Putting Curry-Howard to Work

Tim Sheard

Portland State University

sheard@cs.pdx.edu

Abstract

The Curry-Howard isomorphism states that types are propositions and that programs are proofs. This allows programmers to state and enforce invariants of programs by using types. Unfortunately, the type systems of today's functional languages cannot directly express interesting properties of programs. To alleviate this problem, we propose the addition of three new features to functional programming languages such as Haskell: Generalized Algebraic Datatypes, Extensible Kind Systems, and the generation, propagation, and discharging of Static Propositions. These three new features are backward compatible with existing features, and combine to enable a new programming paradigm for functional programmers. This paradigm makes it possible to state and enforce interesting properties of programs using the type system, and it does this in manner that leaves intact the functional programming style, known and loved by functional programmers everywhere.

Categories and Subject Descriptors D.3 [*Software*]: Programming Languages

General Terms Languages,Theory, Reliabilty, Verification

Keywords Logic, Curry-Howard Isomorphism, GADT, Haskell, Extensional Kind System

1. Introduction

Most functional programmers have certainly heard of the Curry-Howard Isomorphism – types are propositions and programs are proofs. But visualizing how this can be useful in their day-to-day programming tasks is unclear. The reason for this uncertainty can be traced to the weakness of the type system in common functional programming languages. The types expressible in Haskell and ML-like languages are too weak to express any but the most trivial properties of programs.

The advent of *Generalized Algebraic Datatypes* (GADTs) has changed this. GADTs allow programmer to express interesting properties about programs using types. Two example properties are: *sorting programs* whose outputs are guaranteed to be ordered, and *device drivers* whose interactions with the hardware always obey the correct protocols. This paper explains how such types can be exploited to create more robust, reliable, and trustworthy programs. In other words, this paper is about putting the Curry-Howard Isomorphism to work for programmers in a way that is both useful and easy to understand.

Haskell Workshop 2005 September 30, 2005, Tallinn, Estonia

2. The Ωmega Programming Language

This paper proposes certain kinds of extensions to functional languages like Haskell, Standard ML, and O'Caml. Some of these ideas, especially the addition of GADTs, have been argued before by the author[29, 32] and others[41, 8, 11, 6, 18, 3]. Some ideas, like the addition of an extensible kind system[30] have had fewer vocal advocates. Some, like the declaration, propagation, and solving of static propositions, are new to this paper. The goal of this paper is to encourage designers and implementers of functional language compilers to incorporate these ideas into their languages.

To demonstrate both the feasibility and utility of such an approach we have designed and implemented the programming language Ωmega. Descended from Haskell, Ωmega is a proof of concept demonstration. To make our lives easier (implementing several non-trivial extensions plus all of Haskell's normal features is beyond our abilities) we have removed some Haskell features from Ωmega. Most notably, Ωmega has no class system, and Ωmega has a strict evaluation policy. Removing the class system was a strictly pragmatic decision. We'd love to see the features proposed in this paper in a language with classes, but we have neither the time nor the resources to build such an implementation. The evaluation policy of Ωmega is more problematic and is discussed in detail in Section 15. Other than these two changes we have tried to keep all the other features of Haskell not affected by these changes intact. In particular, the syntax of Ωmega is Haskell syntax. An important design choice of Ωmega is to enforce the phase distinction between values and types. This choice distinguishes Ωmega from many systems based on dependent types (e.g. Cayenne), but helps Ωmega programs retain their Haskell look-and-feel. We have tried hard to only add features to Haskell that retain Haskell's programming style.

This paper tells a rather complex story that will culminate with the direct use of the Curry-Howard isomorphism to describe interesting properties of programs. To reach that end, we introduce a few tools along the way. The tools include several small (backward compatible) features to Haskell to support the use of types as propositions, and a number of interesting programming patterns that employ the new features.

The features we have added include Generalized Algebraic Datatypes (GADTs), the ability to extend the kind system to introduce new types with kinds other than *star*, and the ability to define and use type-based constraints to state and enforce static properties on programs. The programming patterns we describe include witness objects, singleton types, constructing static witness dynamically at run-time, and shifting between static and dynamic type checking. We discuss each of these in the next few sections. Each section is labeled either with *Feature* if it is an addition to Haskell, *Pattern* if it is a paradigmatic use of the features to accomplish a particular end, or *Example* if it illustrates the use of the Curry-Howard isomorphism to state and ensure an interesting property of a program.

3. Feature: Extensible Kind System

In Haskell, values are classified by types. Classification is a binary relation, and in Haskell it is indicated by the infix operator (`::`). We write: (`5:: Int`) and (`map:: (a->b) -> [a] -> [b]`), and we say: *5 has type int*. In the same way, types are classified by kinds. The classification of types by kinds is only *implicit* in Haskell programs. The user never mentions kinds explicitly, but the type checker uses kinds to see that types are well-formed. Kinds in Haskell are always inferred, and the range of kinds is extremely limited. Every possible kind in Haskell can be described by the following simple grammar with only two productions. *kind* → `*0`, and *kind* → *kind* `~>` *kind*. We use the symbol composed of *tilde* followed by *greater-than* (`~>`) as an arrow at the kind level, to distinguish it from the arrow at the type level (`->`), since these are semantically two very different things.

For example, using the same overloaded operator for kinding (`::`) as for typing, we write: `Int::*0`, and we say: *int* is classified (or has kind) *star-zero*. Other examples include: `Bool::*0`, and `(Char,Int)::*0`. Type constructors that require types as arguments, such as `IO`, `(->)`, and `[]` have higher order kinds. We write `IO::*0 ~> *0`, and `(->)::*0 ~> *0 ~> *0`, and `[]:: *0 ~> *0`.

In Haskell, all (fully applied) types, such as `Int`, `Bool`, and `(Char,Int)` have kind `*0`. In Ωmega we allow the user to introduce new kinds (other than `*0`), and to introduce new types that populate these kinds. For example, consider:

```
kind State = Locked | Unlocked | Error
kind Nat = Z | S Nat
kind Row x = RCons x (Row x) | RNil
```

The `kind` declaration is just like the `data` declaration in Haskell, except it introduces *new kinds* (`State`, `Nat`, and `Row`) rather than *new types*, and *new types* (`Locked`, `Unlocked`, `Error`, `Z`, `S`, `RNil`, and `RCons`) rather than *new constructor functions*. The names introduced all live in the type-name space (as opposed to the value-name space where variables, functions, and constructor functions live).

An important property of types introduced in `kind` declarations is that these new types do not classify any values. In both Ωmega and Haskell values are classified only by those types that are classified by the kind `*0`. For example,

```
5:: Int                   Int:: *0
id:: (a -> a)             (a -> a):: *0
(5,True):: (Int,Bool)     (Int,Bool):: *0
```

Types introduced in a `kind` declaration are classified by the kind introduced in that declaration. Types that classify no values are not new to Ωmega. All higher order types (type constructors) also classify no values, and are classified by kinds like `*0 ~>*0`.

We can take this classification process several steps further. New *kinds* introduced by the `kind` declaration are classified by *sorts*. The simplest sort is `*1`. Most kinds introduced by `kind` declarations, and the kind `*0`, are classified by the sort `*1`. The exception to this rule are higher order kinds, like `Row`, that are classified by sorts like `(*1 ~> *1)`. The sorts `*1` and `(*1 ~> *1)` are classified by `*2`, etc. For example:

```
*1:: *2
*1 ~> *1:: *2
*0:: *1
State:: *1
Nat:: *1
Locked:: State
Z:: Nat
S:: Nat ~> Nat
RCons :: forall (a:*1) . a ~> Row a ~> Row a
```

Like type constructors, the new types, `Locked`, `(S Z)`, etc. do not classify any values. We say such types are uninhabited. One may ask, "What use are uninhabited types?" The answer lies in their use as indexes to GADTs.

4. Feature: Generalized Algebraic Data Types

Generalized Algebraic Data Types (GADTs) are a generalization of *Algebraic Data Types* (ADTs). Both allow users to define inductively formed structured data. ADTs generalize other forms of structuring data such as enumerations, records, and tagged variants. For example, in Haskell we might write:

```
-- An enumeration type
data Color = Red | Blue | Green
-- A record structure
data Address = MakeAddress Number Street Town
-- A tagged Variant
data Person = Teacher [Class]  | Student Major
-- A polymorphic, recursive type
data Tree a = Fork (Tree a) (Tree a) | Node a | Tip
```

To understand the difference between GADTs and ADTs consider the declaration for `Tree` that introduces the polymorphic `Tree` type constructor. Example tree types include `(Tree Int)` and `(Tree Bool)`. In fact the type constructor `Tree` can be applied to any type whatsoever. This is called *parametric polymorphism*. Note how the constructor functions (`Fork`, `Node`) and constructor constants (`Tip`) are given polymorphic types, where the polymorphism is evident in the *parameter* (a) of `Tree` in the range of each constructor. We have underlined the range to make this clear.

```
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 this 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. The restriction is enforced because 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, where we may only place type variables).

GADTs remove this restriction. They use an alternate syntax to remove the range restriction. In Ωmega an explicit form of a `data` definition is used to define GADTs. In this form, the type being defined is given an explicit kind, and every constructor is given an explicit type. We give two examples of the alternate form below:

```
data Tree:: *0 ~> *0 where
  Fork:: Tree a -> Tree a -> Tree a
  Node:: a -> Tree a
  Tip:: Tree a

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 first is an equivalent declaration for `Tree` (that doesn't make any use of the added flexibility). The second defines new type constructor `Term` that makes use of the added flexibility. This is evident in the type of the constructor `Pair`. Note that `Term` is classified by the kind `*0 ~> *0`. This means it takes types to types.

In the alternate form, no restriction is placed on the types of the constructors 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 `*0`. Note how the range of the constructor function `Pair` is a non-polymorphic instance of `Term`.

To appreciate the power of GADTs, consider that `Term`s are a typed object-language representation, i.e. a data structure that represents terms in some object-language. The added value of using GADTs over ADTs, in this case, is that the meta-level type of the representation (`Term a`), indicates the type of the object-level term (`a`).

```
ex1 :: Term Int
ex1 = App (App (Const (+)) (Const 3)) (Const 5)

ex2 :: Term (Int,String)
ex2 = Pair ex1 (Const "z")
```

Type inference for expressions that perform a case analysis over a GADT structure is not generally possible (except in special cases, see [33] for work on how type inference might be performed), but type checking is. For example, consider:

```
eval :: Term a -> a
eval (Const x) = x
eval (App f x) = eval f (eval x)
eval (Pair x y) = (eval x,eval y)
```

The prototype declaration (`eval :: Term a -> a`) is required in Ωmega for any function that performs a case analysis over a GADT. Fortunately, functions that don't pattern match over GADTs can have Hindley-Milner types inferred for them (see [18] for how this mixture of type-checking and type-inference is done). Requiring prototypes for only some functions should be familiar to Haskell programmers since polymorphic-recursive functions already require prototypes[19].

The problem with type inference for the `eval` function occurs in the third clause: `eval (Pair x y) = (eval x,eval y)`, it seems clear that the right-hand-side of the equation requires the range of `eval` to be some instance of a pair type, i.e. a type of the form `(c,d)`. This implies that `eval` can't have the polymorphic type `(Term a -> a)`. But, given a function prototype, we can check that the type is consistent by the following argument.

Inspection shows that argument to `eval` in this clause `(Pair x y)` has type `(Term(c,d))`, so under the substitution mapping the type variable `a` to the type term `(c,d)` this clause is well-typed. Type checking functions that manipulate GADTs requires some additional work on the part of the compiler to construct and manipulate such substitutions. But this work is performed by the compiler, and the programmer need not be concerned with the details[18].

The parameter of the type constructor `Term` (defined as a GADT) plays a qualitatively different role than type parameters in ordinary ADTs. Consider the declaration for a binary tree datatype: `data Tree a = Fork (Tree a) (Tree a) | Node a | Tip`. In this declaration the type parameter `a` is used to indicate that there are sub components of `Tree`s that are of type `a`. In fact, `Tree`s are polymorphic. Any type of value can be placed in the "sub component" of type `a`. The type of the value placed in a polymorphic sub-component is reflected in the type of such a value as a parameter to the type constructor, e.g. `(Tree a)`.

Contrast this with the `a` in `(Term a)` (defined earlier in this Section). Instead, the type variable `a` is used to stand for an abstract property (the object level type of the term represented). Type variables used in this way lead to what we call *index-polymorphism* (as opposed to parametric polymorphism). Index types have been well studied[39, 43] and will play an important role in what follows.

By using new types introduced by `kind` declarations as type indexes we can express (and enforce) an amazing amount of structure on well-typed programs. Note how it is not necessary for a type index to be inhabited. A type index states some abstract property about the value being constructed, not that a sub-value of that type occurs somewhere as a component. At an intuitive level, this is the difference between a type index and a type parameter. Type indexes are introduced whenever we give a non-polymorphic type to the range of a constructor function.

5. Pattern: Indexed Datatypes

Type indexed arguments to GADTs allow us to ensure static properties of data structures. Consider sequences where the length of the sequence is encoded in its type as a type index. For example the sequence $[a_1, a_2, a_3]$ is classified by $(Seq\ a\ 3)$, and the type of the $Cons$ operator that adds an element to the front of a sequence would be $a \rightarrow Seq\ a\ n \rightarrow Seq\ a\ (n+1)$. By using the `Nat` kind as the index set we define a GADT for sequences with statically known lengths.

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

The types of functions over type indexed GADTS often witness important properties of the function. For example, a map function for sequences with type `(a -> b) -> Seq a n -> Seq b n` encodes a proof that map does not alter the length of the sequence it is applied to.

```
mapSeq ::(a -> b) -> Seq a n -> Seq b n
mapSeq f Nil = Nil
mapSeq f (Cons x xs) = Cons (f x) (mapSeq f xs)
```

Functions over index refined types which manipulate the type index often require solving constraints over the index set. For example, consider type of the append operator: $Seq\ a\ n \rightarrow Seq\ a\ m \rightarrow Seq\ a\ (n+m)$. In order to type such functions it is necessary to do arithmetic at the type level at type checking time. As daunting as this seems, such systems have been found to be extremely useful for eliminating dead code[40], and eliminating array bound checks[42]. Both of these systems employ a decision procedure for solving linear inequalities on integers which is used by the type checker.

6. Feature: Type-Functions

We can illustrate this on a much simpler scale by the Ωmega program in below:

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

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

The code introduces a new *type-function* (`plus`), and the definition of `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 checking app, generates and propagates equalities constraints, and requires solving the equations (S{plus t m} = {plus n m}) and (n = S t). Substituting the second equality into the first we get (S{plus t m} = {plus (S t) m}), and applying the definition of plus we get the identity (S{plus t m} = S{plus t m}).

7. Pattern: Relationships Between Types

In Haskell, a class constraint can be viewed as a predicate (or relation) over types. For example: (Eq a) states that the type a must have an equality function defined over that type. Multi-parameter type classes with two parameters can be viewed as binary predicates (or relations). Any parameterized data type in Haskell can also be considered as a relation on types. For example the type [Int] states that the type Int can be placed in lists. A value of type [Int] can be thought of as a constructive proof that such lists exist. Of course since any type can be placed in a list (because list is parametric-polymorphic) such "proofs" are of little practical use.

With the advent of GADTs and an extensible kind system we can define new types with interesting structure, and define GADTs whose type constructors can be used as meaningful relations. For example the kind Nat introduced in Section 3, classifies types such as Z, and (S Z), and (S (S Z)) that have the structure of the natural numbers. An interesting property of a natural number is whether it is even or odd. Consider the two GADTs:

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

data Odd :: Nat ~> *0  where
  OddS:: Even n -> Odd(S n)
```

Both Even and Odd use their Nat arguments as type indexes. Unlike parametric polymorphic types, there does not exist a well-typed value of type (Even a) for all types a. For example consider:

```
even2 :: Even (S (S Z))
even2 = EvenS(OddS EvenZ)

odd1 :: Odd (S Z)
odd1 = OddS EvenZ
```

A little thought should convince the reader that *if* a well-typed value has type (Even a), *then* the natural number a must be even. A similar result applies to values of type (Odd b). Values of type (Even a) and (Odd b) are proofs of the propositions that a is even, and b is odd. We call such types witness types, since values of such a type witness interesting properties of their type parameters.

The use of natural numbers as indexes is so common, that in Ωmega natural numbers as types can be written using syntactic sugar. In Ωmega we may write #0 for Z, and #1 for (S Z), and #2 for (S (S Z)), etc. We may also write #(1 + n) for (S n), and #(2 + n) for (S (S n)) when n is a type variable.

Of course we are not limited to unary properties over the natural numbers. Consider the binary less-than-or-equal relation.

```
data LE :: Nat ~> Nat ~> *0  where
  Base:: LE Z x
  Step:: LE x y -> LE (S x) (S y)
```

Consider a few values of type (LE a b).

```
le23 :: LE #2 #3
le23 = Step(Step Base)
le2x :: LE #2 #(2+a)
le2x = Step(Step Base)
```

Convince yourself that only true statements about the ordering between natural numbers can be reflected in the type of an LE value. The types Even, Odd, and LE are propositions, and the programs (i.e. terms such as (Step(Step Base)) are proofs.

We are not limited to expressing properties over the natural numbers. Any kind with interesting structure will do. Consider the kind Polarity

```
kind Polarity = Plus *0
              | Minus *0
              | Product Polarity Polarity
```

The types Plus and Minus of kind (*0 ~> Polarity) are used to tag ordinary types that classify values. For example, the type (Plus Int) indicates that Int is to be used in a positive manner, and the type (Minus Bool) indicates that Bool is to be used in a negative manner. In Section 13 we will use polarity to attach information about the inports and outports of programs that stream data. The Product type constructor of Polarity will be used to attach information to streams with multiple inports and outports. We will use Polarity to enforce the invariant that only streams with opposite polarities can be connected. The property that two polarities are opposite can be witnessed by the following GADT.

```
data Opposite :: Polarity ~> Polarity  ~> *0 where
  PM :: Opposite (Plus a) (Minus a)
  MP :: Opposite (Minus a)(Plus a)
  PP :: Opposite x a ->
        Opposite y b ->
        Opposite (Product x y) (Product a b)
```

The nullary constructors PM and MP witness the basic facts that Plus is opposite of Minus, and vice versa. The constructor function PP indicates that compound polarities constructed with Product are opposite if their components are pair-wise opposite. Like all witness types, there are no inhabitants of illegal properties like (Opposite (Plus a) (Plus a)). Two legal witnesses are given below for the reader's inspection.

```
PM:: Opposite (Plus a) (Minus a)

(PP PM MP):: Opposite (Product (Plus a)  (Minus b))
                      (Product (Minus a) (Plus b))
```

8. Pattern: Singleton Types

Using the kind declaration it is possible to construct new types with significant structure. For example types Z and S of kind Nat, allow the user to construct types with the structure of the natural numbers. Sometimes it is useful to compute over this structure in both the type-world and the value-world. For reasons discussed in Section 16 we want to completely separate types from values. Fortunately, we can still compute over types by building reflections of types in the value-world. The idea is to build a completely separate isomorphic copy of the type in the value world, but still retain a connection between the two isomorphic structures. This connection is maintained by indexing the value-world type with the corresponding type-world kind. This is best understood by example. Consider reflecting the kind Nat into the value-world by defining the type constructor Nat' using a data declaration.

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

Here, the value constructors of the data declaration for Nat' mirror the type constructors in the kind declaration of Nat. Note that we exploit that the name space for values and the name space for types are separate. Thus, we can use the same name `Z` for the constructor function of the type `Nat'` (in the value name space) , and the type constructor `Z` of the kind `Nat` (in the type name space). We do the same for `S` as well. This overloading is deliberate, because the structures `Nat'` and `Nat` are so closely related. We maintain the connection between the two isomorphic structures by the use of `Nat'`'s type index argument. This type index is in one-to-one correspondence with the shape of the value. Thus, the type index of `Nat'` exactly mirrors its shape. For example consider the example `two` below, and pay particular attention to the structure of the type index, and the structure of the value with that type.

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

We call such related types singleton types because there is only one element of any singleton type. For example only `S (S Z)` inhabits the type `Nat'(S (S Z))`. It is possible to define a singleton type for any first order type (of any kind). All Singleton types always have kinds of the form `I ~> *0` where `I` is the index we are reflecting into the value world. We sometimes call singleton types *representation types*. We cannot over emphasize the importance of the singleton property. Every singleton type completely characterizes the structure of its single inhabitant, and the structure of a value in a singleton type completely characterizes its type. This pattern is so useful we have considered adding *deriving* clauses for kinds. For example we might write:

```
kind Nat = Z | S Nat
     deriving Singleton Nat'
```

9. Pattern: Creating Propositions at Run-time

Propositions such as `LE` capture static properties of their index types. It is possible to construct such witnesses dynamically at run-time. For example, consider the function `comp :: Nat' a -> Nat' b -> (LE a b + LE b a)`. This type tells us that if we wish to compare two value-world natural numbers (`(Nat' a)` and `(Nat' b)`) at runtime, we must return one of two proofs (`(LE a b)` or `(LE b a)`) depending on which of the natural numbers was larger. The infix sum type `(+)` is isomorphic to the `Either` type constructor in Haskell with injection functions `(L:: a -> a+b)` and `(R:: b -> a+b)`. Because `Nat'` is a singleton type, we can essentially discover the *type* of the index `(x)` by investigating the *structure* of the value with type `(Nat' x)`.

```
comp :: Nat' a -> Nat' b -> (LE a b + LE b a)
comp Z Z     = L Base
comp Z (S x) = L Base
comp (S x) Z = R Base
comp (S x) (S y) = case comp x y of
                    R p -> R (Step p)
                    L p -> L (Step p)
```

We can put `LE` proposition types to use by storing them inside data structures to witness properties of the data structures. For example consider a dynamic sorted sequence type whose `(Nat' x)` elements are always stored in descending order.

```
data Dss:: Nat ~> *0 where
  Dnil:: Dss Z
  Dcons:: (Nat' n) -> (LE m n) -> (Dss m) -> Dss n
```

A sequence of type `(Dss n)` has largest (and first) element of size `n`. The proposition `(LE m n)` stored in every cons cell guarantees that no ascending sequences can be constructed. We can merge two sorted sequences into one larger sequence. We must dynamically compute new `LE` proofs as the new list is constructed. We use the `comp` function to do so.

```
merge :: Dss n -> Dss m -> (Dss n + Dss m)
merge Dnil ys = R ys
merge xs Dnil = L xs
merge (a@(Dcons x px xs)) (b@(Dcons y py ys)) =
  case comp x y of
    L p -> case merge a ys of
             L ws -> R(Dcons y p ws)
             R ws -> R(Dcons y py ws)
    R p -> case merge b xs of
             L ws -> L(Dcons x p ws)
             R ws -> L(Dcons x px ws)
```

10. Example: Sorting

Constructing a sorting function from a merge function is a small step. The function `msort` will produce a sorted sequence from an ordinary list of `(Nat' t)` inputs. Unfortunately a list such as `[Z, S Z]` is ill-typed since `Z::Nat' #0`, and `(S Z)::Nat' #1` have different types. We must hide the type index to `Nat'` so that all the elements in the list have the same type. Fortunately we can use GADTs to encode existentially quantified types.

```
data Covert:: (Nat ~> *0) ~> *0 where
   Hide::  (t x) -> Covert t
```

The type variable `x` in the declaration for `Covert` is existentially quantified since it does not appear in the range of the constructor `Hide`. In Haskell we would accomplish the same thing by:

```
data Covert t = forall x . Hide (t x)
```

To construct a list of covert natural numbers, we apply the `Hide` constructor to each one. In Ωmega, the hash syntactic sugar is overloaded to work on both `Nat` at the type level, and `Nat'` at the value level.

```
inputList :: [Covert Nat']
inputList = [Hide #1,Hide #2,Hide #4, Hide #3]
```

To construct a sorted sequence, split a list into two parts, sort each of the parts, and then merge the results.

```
split [] pair = pair
split [x] (xs,ys) = (x:xs,ys)
split (x:y:zs) (xs,ys) = split zs (x:xs,y:ys)

msort :: [Covert Nat'] -> Covert Dss
msort [] = Hide Dnil
msort [Hide x] = Hide(Dcons x Base Dnil)
msort xs = let (y,z) = split xs ([],[])
               (Hide ys) = msort y
               (Hide zs) = msort z
           in case merge ys zs of
               L z -> Hide z
               R z -> Hide z
```

Note how the type of `msort` guarantees that the result type will be sorted. We can test our function and observe its results by evaluating `(msort inputList)`.

```
Hide (Dcons #4 (Step (Step (Step Base)))
     (Dcons #3 (Step (Step Base))
     (Dcons #2 (Step Base)
     (Dcons #1 Base Dnil))))
```

11. Feature: Static Propositions

There is one glaring problem in the process illustrated above. We must construct and store the LE proofs in the sorted list as we go, and these proofs can be large. We can remove this obligation by requiring that these proofs can be constructed statically at compile-time. The proofs do not need to be passed around, constructed, or manipulated at run-time. In Ωmega static propositions are much like class constraints in Haskell. A static proposition appears as constraint in the type of a term. We illustrate this with the Static Sorted Sequence data type.

```
data Sss:: Nat ~> *0 where
  Snil:: Sss #0
  Scons:: LE a b => Nat' b -> Sss a -> Sss b
```

Note that the type of Scons is a constrained type. Whenever the Scons constructor function is used, a static obligation is generated. This obligation requires a static proof that the largest element (a) in the sub list is less-than-or-equal to the element being added (b). Such static propositions are treated exactly like class constraints in Haskell. They are propagated and solved by the type checker. For example. `\ x y z -> Scons x (Scons y z)` is given the type:

```
forall a b c . (LE a b,LE b c) =>
   Nat' c -> Nat' b -> Sss a -> Sss c
```

Any 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, and has almost exactly the same semantics. It introduces a new type constructor and its associated (value) constructor functions, but in addition, it also informs the compiler that the new type constructor can be used as a static level proposition that can be used as a constraint. For example, by changing data to prop we declare:

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

The type LE is introduced (exactly as before) with constructor functions Base and Step. These construct ordinary values. But the type LE can now also be used as static constraint. The compiler uses the type of the constructor functions for LE to build the following constraint solving rules, that reduce a constraint to a set of simpler constraints.

```
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. These rules follow directly from the types of the constructor functions that are the only legal mechanism for constructing values of type LE.

The user can introduce additional rules describing how to discharge constraints by writing normal value-world functions that manipulate values of the constraint type as ordinary data. This allows programmers to introduce and prove theorems about static level constraints.

These "theorem" functions describe how new propositional facts can be introduced by computing over older facts. 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 CODE
trans Base (Step z) = Base
trans (Step x) (Step y) = (Step(trans x y))
```

Because `(Base z)::LE (S w) (S x)` and `Base::LE Z y`, the second clause is unreachable. Such a call would require the type variable b in the prototype declaration to be simultaneously `(S w)` and Z, a contradiction. Thus a call to `(trans (Step z) Base)` can never be well typed. Since this function is total, the term describing the definition of the function *is a proof*, and its type *is a proposition*. The type of the function becomes a new rule that can be used to satisfy propositional obligations. The type of trans is now added to the theorem prover used by the type checker to discharge obligations.

```
trans: LE a b --> {}
   if (Exists c). LE a c, LE c b
```

This rule say that we can satisfy `(LE a b)` if we can find a concrete c such that (both) `(LE a c)` and `(LE c b)`.

12. Pattern: Static Versions of Dynamic Types

The type `(LE' n m)` is a static version of the LE proposition type from Section 7.

```
data LE' :: Nat ~> Nat ~> *0 where
  LE:: (LE m n) => LE' m n
```

Unlike its dynamic counterpart, LE' is a one-point type. But it is a polymorphic type, and its single value can have many types. Contrast this with values of type LE that can have arbitrarily large size, whereas LE' values always have unit size. We can now redo the function comp, but make it return a static LE' type instead of a LE proposition.

```
compare :: Nat' a -> Nat' b -> (LE' a b + LE' b a)
compare Z Z     = L LE
compare Z (S x) = L LE
compare (S x) Z =  R LE
compare (a@(S x)) (b@(S y)) =
   case compare x y of
      R (p@LE) -> R LE  -- HERE
      L LE -> L LE
```

The fourth clause of compare requires some explanation. Contrast this with the dynamnic clause in the function comp. It appears that the pattern and the left-hand-side of each arm of the case is identical, making the case analysis unnecessary. But in reality the two LE witnesses (on the left and right hand sides of the match) have different types.

Pattern matching against a constructor function (like Scons or LE) causes the static constraints associated with that constructors type to be available as assumptions in the scope of the match. For example in the scope labeled by the comment HERE in the definition of compare we have the following types for the variables in scope.

```
a :: Nat' #(1+_c)
b :: Nat' #(1+_d)
x :: Nat' _c
y :: Nat' _d
compare x y :: ((LE' _c _d)+(LE' _d _c))
p :: LE' _d _c
```

Because we are in the scope of p we can assume `(LE _d _c)`. In this context we return `(R LE)` that we expect to have the type `((LE' #(1+_c) #(1+_d))+(LE' #(1+_d) #(1+_c)))` Striping off the sum injection R, the term LE can have the type `(LE' #(1+_d) #(1+_c))` if in the current context we can discharge the obligation `(LE #(1+_d) #(1+_c))`. Using the rule Step this is reduced to `(LE _d _c)`, but this is exactly the assumption introduced by the pattern p.

The function merge2 is analogous to merge except it employs a static LE' proof, and builds static sorted sequences Sss.

```
merge2 :: Sss n -> Sss m -> (Sss n + Sss m)
merge2 Snil ys = R ys
merge2 xs Snil = L xs
merge2 (a@(Scons x xs)) (b@(Scons y ys)) =
  case compare x y of
    L LE -> case merge2 a ys of
             L ws -> R(Scons y ws)
             R ws -> R(Scons y ws)
    R LE -> case merge2 b xs of
             L ws -> L(Scons x ws)
             R ws -> L(Scons x ws)
```

The function `merge2` is a vast improvement over `merge` in that it constructs only 1 point `LE'` proofs, and does all the reasoning about proofs at compile-time. The function `msort2` is defined almost exactly the same as `msort` except it builds static sorted sequences.

```
msort2 :: [Covert Nat'] -> Covert Sss
msort2 [] = Hide Snil
msort2 [Hide x] = Hide(Scons x Snil)
msort2 xs =
  let (y,z) = split xs ([],[])
      (Hide ys) = msort2 y
      (Hide zs) = msort2 z
  in case merge2 ys zs of
      L z -> Hide z
      R z -> Hide z
```

The result of sorting, as static sorted sequence contains no less-than-or-equal proofs.

```
Hide (Scons #4 (Scons #3 (Scons #2 (Scons #1 Snil))))
```

Contrast this with the output of `msort` at the end of Section 10.

13. Example: Infopipes

The *Infopipes Project* defines a set of generic tools for building data-streaming applications. An Infopipeline is more than a stream transducer capable of transforming data, it can indicate non-functional aspects of the stream, including properties like the presence of buffers, and whether the data is pushed or pulled through the pipe. A good way to understand these aspects is to use an analogy with real plumbing, where the water is replaced with data, and data can be pumped, drained, or stored in tanks. Further information and details on Infopipes can be obtained at the infopipe websites[15].

An example Infopipeline is a video-streaming application, where data from multiple sources (a camera and a microphone), is buffered, encoded and compressed, transmitted over a TCP connection, and then decoded and displayed at a remote location. Interesting characteristics of Infopipes include:

- The ends of an Infopipe can be either positive or negative, pushing or pulling data, or waiting for data to be pulled-from or pushed-into adjacent connections in the Infopipeline. In the Infopipe lexicon this is called *polarity*.
- Infopipes can have varying numbers of *inports* and *outports*.
- Infopipes can split and merge in arbitrary ways.
- Infopipe components are *polymorphic* over the kind of data streamed.
- Every complete Infopipe starts at one or more *source*s and ends at one or more *sink*s. Data may only stream through Infopipes that are *grounded* at both ends with sources and sinks, but *open* pipes, that are not grounded, are useful components for building larger pipes.

Infopipes are meant to be high-level abstractions for describing generic pipelines that can be instantiated to build efficient and maintainable systems. There are two implementations of Infopipes. The first is built from object-oriented components in a version of Smalltalk, and the second is built in Haskell. Infopipes are dynamically configured at start-up time, and once configured, are set into action. The Infopipes project has been studying a domain-specific language (DSL) for static configuration of Infopipes. The type system of the DSL should ensure a number of properties statically:

- Positive ports are connected only to negative ports, and vice-versa.
- Outports of a pipe streaming objects of type `a` should only be connected to inports of pipes streaming objects of type `a`.
- The number, type, and polarity of inports and outports should be evident in the type of an Infopipe.
- Only pipes that are completely grounded should be set into action. There should be no loose ends.

We have used Ωmega's type system to design an interface that meets all these requirements. The type system introduces one `data` type (`Pipe`) to denote the type of an Infopipe, two new kinds (`Polarity` and `Port`), one proposition type (`Opposite`).

```
Pipe:: Port ~> Port ~> *0

kind Port = Source | Sink | Open Polarity

kind Polarity = Plus *0
              | Minus *0
              | Product Polarity Polarity
```

The type `Pipe` is indexed by two `Port`s. These are intended to indicate the input and output ports of the pipe. Details (and a complete definition) are given later. A `Port` is either a grounded `Source`, a grounded `Sink`, or an `Open` port. The kind declaration `Polarity` (repeated here from Section 7) is used to tag an open port with its polarity. For example, `(Plus Int)` denotes an open positive port streaming integers, while `(Minus Bool)` denotes an open negative port streaming booleans.

Thus `(Pipe Source (Open (Plus Int)))` is the type of a pipe with no inputs (it's a source), and one (ungrounded) positive integer output. The type constructor `Product` is used to construct Infopipes with multiple inputs or outputs. For example: `(Pipe (Open (Product (Plus Int) (Minus Bool))) Sink)` is a pipe with two inputs. The first is a positive port pulling integers, and the second is a negative port accepting booleans. It has no output ports (it's a sink).

Connecting ports requires that the two ends of the connection are open and are of opposite polarity. For this we use the `Opposite` proposition data type introduced in Section 7, but we redeclare it here as a static proposition (we use the keyword `prop` instead of `data`) because we intend to use it statically.

```
prop Opposite :: Polarity ~> Polarity  ~> *0 where
  PM :: Opposite (Plus a) (Minus a)
  MP :: Opposite (Minus a)(Plus a)
  PP :: Opposite x a ->
        Opposite y b ->
        Opposite (Product x y) (Product a b)
```

This introduces the following rules that can be used by the type checker to discharge `Opposite` obligations

```
PM: Opposite (Plus 'a) (Minus 'a) --> { }
```

```
MP: Opposite (Minus 'a) (Plus 'a) --> { }

PP: Opposite (Product 'a 'b) (Product 'c 'd)
    --> {Opposite 'a 'c, Opposite 'b 'd}
```

The Pipe datatype declares the operators in the DSL for configuring Infopipes.

```
data Pipe  :: Port ~> Port ~> *0 where
  PosSrc   :: Pipe Source (Open (Plus a))
  NegSrc  :: Pipe Source (Open (Minus a))
  PosSink  :: Pipe (Open (Plus a)) Sink
  NegSink :: Pipe (Open (Minus a)) Sink

  Buffer   :: Pipe (Open (Minus a)) (Open (Minus a))
  Pump     :: Pipe (Open (Plus a)) (Open (Plus a))

  -- stream transducers
  Filter   :: (a -> Bool) ->
              Pipe x (Open (n a)) ->
              Pipe x (Open (n a))
  Map      :: (a -> b) ->
              Pipe x (Open (n a)) ->
              Pipe x (Open (n b))

  -- Connecting two pipes
  C        :: Opposite (m a) (n a) =>
              Pipe x (Open (m a)) ->
              Pipe (Open (n a)) y -> Pipe x y

  -- Pipes with multiple inputs and outputs
  Dup      :: Pipe x (Open (y b)) ->
              Pipe x (Open (Product (y b) (y b)))
  Merge    :: ([a] -> [b] -> [c]) ->
              Pipe x (Open (Product (y a) (y b))) ->
              Pipe x (Open (y c))
  Bundle   :: Pipe (Open x) (Open y) ->
              Pipe (Open a) (Open b) ->
              Pipe (Open (Product x a)) (Open (Product y b))

  -- Grounding one input or output of a multi-pipe
  Source1  :: Opposite (n x) (m x) =>
              Pipe Source (Open (n x)) ->
              Pipe (Open (Product (m x) b)) y ->
              Pipe (Open b) y
  Source2  :: Opposite (n x) (m x) =>
              Pipe Source (Open (n x)) ->
              Pipe (Open (Product b (m x))) y ->
              Pipe (Open b) y
  Sink1    :: Opposite (n x) (m x) =>
              Pipe y (Open (Product (m x) b)) ->
              Pipe (Open (n x)) Sink ->
              Pipe y (Open b)
  Sink2    :: Opposite (n x) (m x) =>
              Pipe y (Open (Product b (m x))) ->
              Pipe (Open (n x)) Sink ->
              Pipe y (Open b)

-- Begin processing data in a grounded pipe
start :: Pipe Source Sink -> IO()
```

There are operators for creating sources (PosSrc and NegSrc), and sinks (NegSink and PosSink), Buffers, Pumps, Maps and Filters. The C (*C*onnection or *C*omposition) operator has an interesting type:

```
C:: Opposite (m a) (n a) =>
    Pipe x (Open (m a)) ->
    Pipe (Open (n a)) y ->
    Pipe x y
```

We can connect an positive source to a negative sink by using the function C: (C PosSrc NegSink) or by using C as an infix operator (by surrounding it with back quotes) we can write (PosSrc ‘C‘ NegSink).

If we can statically show that (m a) and (n a) are opposites, then we are allowed to compose a (Pipe x (Open (m a))) with a (Pipe (Open (n a)) y) to get a (Pipe x y).

Dup, Merge, and Bundle, split, consume, and create pipes with multiple inputs and outputs. Source1, Source2, Sink1, and Sink2 ground one part of a multiple port pipe simplifying it to a single port pipe.

Pipes are constructed by combining the operators. Illegal combinations are detected and disallowed statically by the type system. The static Opposite constraints, arising from the use of the constructors C, Source1, Sink1, etc., are collected and discharged silently by the type system as long as they are used correctly.

```
p1 = NegSrc ‘C‘ Pump ‘f‘ Dup ‘f‘ merge ‘C‘ NegSink
  where alternate (x:xs) ys = x : alternate ys xs
        merge = Merge alternate
        f x y = y x
```

But, if used incorrectly, an error message is returned. For example connecting an positive source to an positive sink, raises the following message while discharging the accumulated obligations:

```
prompt> PosSrc ‘C‘ PosSink
The proposition: (Opposite (Plus a) (Plus a))
can never be solved.
```

Ill-formed pipelines with respect to polarity can never be constructed.

This example merely shows that we can use the extensions to Haskell we have built into Ωmega to define a well-typed interface to the Infopipes paradigm. It does not address how we might implement an Infopipe toolbox. This requires building actual implementations of infopipe components, and building wrapper functions to give actual hardware devices the types necessary.

14. Example: Modular Arithmetic

In this example we show how to use the Curry-Howard isomorphism to define the modular integers. The value of the modulus will be captured by a type index of the new type. This allows us to use a single data declaration to describe the integers with any modulus, and the types with different moduli to be statically different, yet we need only write a single definition for each of the modular arithmetic definitions. Consider:

```
data Mod :: Nat ~> *0 where
  Mod :: Int -> Nat' n -> Mod n

x:: Mod #3
x = Mod 6 #3

y:: Mod #2
y = Mod 6 #2
```

A value of type (Mod n) stores a normal Int (its value) and a (Nat' n) (its modulus). Note how the variables x and y have the same value (6) but different types. Thus we can use the type system to prevent programmers from adding numbers with different moduli. Yet, we need write each operation only once.

```
natToInt :: Nat' n -> Int  -- A conversion function
natToInt Z = 0
natToInt (S n) = 1 + natToInt n

normalize :: Mod n -> Mod n
normalize (Mod val n) = Mod (mod val (natToInt n)) n
```

```
plusM :: Mod n -> Mod n -> Mod n
plusM (Mod a n) (Mod b _) = normalize (Mod (a+b) n)

minusM :: Mod n -> Mod n -> Mod n
minusM (Mod a n) (Mod b _) = normalize (Mod (a-b) n)

timesM :: Mod n -> Mod n -> Mod n
timesM (Mod a n) (Mod b _) = normalize (Mod (a*b) n)
```

Some moduli, but not all, support a multiplicative inverse function such that $x * (inv\ \ y) = 1$. The only moduli that support the inverse function are moduli that are prime numbers. We can define this function by using Euclid's extended greatest-common-denominator function. But more importantly we can give it a type that *prevents its use if the modulus is not prime*!

```
invM :: forall n . (Prime n) => Mod n -> Mod n
invM (Mod a n) = Mod (mod b n') n
  where n' = natToInt n
        (_,_,b) = gcd n' a

gcd p q | q > p = gcd q p
gcd p q | q==0 = (p,1,0)
gcd p q = (g,y1,x1 - (p `div` q)*y1)
  where (g,x1,y1) = gcd q (p `mod` q)
```

Here we have used a static proposition `Prime` that must be established at compile-time, as an obligation to the type of the function `invM`. Writing a completely general `Prime` proposition is difficult (but not impossible, but still beyond the scope of this short note). Fortunately if we know in advance the range of prime numbers we wish to use we can write a `Prime` proposition that simply enumerates the prime numbers of interest.

```
-- Primes:
prop Prime :: Nat ~> *0 where
  Two   :: Prime #2
  Three :: Prime #3
  Five  :: Prime #5
  Seven :: Prime #7
```

We tend to think of this as a programming pattern similar in structure to the use of the class system in Haskell. When we don't have a general prescription for defining operations over all types, we simply enumerate the types of interest by writing a class instance for each of them.

15. Why is Ωmega Strict?

In a strict language, functions completely evaluate their arguments before execution of the body of the function begins. This turns out to have a benefical effect on Ωmega. Implementing Ωmega as a strict language was originally a pragmatic concern – we knew how to implement strict languages. We were more concerned with trying out the ideas embedded in our new features than being strictly faithful to the laziness of Haskell. But, as our experience with Ωmega accumulated, the use of GADTs as proof objects became more important than we had first imagined.

It is important for proof objects in Ωmega to attest to true things. That is, regarded as a logic, Ωmega should be sound. This requires that the only inhabitant of a proposition be one of the well formed objects built from its constructors. This requires that bottom not be an inhabitant. Strictness is part of the means to that end (though not the only means).

Our goal is to eventually prove that Ωmega is sound logic. We have not done that yet. We expect the soundness of Ωmega will follow from the following properties:

- **Subject Reduction.** The type of every term in the language should be invariant under evaluation.
- **Termination.** Non-termination will be tracked. All "theorem" functions, whose types are used as propositions, must be total and terminating. Note that dynamic witness objects are not a problem. In a strict language if a program reaches a program point with a value that has a dynamic witness type, that witness value cannot be bottom.
- **Consistency.** There will be types in the terminating fragment of the language that do not have any inhabitants.

The possible mechanism for tracking termination in Ωmega are currently being discussed. Comments on this issue are welcomed.

16. Separating Types from Values

The design of Ωmega separates types from values by design. Our reason for completely separating types from values stems from two desires. First, we wish to maintain the programming style that functional programmers are accustomed to, and this includes a strong distinction between types an values. Our experience with Cayenne has convinced us that removing this distinction drastically changes the way we program whenever we use any of the dependent-typing features of Cayenne. Second, we hope to build efficient implementations, and this requires performing as much computation at compile-time as is possible. Computation at the type level happens only at compile time. The goal is to allow users to specify how to distribute the computation between compile time and run time.

The most important consequence of not separating values from types is the loss of the opportunity to use an erasure semantics, and the ensuing increase in the amount of explicit type annotation required. An erasure semantics is only possible when there exists a strict phase distinction between values and types. In such a setting, type abstractions and type applications in polymorphic programs can be made implicit, and their effect can be "erased" at runtime.

This can be best illustrated by an example. Consider the `Seq` datatype in two different contexts. In the first context, where values and types are the same, we borrow some notation from Dybjer[11] (but cast it in the style of Ωmega for consistency). We define refinement of the list type, where the static length of the list is a type index to the constructor function `Seq`.

```
data Seq :: (a::Set) ~> (n:: Nat) ~> Set  where
  Nil:: (a :: Set) -> Seq a Z
  Cons:: (a::Set) -> (n::Nat) ->
         a -> Seq a n -> Seq a (S n)
```

We use `Set` to be the "type of types" (sort of like `*0`), and the constructors must be explicitly applied to their type parameters (`a` and `n`), like any of their other arguments. We have to write these arguments every time we use one of the constructors `Nil` or `Cons`.

In the second context, types and values are separated, the type parameters are implicit, and while they are still there, the type checking mechanism can automatically insert them, and the dynamic semantics can safely ignore them.

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

This helps keep the typing annotations to a minimum, one of our goals. Several systems[21, 1] allow the user to indicate that some type parameters are strictly constant, and thus their annotations can be placed by an inference mechanism. If the user's indication is incorrect, then the inference mechanism can fail. With a strict separation between types and values, the type abstractions and type application annotations can always we inferred. Of course we pay for this with a loss in expressivity, but our experience indicates that much (if not all) of this loss can be recovered by the use singleton types. Our experience in writing scores of Ωmega programs leads us to believe that is the case.

In addition, singleton types allow users to *choose* when they must pass types at run-time, and they allow users to slip between static and dynamic type-checking at will, in a single framework, and even in a single program.

It might be desirable to have different termination policies at the type and value level. Most current systems enforce that type checking terminates. In the current Ωmega system, it is almost certainly possible to define theorems that cause the constraint discharging mechanism to loop, and thus fail to terminate. But is this really dangerous? Shouldn't the goal be to prevent the deployment of programs that go wrong at run-time? It might be advantageous to use heuristic approaches for solving compile- time computational questions even if they don't always terminate.

17. Related Work.

The design space of languages and systems that exploit the Curry-Howard isomorphism is large. We have chosen a small corner of the design space that makes a strict phase distinction between types and values. We also have a strong desire to minimize type annotations and other administrative work. A comparison of related work is best performed in the light of this design decision.

The programming language Cayenne[2, 1] is closely related to Ωmega, but chooses not to separate types and values. This leads to a style of programming where types are passed as values. This changes the style in which we program, sometime drastically. First and foremost, our experience about which programs will type-check is often wrong. It is often hard to determine when type checking will succeed and when it will fail.

In contrast, Ωmega allows users several static alternatives to passing types as values. Type indexes can be used to index a data structure by its properties, rather than pass a separate (type-like) value to describe its properties. In addition Cayenne has nothing like Ωmega's static constraints.

Much of the power of Ωmega stems from the power of the GADT mechanism. Its most important feature is that the range of a value constructor for a type constructor *T* may be an arbitrary instance of the type constructor *T*. Several other mechanisms also support this feature, but choose a point in the design space where types and values are indistinguishable. They include Inductive Families[8, 11], theorem provers (Coq[37], Isabelle[26]), logical frameworks (Twelf[28], LEGO[20]), proof assistants (ALF[23], Agda[7]), and dependently typed languages (Epigram[21], RSP[35]).

Several systems choose a point in the design space closer to ours, where the distinction between types and values is preserved, but without the static propositions of Ωmega they are less expressive. We owe much to these works for inspiration, examples, and implementation techniques. These include Guarded Recursive Datatype Constructors[41], First-class phantom types[6], Wobbly types[18], and Silly Type Families[3]. In these systems, type indexes are restricted to types classified by `*0`, because the systems have no way of introducing new kinds. We consider the introduction of new kinds as an important contribution of our work.

Aside from the introduction of new kinds and static propositions, there are mostly minor syntactic differences between Ωmega and Guarded Recursive Datatype Constructors(GRDC) [41]. We believe that an important syntactic difference between Ωmega and GRDC is that in GRDC prototype information for type checking is attached to pattern matching forms like `case`. In Ωmega type checking information is attached to function declarations, and is propagated by the type checker inward to `case` expressions and other pattern binding mechanisms. We believe this minimizes the burden on the programmer.

The work of Hinze and Cheney[6] is the inspiration for our implementation. Ωmega started life as an implementation of the language in their paper. It quickly grew a richer syntax, rank-N polymorphism[17], extensible kinds, static constraints, and type functions.

Wobbly Types[18] describes Simon Peyton Jones and colleagues' attempt to add GADTs to the Glasgow Haskell Compiler. It focuses on minimizing user type annotations, and develops an alternative to using equality qualified types in the type checking process. It replaces the set of equalities inside the type checker with an explicit substitution. Discussions with Simon helped increase the robustness of the Ωmega type checker.

Silly Type Families[3] is an old (1994) unpublished paper by Lennart Augustsson and Kent Petersson. The idea of GADTs is completely evident, including several interesting examples. Ironically, in the conclusion, the authors deprecate the usefulness of GADTs because they did not know how to construct GADT values algorithmically. As we demonstrate, these obstacles no longer hold. Their interesting paper is way before its time, and is now available on the author's web site by permission of Lennart.

The work on Refinement Types[43, 9] stands alone in separating types from values and in supporting indexes of kinds other than `*0`. Here the set of indexed types is usually viewed as fixed by the compiler. And each one is accompanied by a decision procedure. Ωmega can be viewed as a next logical step in this direction, allowing users to define their own indexes, and their own functions over them.

Work on typing GADTS includes Vincent Simonet and François Pottier's paper[33] on type inference (rather than type checking) for GADTs. And Martin Sulzmann's work[34, 36] on translating GADTs into existential types, and using type constraints to do type inference.

Our path to GADTs started in the work on equality types. This work was based on the idea of using Leibniz equality to build an explicit witness of type equality. In Ωmega we would write

```
data Eq a b = Witness (forall f. f a -> f b)
```

The logical intuition behind this definition is that two types are equal if, and only if, they are interchangeable in any context (the arbitrary type constructor `f`). Note how this relies heavily on the use of higher rank polymorphism. The author first encountered the germ of this idea in 2000[38]. It was well developed two years later in 2002[4, 14].

By judicious use of equality types, one can code up any GADT like structure. Consider the `Term` GADT redone this way.

```
data Term a
  = Const a
  | exists x y.Pair (Eq a (x,y))(Term x)(Term y)
  | exists b . App (Term(b -> a)) (Term b)

pair :: Term a -> Term b -> Term (a,b)
pair = Pair (Witness id)
```

Programming with `Eq` witnesses requires building explicit casting functions $C[\mathtt{a}] \rightarrow C[\mathtt{b}]$ for different contexts type C. This is both tedious and error prone. Programming with witnesses also has some problems for which no solution is known[1]. The thesis by Emir Pasalic[24] illustrates this on many examples. It also illustrates how important it is for the compiler to maintain the equality constraints.

[1] I.e. given a witness with type `(Eq (a,b) (c,d))` it is not known how to construct another witness with type `(Eq a c)` or `(Eq b d)`. This should be possible since it is a straightforward consequence of congruence.

The use of kinds to classify types has a long history[5, 16, 22]. Adding extensible kinds (and higher classifications) to a practical programming language like Ωmega is the natural next step. The notion of an infinite hierarchy of values, types, kinds, sorts, etc. has been used in constructive type theories for a variety of reasons. Harper and Pollack[12] call this hierarchy *Universes* and enumerate the reasons they have been used as follows:

1. Philosophical – predicative v.s. impredicative type theories.
2. Theoretical – limitations on the closure properties of type theories.
3. Practical – To achieve some advantages of a type of all types without sacrificing consistency.

They describe a system with universe polymorphism, allowing users to introduce a single new structure, and then use it at any level of the hierarchy, as long as a consistent assignment of each term to some level of the hierarchy can be found (or inferred).

If such a system could be incorporated into Ωmega, then the distinction between `data` and `kind` declarations could be eliminated and inference could be used to assign a level to each term.

Finally, Duggan makes use of kinds in his work on dynamic typing[10] in a manner reminiscent of our work, but the introduction of new kinds is tied to the introduction of types.

18. Conclusion

In logic the logical language is typically divided between the logical part, which includes the quantifiers and connectives, and the extra-logical (often called non-logical) part, which is specific to the domain of discourse and includes constants, function symbols, and predicate symbols.

Curry-Howard, for traditionally kinded systems gives you all of the logical sentences, but these don't express things of direct interest to the programmer. What's missing is all the extra-logical predicates and terms. Of course, one can use the Calculus of Constructions trick, and find that for higher-order logic the models are rich enough that they contain isomorphic copies of every structure in ordinary mathematics, but one is still using a logical language with an empty extra-logical part. The problem with using the Curry-Howard isomorphism as a practical tool in a programming language is not the weakness of the logic, it is the lack of structures (that relate directly to the program) with which to say interesting things.

The three extensions we propose allow the user to define interesting extra-logical structures. In particular, our propositions are predicate symbols, and our type functions like `plus` on kind `Nat` are function symbols in the logic. Extended kinds, and the ability to use these kinds as type indexes to types of kind `*0`, provide the power to have a non-trivial extra-logical language that directly relates to the program written by the user/developer.

GADTs and an extensible kind system provide a natural way for programmers to extend the logical language of the type system with concepts relevant to the program being developed. The result is that the programmer obtains the ability of Logical Frameworks[13, 27] to enrich the logical language with extra-logical features without sacrificing the look and feel of programming.

Singleton types, and other indexed types bridge the gap created by the phase distinction. They allow the extra-logical language, introduced by the programmer using GADTs and kinds, to be directly related to values. And more importantly, they allow the programmer to manipulate these values, while safely maintaining their connection to the logical world.

Our primary interest in using GADTs, kinds, and static propositions, is using these new features effectively [25, 29, 31, 32]. In particular, what other features (such as monads, rank-N polymorphism) magnify their effect, and what programming patterns (witness objects, singleton types) can be used to solve recurrent problems? Will these techniques lead to more reliable and trustworthy programs? We hope so, but only if users find them in their programming language of choice.

19. Status

Ωmega has been implemented. It can be downloaded by following the links from the author's home page: `www.cs.pdx.edu/~sheard`. All the features illustrated in the paper are available in the current implementation.

Much work remains, especially for the `prop` construct. The kinds of properties we can express in Ωmega are limited by what we can express as propositions, and how effectively we can discharge propositions. The structure and organization of the theorem prover inside the type checker that discharges propositions is currently the focus of much of this work. For example, handling overlapping rules, permutative rules, and providing strategies on how to apply rules will be needed. Currently the theorem prover consists of a non-backtracking, backing-chaining, inference engine. This has been sufficient to demonstrate the concept, but will certainly need to be strengthened and generalized.

The power of Ωmega stems from its unique combination of features.

- New kinds, and witness objects provide a direct link between the program and its properties. These first class objects provide the extra-logical vocabulary to describe the properties of programs. Each has semantic meaning within the programming language independent of its role as a logical entity. The connection between program and property is not clouded by some imprecise encoding. They provide semantic links that can't be forged.
- Separate values from types to maintain a familiar functional programming style. Singleton types and other indexed types bridge the gap created by the phase distinction. They allow the extra-logical language, introduced by the programmer using GADTs and kinds, to be directly related to values. And more importantly, they allow the programmer to manipulate these values, while safely maintaining their connection to the logical world.
- Management of the constraints is performed *inside* the language using the well understood mechanism of constrained types. Constraint generation, propagation, and discharging are handled automatically by the programming language type system. They cannot be lost, forgotten or mislaid.
- Partitioning of the constraint management into static and dynamic parts, allows the user to choose when constraints can be discharged. If a constraint cannot be discharged at compile-time, the framework supplies a sound mechanism to discharge it dynamically at run-time by a simple pattern match against a value of a dynamic witness type. This allows the user to write a program that effortlessly slides between static and dynamic checking.

20. Acknowledgements

The author would like to thank Jim Hook for many conversations about the logical structure of Ωmega. In particular, the discussion in Section 18 stems from Jim's deep insights. He would also like to thank Emerson Murphy-Brown, Tom Harke, and Nathan Linger for comments on drafts of the paper.

References

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

[2] Lennart Augustsson. Equality proofs in cayenne, July 11 2000.

[3] Lennart Augustsson and Kent Petersson. Silly type families. Available from: http://www.cs.pdx.edu/~sheard/papers/silly.pdf, 1994.

[4] Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 157–166. ACM Press, New York, September 2002. Also appears in ACM SIGPLAN Notices 37/9.

[5] H. P. Barendregt. Lambda calculi with types. In D. M. Gabbay, Samson Abramsky, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, Oxford, 1992.

[6] James Cheney and Ralf Hinze. First-class phantom types. Technical Report TR2003-1901, Cornell University, 2003. Also available from: http://www.informatik.uni-bonn.de/~ralf/publications/Phantom.pdf.

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

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

[9] 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.

[10] Dominic Duggan. Dynamic typing for distributed programming in polymorphic languages. *ACM Transactions on Programming Languages and Systems*, 21(1):11–45, January 1999.

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

[12] R. Harper and R. Pollack. Type checking with universes. *Theoretical Computer Science*, 89(1):107–136, October 1991.

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

[14] 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.

[15] Infopipe web sites:. http://www.cs.pdx.edu/~walpole/infopipes.html, and http://woodworm.cs.uml.edu/~rprice/ep/koster/.

[16] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, June 1993.

[17] 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/.

[18] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. http://research.microsoft.com/Users/simonpj/, 2004.

[19] A. J. Kfoury and Said Jahama. Type reconstruction in the presence of polymorphic recursion and recursive types. Technical report, March 21 2000.

[20] 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.

[21] 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 .

[22] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):528–569, May 1999.

[23] Bengt Nordstrom. The ALF proof editor, March 20 1996.

[24] Emir Pasalic. *The Role of Type Equality in Meta-programming*. PhD thesis, OGI School of Science & Engineering at OHSU, October 2004. Available from: http://www.cs.rice.edu/~pasalic/thesis/body.pdf.

[25] 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.

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

[27] 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.

[28] 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.

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

[30] Tim Sheard, James Hook, and Nathan Linger. GADTs + extensible kind system = dependent programming. Technical report, Portland State University, 2005. http://www.cs.pdx.edu/~sheard.

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

[32] 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/.

[33] Vincent Simonet and François Pottier. Constraint-based type inference for guarded algebraic data types. Available from: http://cristal.inria.fr/~simonet/publis/index.en.html .

[34] Peter J. Stuckey and Martin Sulzmann. Type inference for guarded recursive data types, February 2005. Available from: http://www.comp.nus.edu.sg/~sulzmann/ .

[35] 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/.

[36] Martin Sulzmann and Meng Wang. A systematic translation of guarded recursive data types to existential types, February 2005. Available from: http://www.comp.nus.edu.sg/~sulzmann/ .

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

[38] Stephanie Weirich. Type-safe cast: (functional pearl). *ACM SIGPLAN Notices*, 35(9):58–67, September 2000.

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

[40] Hongwei Xi. Dead code elimination through dependent types. *Lecture Notes in Computer Science*, 1551:228–242, 1999.

[41] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *POPL: 30th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2003.

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

[43] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, ACM SIGPLAN Notices, pages 214–227, New York, NY, USA, 1999. ACM Press.