pleasant interface to the use of `sum`. Observe the example use in the interactive session below:

```
prompt> sum 0s
0 : Int
prompt> sum 1s 3
3 : Int
prompt> sum 2s 4 7
11 : Int
prompt> sum 3s 3 5 1
9 : Int
prompt> sum 3s 3 5
<fn> : Int -> Int
```

22.1.3 Old-style Record example: Records and Rows

Finally, we demonstrate how using Tags and Labels (see Section 13) we can roll our own record structures.

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

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

The `Row` and `Record` type are like the `Prod` and `Tuple` type, but they *cons* a *pair* of things on to linear sequence, rather than a single thing. By specializing the first part of the pair to a `Label` we can build labeled tuples or records.

```
prompt> { `name="tim", `age=21}
{`name="tim",`age=21} : Record {`name=[Char],`age=Int}r
```

Here we demostrate that the tag for `Records` is the empty tag. The empty tag for `Nat` is reserved for ordinary `Int`, the empty tag for `List` is reserved for ordinary lists, and the empty tag for `Pair` is reserved for Ωmega's builtin binary product.

22.2 New-style derivations

The old-style syntax extension allows the user to choose only one style of syntactic extension per datatype declaration, and limits the number of constructors of the datatype to exactly the number of syntax functions associated with the extension. The new-style of derivation lifts both these restrictions.

22.2.1 New-style Unary increment example: Cdr

This example illustrates why it is often desireable to allow more constructors than those supported by a single extension. It is inspired by the family of Lisp functions *cdr*, *cddr*, *cdddr*, *cddddr*, etc. Which select the nth second component of a set of nested pairs, where n corresponds to the number of ds. The idea is to write `(x`n)c` as specification for the function (c$d^n$r y), where $c$ is the syntax extension tag for the datatype. The idea is to support a `Projection` specification type, such that `Projection a b` specifies a projection from the type `a` to the type `b`, and the function `cdr:: (Projection a b) -> a -> b`. For example the partial applications of `cdr` will have the following types:

```
cdr (Id`0)p :: a -> a
cdr (Id`1)p :: (a,b) -> b
cdr (Id`2)p :: (a,(b,c)) -> c
cdr (Id`3)p :: (a,(b,(c,d))) -> d
```

We define this by using the following dattype declaration with a new-style `Tick` extension.

```
data Projection :: *0 ~> *0 ~> *0 where
  Car:: Projection e t -> Projection (e,s) t
  Cdr:: Projection e t -> Projection (s,e) t
  Id:: Projection t t
 deriving syntax(p) Tick(Cdr)
```

In a newstyle extention, the keyword `syntax` is followed by the tag, and the types of extensions are followed by the names of the constructors that will be associated with that extensions functions. Thus `(x‘2)p` stands for `(Cdr(Cdr x))`. Note the types of a few terms constructed this way.

```
zero = (Id‘0)p :: Projection a a
one  = (Id‘1)p :: Projection (a,b) b
two  = (Id‘2)p :: Projection (a,(b,c)) c
```

Of course the purpose of this type is to serve as the specification for a family of `cdr` functions. We write this below, illustrating that the syntax extension can also be used in the pattern matching language.

```
cdr :: Projection a b -> a -> b
cdr Id x = x
cdr (Car x) (a,b) = cdr x a
cdr (x‘1)p  (a,b) = cdr x b
```

22.2.2 New-style multiple syntax example: Test

Finally we illustrate that one datatype can support multiple syntax extensions.

```
data Test:: *0 where
  Nil :: Test
  Cons :: Test -> Test -> Test
  RNil :: Test
  RCons :: String -> Test -> Test -> Test
  Zero :: Test
  Succ :: Test -> Test
  Pair :: Test -> Test -> Test
  Next :: Test -> Test
  A:: Test
 deriving syntax(w) List(Nil,Cons) Nat(Zero,Succ)
                    Pair(Pair) Tick(Next) Record(RNil,RCons)
```

This allows one to write things like the following.

```
test = { "name" = A
       , "age"= 2w
       , "xs" = [ (A‘1)w ]w
       , "ps" = (0w,1w)w }w
```

Rather than the much more verbose

```
verbose = RCons "name" A (
          RCons "age" (Succ (Succ Zero)) (
          RCons "xs" (Cons (Next A) Nil) (
          RCons "ps" (Pair Zero (Succ Zero)) RNil)))
```

23 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 simliar 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`) types indexable by types at all levels. For example consider the kind of the type constructor `Equal` and the type of its constructor `Eq`.

```
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 (Prod *0))` would not be, since both `Nat` and `(Prod *0)` 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. A similar restriction would make Row kinds less useful without level polymorphism.

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

24 Tracing

The Ωmega system includes a number of type-checking time tracing mechanisms that work well with the type-checking interactive loop (Section 16). The tracing mechanisms can be controlled by using the `:set` and `:clear` command on the system modes. These commands are accessed as one of the command level prompts (Section 15).

24.1 System Modes for Tracing

The commands level `:set` and `:clear` control the Ωmega system mode variables. The mode variables control the tracing behavior of the system. The modes allow the user to display internal and intermediate type information for debugging, or understanding purposes. The available modes are:

- `solving`. Initial value = `False`. Turns on tracing in the predicate solving engine.
- `predicate_emission`. Initial value = `False`. Reports the fact that a predicate has been emitted. Predicates are collected at generalization sites, and are either solved, or abstracted into constrained types.
- `narrowing`. Initial value = `False`. Shows the steps taken when narrowing. Useful for debugging when narrowing does not return the result desired.
- `theorem`. Initial value = `False`. Reports when a lemma is added by a 'theorem' declaration, and when such a lemma is applicable.
- `kind`. Initial value = `False`. Displays kinds of subterms when using the :k or :t commands.
- `unreachable`. Initial value = `False`. Displays information when checking for unreachable clauses.
- `verbose`. Initial value = `False`. For Debugging the compiler

To illustrate the use of the tracing modes, we will trace type checking the function `app`. Note the insertion of the `check` keyword in the second clause.

```
app::List n a -> List m a -> List {plus n m} a
app Nil ys = ys
app (Cons x xs) ys = check Cons x (app xs ys)
```

This allows the user to selectively control when the tracing is to be in effect. Note in the transcript below, the use of the `:set narrow` command, to turn narrowing tracing on, and then, the use of the `:q` command, to quit the interactive debugger. Now as type-checking continues from this point, narrowing trace information is printed. While tracing, the trace output is paused, and the user is asked to `step` or `continue`.

```
*** Checking: Cons x (app xs ys)
*** expected type: List {plus (1+_c)t _m} _a
***    refinement: {_b=_a, _n=(1+_c)t}
***   assumptions:
***      theorems:
check> :set narrow
check> :q
Norm {plus (1+_c)t _m} ---> (1+{plus _c _m})t

####################c
Solve By Narrowing: Equal {plus (1+_c)t _m} (1+{plus _c _m})t
Collected by type checking in scope case 9.
line: 219 column: 1
app (Cons x xs) ys = (Check Cons x (app x ...
Normal form: Equal (1+{plus _c _m})t (1+{plus _c _m})t
Assumptions:
   Theorems:

-------------------------------------
25 Narrowing the list (looking for 3 solutions) found 0
   Equal (1+{plus _c _m})t (1+{plus _c _m})t

with truths:
   and()

press return to step, 'c' to continue:

-------------------------------------
24 Narrowing the list (looking for 3 solutions) found 0
   Equal {plus _c _m} {plus _c _m}

with truths:
   and()

press return to step, 'c' to continue:

*********************
Found a solution for:
  Equal (1+{plus _c _m})t (1+{plus _c _m})t

Answers = {}
```

25 Freshness

Ωmega includes an experimental implementation of Pitts and Gabbay's fresh types[GP02]. The interface to this mechanism is the type `Symbol`, and the functions:

```
fresh:: Char -> Symbol
swap:: Symbol -> Symbol -> a -> a
symbolEq:: Symbol -> Symbol -> Bool
freshen:: a -> (a,[(Symbol,Symbol)])
```

See the paper for how these might be used.

26 Resource Bounds

In Ωmega, there are several resource bounds that can be controlled by the user. The current resource bounds control the number of steps taken while narrowing, and the number of times a backchaining theorem can be applied. The narrowing bound is important because narrowing may find an inifinite number of solutions to a certain problem, and the resource bound cuts off such search. The backchaining bound is important because some theorems (especially commutative and associative theorems) can lead to infinite rewriting sequences. These bounds are controlled using the `:bounds` command in the toplevel loop. The command `:bounds` with no argument lists the current resource bounds and their values. The command (`:bounds` *bound n*), sets the bound *bound* to *n*. This is illustrated in the transcript below:

```
prompt> :bounds
narrow = 25   Number of steps to take when narrowing.
backchain = 4 Number of times a backChain lemma can be applied.

prompt> :bounds narrow 30

prompt> :bounds
narrow = 30   Number of steps to take when narrowing.
backchain = 4 Number of times a backChain lemma can be applied.
```

27 More Examples

In appendices A and B are two short Ωmega programs that illustrate the use of the `theorem` declaration and the use of static constraints. In addition, there are many more examples of use of Ωmega in the papers listed below:

- Meta-Programming with Typed Object-Language Representations[PL04]
- Meta-programming with Built-in Type Equality[SP04]
- Languages of the Future[She04]
- Programming with Static Invariants in Omega[SL04]

- GADTs, Refinement Types, and Dependent Programming[SHL05]
- Putting Curry-Howard to Work[She05b]
- Playing with Types[She05a]
- Type-Level Computation Using Narrowing in Omega.

  All these papers are available at `http://web.cecs.pdx.edu/~sheard`

References

[Ant92] Sergio Antoy. Definitional trees. In *ALP*, pages 143–157, 1992.

[Ant05] Sergio Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, July 2005.

[Aug99] Lennart Augustsson. Cayenne — a language with dependent types. *ACM SIGPLAN Notices*, 34(1):239–250, January 1999.

[Aug00] Lennart Augustsson. Equality proofs in Cayenne, July 11 2000. `http://www.cs.chalmers.se/~augustss/cayenne/eqproof.ps`.

[CD94] T. Coquand and P. Dybjer. Inductive definitions and type theory: an introduction. (preliminary version). *Lecture Notes in Computer Science*, 880:60–76, 1994.

[Coq] Catarina Coquand. Agda is a system for incrementally developing proofs and programs. Web page describing AGDA: `http://www.cs.chalmers.se/~catarina/agda/` .

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[CX05a] Chen and Xi. Combining programming with theorem proving. In *Proceedings of the 2005 ACM/SIGPLAN International Conference on Functional Programming (ICFP'05)*, pages 66 – 77, September 2005.

[CX05b] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *ICFP 2005*, 2005. `http://www.cs.bu.edu/~hwxi/`.

[Dav97] 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.

[DS99] P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. *Lecture Notes in Computer Science*, 1581:129–146, 1999.

[FI00] Daniel Fridlender and Mia Indrika. Do we need dependent types? *J. Funct. Program*, 10(4):409–415, 2000.

[GP02] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3–5):341–363, July 2002.

[HC02] 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.

[He06] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry`, March 28, 2006.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[HS99] Michael Hanus and Ramin Sadre. An abstract machine for curry and its concurrent implementation in java. *Journal of Functional and Logic Programming*, 1999(Special Issue 1), 1999.

[JS03] Simon Peyton Jones and Mark Shields. Practical type inference for arbitrary-rank types. Technical report, Microsoft Research, "December" 2003. `http://research.microsoft.com/Users/simonpj/papers/putting/index.htm.`

[LP92] 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.

[Mcb01] Conor Mcbride. First-order unification by structural recursion, August 16 2001.

[McB04] 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 .`

[Nor96] Bengt Nordstrom. The ALF proof editor, March 20 1996.
`ftp://ftp.cs.chalmers.se/pub/users/ilya/FMC/alfintro.ps.gz`.

[Pau90] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[Pey03] Simon Peyton Jones. Special issue: Haskell 98 language and libraries. *Journal of Functional Programming*, 13, January 2003.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181, Cambridge, England, 1991. Cambridge University Press.

[PL04] 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.

[PS99] 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.

[SH00] 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.

[She99] T. Sheard. Using MetaML: A staged programming language. *Lecture Notes in Computer Science*, 1608:207–239, 1999.

[She03] Tim Sheard. A pure language with default strict evaluation order and explicit laziness, August 2003. Available from: `http://www.cs.pdx.edu/~sheard/` .

[She04] Tim Sheard. Languages of the future. *Onward Track, OOPSLA'04. Reprinted in: ACM SIGPLAN Notices, Dec 2004*, 39(10):116–119, October 2004.

[She05a] Tim Sheard. Playing with types. Technical report, Portland State University, 2005. `http://www.cs.pdx.edu/~sheard`.

[She05b] Tim Sheard. Putting Curry-Howard to work. In *Proceedings of the ACM SIGPLAN 2005 Haskell Workshop*, pages 74–85, 2005.

[SHL05] Tim Sheard, Jim Hook, and Nathan Linger. Gadts, refinement types, and dependent programming, March 2005. Available from: `http://www.cs.pdx.edu/~sheard/` .

[SL04] Tim Sheard and Nathan Linger. Programming with static invariants in omega, September 2004. Available from: `http://www.cs.pdx.edu/~sheard/` .

[SP04] Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. In *Logical Frameworks and Meta-Languages workshop*, July 2004. Available at: `http://cs-www.cs.yale.edu/homes/carsten/lfm04/`.

[SPJ02] T. Sheard and S. Peyton-Jones. Template meta-programming for Haskell. In *Proc. of the workshop on Haskell*, pages 1–16. ACM, 2002.

[SSTP02] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. *ACM SIGPLAN Notices*, 37(1):217–232, January 2002.

[Stu04] 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/`.

[The03] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 7.4*. INRIA, 2003. `http://pauillac.inria.fr/coq/doc/main.html`.

[TS00] Walid Taha and Tim Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2), 2000.

[WSW05] 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/`.

[Xi97] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1997.

[Xi04] Hongwei Xi. Applied type systems (extended abstract). In *In post-workshop proceedings of TYPES 2003*, pages 394 – 408, 2004. Springer-Verlag LNCS 3085.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. *ACM SIGPLAN Notices*, 33(5):249–257, May 1998.

[XP99] 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 Proofs by Induction over the Natural Numbers

–

```
----------------------------------------------------
-- This file depends upon the predefined GADTs
-- kind Nat = Z | S Nat
--
-- prop Nat' :: Nat ~> *0 where
--   Z:: Nat' Z
--   S:: forall (a:: Nat) . Nat' a -> Nat' (S a)
--
-- data Equal :: forall (a:: *1) . a ~> a ~> *0 where
--   Eq:: forall (b:: *1) (x:: b) . Equal x x

---------------------------------------------------
-- define plus as a recursive type-function

plus:: Nat ~> Nat ~> Nat
{plus Z m} = m
{plus (S n) m} = S {plus n m}  -- deriving Relation Plus

---------------------------------------------------
-- plusZ is proved by induction. Note how we use
-- a recursive call "plusZ m" to produce the induction
```

```
-- hypothesis, and how we use the "theorem" decl.

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

---------------------------------------------------
-- plusS is proved in a manner similar to plusZ

plusS :: Nat' n -> Equal {plus n (S m)} (S{plus n m})
plusS Z  = Eq
plusS (S x)  = Eq
  where theorem ih = plusS x

---------------------------------------------------
-- we use both induction, and plusZ and plusS to
-- show that plus commutes.

plusCommutes :: Nat' n -> Nat' m -> Equal {plus n m} {plus m n}
plusCommutes Z m = Eq
   where theorem lemma = plusZ m
plusCommutes (S x) m = Eq
   where theorem plusS,
                 indHyp = plusCommutes x m

-------------------------------------------------
-- We can prove similar results, first by defining
-- plus as a ternary relation "Plus". This
-- relation is a witness type.

data Plus:: Nat ~> Nat ~> Nat ~> *0 where
  Plus0 :: Plus Z m m
  Plus1 :: Plus n m c -> Plus (S n) m (S c)

-------------------------------------------------
-- the relational analog to plusS

pluS:: Plus a b c -> Plus a (S b) (S c)
pluS Plus0 = Plus0
pluS (Plus1 p) = Plus1(pluS p)


------------------------------------------------
-- the relational analog to plusCommutes
```

```
plusCom:: Nat' b -> Plus a b c -> Plus b a c
plusCom Z Plus0 = Plus0
plusCom (S n) Plus0 = Plus1(plusCom n Plus0)
plusCom Z (Plus1 p) = pluS(plusCom Z p)
plusCom (S n) (Plus1 p) = pluS(plusCom (S n) p)

--------------------------------------------------
-- To show that the two approaches are equivalent
-- show that we can convert or translate from each
-- to the other.

trans:: Plus a b c -> Equal {plus a b} c
trans Plus0 = Eq
trans (Plus1 p) =  Eq
  where theorem indHyp = trans p

transInv :: Nat' a -> Equal {plus a b} c -> Plus a b c
transInv Z Eq = Plus0
transInv (x@(S n)) (p@Eq) = Plus1(transInv n Eq)

---------------------------------------------------
-- Show that the relation Plus is functional. I.e.
-- if we know the first two argument, we can
-- determine the third.

funcDepend:: (Plus a b c,Plus a b d) -> Equal c d
funcDepend (Plus0,Plus0) = Eq
funcDepend (Plus0,Plus1 p) = unreachable
funcDepend (Plus1 p,Plus0) = unreachable
funcDepend (Plus1 p,Plus1 q) =
   case funcDepend(p,q) of
     (Eq) -> Eq

--
```

B Quicksort Algorithm in Ωmega

–

```
--------------------------------------------------------
-- (LE n m) A witness that n is less than or equal to m

prop LE :: Nat ~> Nat ~> *0 where
  Base_LE:: LE Z a
```

```
  Step_LE:: LE a b -> LE (S a) (S b)

--------------------------------------------------------
-- A bounded sequence. The elements appear in no
-- particular order, but if a sequence has type
-- (BSeq min max), every element is between min and max
-- This is maintained by static constraints.

data BSeq :: Nat ~> Nat ~> *0 where
  Nil :: LE min max => BSeq min max
  Cons :: (LE min m, LE m max) => Nat' m -> BSeq min max -> BSeq min max

--------------------------------------------------------
-- helper function for applying one of two functions
-- over a sum type, depending upon the sum injection.

mapP :: (a -> b) -> (c -> d) -> (a+c) -> (b+d)
mapP f g (L x) = L(f x)
mapP f g (R x) = R(g x)

------------------------------------------------------
-- Comparison of two Nat' elements, returns one
-- of two possible witnesses.

compare :: Nat' a -> Nat' b -> (LE a b + LE b a)
compare Z _ = L Base_LE
compare (S x) Z = R Base_LE
compare (S x) (S y) = mapP Step_LE Step_LE (compare x y)

------------------------------------------------------
-- split a Bounded sequence into bounded sequences

qsplit :: (LE min piv, LE piv max) =>
          Nat' piv -> BSeq min max -> (BSeq min piv,BSeq piv max)
qsplit piv Nil = (Nil,Nil)
qsplit piv (Cons x xs) =
    case compare x piv of
      L p1 -> (Cons x small, large)
      R p1 -> (small, Cons x large)
  where (small,large) = qsplit piv xs

------------------------------------------------------
-- A sorted list. Elements are statically guaranteed
-- to appear in ascending order. Note that a list with
-- type (SL min max) has (Nat' min) as first element
```

```
-- and every other element less than or equal to max.
-- Note (Nat' max) may not actually be in the list.

data SL :: Nat ~> Nat ~> *0 where
  SNil :: SL x x
  SCons :: LE min min' => Nat' min -> SL min' max -> SL min max

------------------------------------------------------
-- append two sorted sequences.

app :: SL min piv -> SL piv max -> SL min max
app SNil ys = ys
app (SCons min xs) ys = SCons min (app xs ys)

------------------------------------------------------
-- rearranges a bounded list into a sorted list. Note
-- qsort1 maintains the static invariant that the
-- first element of the bounded list is less than or
-- equal to the first element of the output.

qsort1 :: BSeq min max -> exists t . LE min t => SL t max
qsort1 Nil = Ex(SNil)
qsort1 (x@(Cons pivot tail)) =  (Ex(app smaller' (SCons pivot larger')))
  where (smaller,larger) = qsplit pivot tail
        (Ex(smaller')) = (qsort1 smaller)
        (Ex(larger')) = (qsort1 larger)

--
```