GADTs + Extensible Kinds = Dependent Programming

A Programming Pearl for the 21^{st} Century

Tim Sheard James Hook Nathan Linger

Portland State University

{sheard,hook,rlinger}@cs.pdx.edu

Abstract

We hope this paper stimulates a discussion about the future of functional programming languages. To start that discussion we make a concrete proposal. We explore a new point in the design space of programming languages. The basic idea is an old one: use types to express properties of programs, and then use the type checker of the programming language to ensure that only those programs with the properties desired can be expressed. But, unlike dependently typed systems, we propose maintaining the complete separation between types and values, as is done in languages like Haskell, Standard ML, and O'Caml. Our thesis is that functional programmers can gain all the advantages of dependent-style programs without using real dependent types, and without materially changing the style in which they currently work.

The first key to this style of programming is the use of Generalized Algebraic Datatypes (GADTs), a generalization of the normal Algebraic Datatypes (ADTs). Implementing GADTs in a functional language requires only a small, backward compatible, change to the ADT notion that is easily understood by functional programmers. The second key is the use of extensible kinds, (the ability to create new types whose kind is different than *star*).

Once a language has GADTs and extensible kinds it can support several closely related concepts such Refinement Types[12, 46, 9], Guarded Recursive Datatype Constructors[42], Inductive Families[8, 11], First-class phantom types[6], Wobbly types[17], Silly Type Families[1], and Equality Qualified Types[32, 35]. These new features allow functional programmers to code any dependently-typed program by using some simple encoding tricks, yet still use the programming style to which they are accustomed. There are many examples of the usefulness of such concepts in the recent literature[3, 5, 13, 26, 29, 31, 45].

1. Where are we going?

This paper is a *vision* of how functional programmers might structure their programs in a way that makes them more reliable and trustworthy. It is *not* a vision of the *far-off* future, but one attainable in just a few years. We have written this paper to stimulate a discussion about how we might get there. To answer the question: *What should the functional languages of the near future look like?* We believe that there has been a serious lack of debate in this area in the last few years. One way to start any discussion is to propose something concrete, and then let others propose contrasting views. This is our strategy. We have been exploring some previously unexplored points in the design space of programming languages, and are ready to make a proposal. It is not an abstract one. We have built a real implementation we call Ωmega. Every example in this paper is a type-checked, running program in Ωmega. Yet, there is still much work to do, and this paper is in part an invitation for you,

ICFP '05 September 26-28, 2005, Tallinn, Estonia

the reader, to join the discussion. It is our sincere hope that an active debate about where we should be headed will stimulate new ideas that will migrate into languages of the future.

2. Our vision.

There is a huge semantic gap between what the programmer knows about his program and the way he has to express this knowledge to a system for reasoning about that program. It is necessary to narrow this gap to consistently build reliable and trustworthy programs. One way to reason about program properties is to use the Curry-Howard Isomorphism. Theorem provers (Coq[40], Isabelle[27]), logical frameworks (Twelf[28], LEGO[20]), proof assistants (ALF[24], Agda[7]) and dependently typed languages (Epigram[21], RSP[38]) provide a means to put the Curry-Howard Isomorphism to work, but each requires not only learning a whole new system, but also a whole new programming paradigm as well. This paper hopes to show that this powerful tool can be put to work in an ordinary functional language without changing the functional programming style. The strategy we employ is the following.

- Design a language that supports construction of programs and their properties simultaneously. Base it on an existing language so that it is easy for existing programmers to learn. **Design Choice -** We chose Haskell but a similar approach could be followed in many other languages.
- Start simple. This might require removing some features. **Design Choice -** To meet the simple criteria we have removed the class system of Haskell. We have tried hard to keep all the other features of Haskell not affected by this choice intact.
- Add additional small (backward compatible) features to support dependent-style programs, but do not disturb the functional programming style. In particular, leave intact the phase distinction between values and types. **Design Choice -** We have added several small extensions to Haskell such as Generalized Algebraic Datatypes (GADTs) and the ability to extend the kind system to facilitate our goals.
- Demonstrate that this is possible by building a non-trivial implementation and programming up a wide array of examples from the literature. **Design Choice -** We have built a small interpreter for our Haskell derivative we call Ωmega. We have actively sought out examples from the literature on theorem provers, logical frameworks, dependent types, language design, and formal methods. We have coded each example in Ωmega.

While preliminary, we believe our experience has much to offer to the community at large. We have coded up scores of examples from the literature (some of which have been published else where[26, 32, 35]) and have been more than pleased with the results. We have found it easy to capture the essence of the examples we have attempted, and yet our implementations still look like Haskell programs. In addition, the learning curve for Haskell programmers appears not to be too steep. The Ωmega interpreter is now available (`http://www.cs.pdx.edu/~sheard/Omega/`) for public use and we hope you try it out.

This paper is more a vision of the future than a paper filled with technical results. Think of it as a programming pearl for the next century. Our vision (and our implementation) would not be possible without the reported results of many others. We try to attribute credit fairly as we proceed, but we like to think a few of the ideas are our own (especially the notion of extensible kinds).

3. Generalized Algebraic Datatypes.

Algebraic data types are an abstraction available in many functional languages (such as Haskell, ML, or O'Caml) that allow users to define inductively formed structured data. They generalize other forms of structuring data such as enumerations, records, and tagged variants. For example, in Haskell we might write:

```
data Color = Red | Blue | Green                   -- An enumeration type
data Address = MakeAddress Number Street Town     -- A record structure
data Person = Teacher [Class]  | Student Major    -- A tagged Variant
```

We assume the reader has a certain familiarity with ADTs. In particular that values of ADTs are constructed by *constructors* and that they are taken apart by the use of *pattern matching*. Two valuable features of ADTS are their ability to be *parameterized by a type* and to be *recursive*. A simple example that employs both these features is:

```
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)`. In fact the type constructor `Tree` can be applied to any type whatsoever. Note how the constructor functions (`Fork`, `Node`) and constructor 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
```

An annoying restriction. 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*. 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), an alternate syntax is necessary to remove the range restriction. In Ωmega 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:

```
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
```

In addition to the second declaration for `Tree` (which doesn't require the added flexibility), we introduced the new type constructor `Term` which does. `Term` is classified by the kind `*0 ~> *0`. This means it takes types to types. `*0` is the kind that classifies all types that classify computable values. For more on kinds, see Section 6. 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 `Pair` is a non-polymorphic instance of `Term`.

`Terms` 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")
```

Using typed object-level terms, it is impossible to construct meta- level data structures representing object-level terms that are ill-typed, because the meta-language type system rejects such programs. Attempting to construct an ill-typed object term, like `(App (Const 3) (Const 5))`, causes a meta-level (Ωmega) type error. GADTs also allow one to construct a tagless interpreter. 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)
```

In a language without GADTs we would need to employ an untyped object level representation and make the range of the eval function a universal value domain like: `data V = IV Int | PV V V | FV (V -> V)`. See [26] for a detailed discussion of this phenomena.

While we worked hard to make this look like Haskell programming, there are some key differences. First, the prototype declaration (`eval :: Term a -> a`) is required, not optional. Functions which pattern match over GADTs can be type checked, but type inference is much harder (see [36] for work on how this might be done). Functions that don't pattern match over GADTs can have Hindley-Milner types inferred for them (see [17] for how this mixture of type-checking and type-inference is done). Requiring prototypes for only some functions should be familiar to Haskell programmers because polymorphic-recursive functions already require prototypes[18].

4. GADTs as Equality Constrained Types.

GADTs can be explained in terms of the well studied notion of qualified types [14]. The *explicit* constructor declaration can be translated into a normal datatype declaration that uses equality-qualified types[6, 25]. We illustrate both forms below for object-level typed-term representations.

```
-- explicit constructor function declaration
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

-- equivalent using equality-qualifications
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)
```

We've seen the explicit declaration form in our earlier discussion. It lists the full types for each constructor. We are free to make the range type of each constructor be any instance of the type being defined. This form can be easily translated (see Figure 1) into the second form that explains the removal of the range restriction in terms of qualified types. In the equality qualified syntax we use the original syntax for `data` declaration, but also allow every constructor to be followed by a `where` clause which lists a set of type equalities that must hold for that constructor.

Constructor functions (`Const`, `Pair`, and `App`) construct elements of the `data` type `Term`. Their types are implicitly described. For example, the clause in the `Pair` declaration:
`exists x y . Pair (Term x) (Term y) where a = (x,y)`
introduces the `Pair` constructor function. Its range is the type to the left of the equal sign in the declaration (`Term a`), and its domain comes from the types following the constructor name.

The constructor function `Pair` has the type (`Pair:: Term x -> Term y -> Term a`) *provided* `a=(x,y)`. We capture this formally by writing the qualified type: `Pair::(forall a x y.(a=(x,y))=> Term x -> Term y -> Term a`. The equations behind the *fat arrow* (`=>`) are equality qualifications. Since `a` is a universally quantified type variable, there is only one way to *solve* the qualification `a=(x,y)` (by making `a` equal to `(x,y)`). Because of this unique solution, `Pair` also has the type `(forall x y . Term x -> Term y -> Term(x,y))`. This type guarantees that `Cons` can only be applied in contexts where `a=(x,y)`. This is the same type we wrote using the explicit constructor declaration method.

Figure 1. The Ωmega implementation translates explicit data declarations to equality constrained normal ones. Currently both forms are accepted by the implementation. We currently find both forms useful in certain contexts. The algorithm to translate is:

1. From the kind of the type constructor compute its arity n. For example, `(Term:: *0 ~> *0)` so `Term` has arity $n = 1$.
2. Invent n new type variables, and construct a new type by applying the type constructor to them. For example, let the one new type variable be `m`, and then the new type is `(Term m)`.
3. For each constructor, unify the new type (from step 2) with the range of the constructor producing a substitution σ. For example, the constructors are `Const`, `Pair`, and `App`. We'll consider `Pair`, the range is `Term(a,b)` and the substitution σ (as an association list) is `[(m,(a,b))]`.
4. Split the substitution into two parts, the first binds new type variables to other type variables, and the other binds new type variables to type expressions. For example for the constructor `Pair` we get `[]` and `[(m,(a,b))]`.
5. Create an ordered list of qualified types by constructing equalities from the second part (of step 4) as the qualifiers, and the domains from the explicit `data` declaration as the list elements. For example: `(m = (a,b)) => [Term a, Term b]`
6. Apply the variable to variable substitution to the list (replacing old variables with the new ones introduced in step 2). For the `Pair` example, the substitution is the identity substitution so the type remains the same.
7. Existentially quantify over all type variables which are not the ones introduced in step 2 above. For Example: `exists a b.(m = (a,b)) => [Term a, Term b] .`
8. Finally, construct a complete data declaration using the new type introduced in step 2 as the left-hand-side of the equality. For each constructor, use the list elements and the quantification to construct the domains of the constructor, and the qualification (if any) as the `where` clause for that constructor.

```
data Term m =
  exists a b.Pair (Term a)(term b) where m=(a,b)
```

Human Factors Question. Counting the original form (no `where` clauses), there are three different mechanisms for declaring new data types. Perhaps we should standardize on a single form?

Type Checking GADT Pattern Matching. The type checker for Ωmega collects, propagates, and solves equality constraints using the same mechanism Haskell uses to deal with class constraints. It uses the algorithm described in Ralf Hinze's paper[6], which we describe briefly below.

When type-checking an expression, the Ωmega type checker keeps two sets of equality constraints: *obligations* and *assumptions*.

Obligations. The first set of constraints is a set of *obligations*. Obligations are generated by the type-checker when a program uses an expression with an equality constrained type. For example the term `(Pair (Const True) (Const 5))` is assigned a type `(Term a)` and generates an obligation that `(a = (Bool,Int))`.

Assumptions. The second set of constraints is a set of *assumptions* or *facts*. Whenever, a constructor based pattern appears in a binding position, and the constructor has an equality qualified type, the equalities from the qualification are added to the current set of assumptions in the scope of the pattern. These assumptions can be used to discharge obligations. Consider type checking the third clause of `eval`.

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

The system is type checking, so the prototype declaration assigns the domain of `eval` to the type `Term a`. This information is propagated by the type checker to the pattern `(Pair x y)` which is also assigned the type `Term a`. This assigns the types `(x::Term x)` and `(y::Term y)` and propagates the assumption `a=(x,y)` into the scope of the pattern. From the type of `eval` and the typing rules for pairs, the term `(eval x,eval y)::(x,y)`. But, from the prototype, the type checker expects it to have the type `a`. Fortunately, under the assumption `a=(x,y)` it is trivial to prove that `a` equals `(x,y)` so the type checker succeeds.

5. Type Indexes versus Type Parameters.

The parameter of the type constructor `Term` 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 `Trees` that are of type `a`. In fact, `Trees` are polymorphic. Any type of value can be placed in the "sub component" of type `a`. The type of the value placed there is reflected in the `Tree`'s type. Contrast this with the `a` in `(Term a)`. Instead, the parameter `a` is used to stand for an abstract property (the object level type of the term represented). The `where` qualifications restrict the legal instances of `a`. Type parameters used in this way are sometimes called index types[43, 46], and will play an important role in what follows.

Research Question. If parameters to type constructors play two qualitatively different roles (polymorphic parameters and type indexes), perhaps there should be two different mechanisms with syntactically different expression?

6. Kinds: Defining New Indexes.

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
```

The type `Z` is classified by `Nat`, and `S` is classified by `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)`, and `(Nat:: *1)`. The classification `Nat::*1` indicates that `Nat` is at the same "level" as `*0` — they are both classified by `*1`. There is an infinite hierarchy of classifications. `*0` is 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, and kinds in Figure 2.

The introduction of new kinds is very important because it allows the user to define indexed GADTs that are indexed by structures which cannot by captured by the structure of types classified by `*0`.

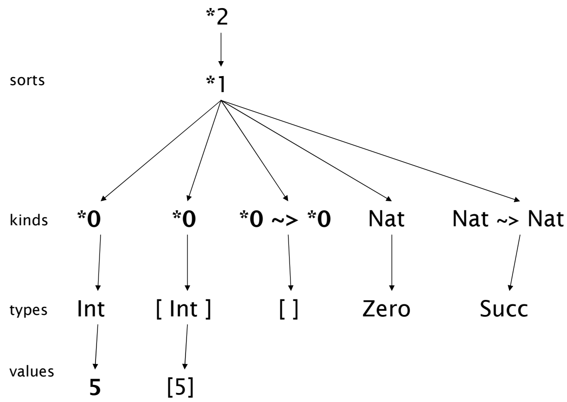
6.1 A Kind Example: Units of Measurement.

An interesting use of kinds other than *0 is the introduction of scalar values with units of measurement. For example, consider the measurement of temperatures. There are several different scales that might be used. We can introduce a new kind that enumerates the scales (`TempUnit`).

```
kind TempUnit = Fahrenheit | Celsius | Kelvin

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

Figure 2. 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.




```
  K :: Float -> Degree Kelvin
```

The `data` declaration for the GADT `Degree` is indexed by `TempUnit`. The constructors each lift a scalar `Float` to a `Degree` with a different unit. Operations on `Degree` are easy to define. Note that in Ωmega, `(#+)` is the infix operator for addition for `Float`.

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

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

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

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

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

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

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

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

where we have shown only two of the nine possible cases.

7. Refining Types.

The purpose of a type refinement is to make finer distinctions among the values of a type than an ordinary type can support. For example, we might want to distinguish between empty and non-empty lists using the list's type. Several approaches to refining types have been investigated. The first approach[12] relies on naming recursively defined subsets of a previously defined type, and using abstract interpretation over the lattice defined by the subsets. The second approach[46] involves indexing a type by an index set. This approach is naturally captured by using GADTs and extensible kinds.

To do this we first introduce a new kind to represent the index set. We then define a GADT, one of whose indexed parameters has this kind. Consider sequences where the length of the sequence is encoded in its type. 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)
```

Functions over index refined types (like all functions defined by pattern matching over GADTs) require a prototype declaration, and their types often witness important properties of the function. For example, a map function for sequences with type $(a \rightarrow b) \rightarrow Seq\ a\ n \rightarrow Seq\ b\ n$ 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[44], and eliminating array bound checks[45]. Both of these systems employ a decision procedure for solving linear inequalities on integers which is used by the type checker.

We can illustrate this on a much simpler scale by the Ωmega program in Figure 3. 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, 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}`).

Ωmega allows programmers to define arbitrary index sets, and to write arbitrary type functions as confluent and terminating sets of rewrite rules over the index sets. Still, the current mechanism is quite weak. Suppose we had declared the range of `app` as `Seq a {plus m n}` rather than `Seq a {plus n m}` (i.e. switched the order of `m` and `n`). We would then have to solve (`S{plus m t} = {plus m (S t)}`), which matches none of the rewrite rules defining `plus`. A stronger system would have to support the definition, proof, and use of arbitrary rules over type functions. This means type functions may not be confluent and terminating, and that

Figure 3. Defining and using functions at the type level are required for type checking some functions over types with indexes.

```
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 type-checker must incorporate a full blown theorem prover. Of course this is exactly the role of the decision procedure in [44, 45].

Research Question. Ωmega provides a mechanism for defining multiple index sets, different theories over each of them, and for programs (or even individual GADTs for that matter) to be indexed by multiple index sets. Is it possible for users to define (or import) their own decision procedures for each index algebra? What about interaction between index algebras? Adding rich computation mechanisms at the type-level will be an important attribute of future languages.

7.1 Illustrating Curry-Howard – Proofs & Witnesses.

GADTs can model a rich collection of relations between index types by defining witness types. A witness is a value with an indexed type whose existence witnesses a relationship between its indexed type parameters. The very existence of the witness (i.e. a non-bottom value with the given type) implies that the property must be true. Witnesses to untrue properties cannot be constructed since such values would be ill-typed. The simplest witness is the `Equal` type constructor.

```
data Equal a b = Eq where a=b
```

A value of type `Equal a b` is a dynamic witness to the static property that `a` = `b`. We will see some uses for this type later in the paper. Another example is `Sum`. A value of type `(Sum n m p)` can only be constructed if the `Nat` kinded parameters `n`, `m`, and `p` are in the following relationship: `n + m` = `p`.

```
data Sum:: Nat ~> Nat ~> Nat ~> *0 where
  SumBase:: Sum Z u u
  SumStep:: (Sum u v w) -> Sum (S u) v (S w)
```

This witness allows us to define an alternative append function for static-length lists.

```
data Ans a n m = exists p . Ans (Sum n m p) (Seq a p)

append :: Seq a n -> Seq a m -> Ans a n m
append Nil ys = Ans SumBase ys
append (Cons x xs) ys = Ans (SumStep p) (Cons x zs)
   where (Ans p zs) = append xs ys
```

If we can't statically determine the length of appending two lists, we can dynamically compute both the new list and a witness to its relationship between the lengths of the inputs. These are the two components to the constructor Ans. Other interesting witness types over the Natural numbers include tests for unary odd and even relations, and binary not equal relation.

```
data Even:: Nat ~> *0 where
  Z:: Even Z
  E:: Odd m -> Even (S m)

data Odd:: Nat ~> *0 where
  O:: Even m -> Odd (S m)

ex3 :: Even (S (S Z))
ex3 =  Even (Odd Zero)
```

```
data NE:: Nat ~> Nat ~> *0 where
  Z1 :: NE (S x) Z
  Z2 :: NE Z (S x)
  Step:: NE x y -> NE (S x) (S y)

ex4 :: NE (S (S Z)) (S (S (S u)))
ex4 = Step (Step Z2)
```

GADTS often play the role of dynamic witnesses to static properties. They allow the programmer to move freely between static and dynamic checking, and to control when certain computation should happen, at compile-time or at run-time.

8. Singleton Types.

It is sometimes useful to compute with index types as if they were values. But since one of our design goals was to strictly separate the worlds of types and values this seems beyond our reach. Fortunately, we can use GADTs to build a reflection of any type in the value world. And we can then compute over this reflection instead of the type itself. Consider the reflection of the Nat type into the value world where the value constructors of the data declaration for Nat' mirror the type constructors in the kind declaration of Nat, and the type index of Nat' exactly captures the kind reflected.

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

```
ex5 ::  Nat' (S (S Z))
ex5 = S (S Z)
```

Note that we exploit the fact the name space for values and the name space for types are separate. We use the same name Z for the constructor function of the type Nat', and the type constructor Z of the kind Nat (and we do the same for S as well). This is overloading is deliberate, because the structures Nat' and Nat are so closely related.

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.

We can use singleton types when translating a normal list ([a]) into (Seq a n). Of course we can't know the length, n, of the translated list statically, so we compute a representation of its length and its translation simultaneously.

```
data DynSeq a = exists n . DS (Nat' n) (Seq a n)

toSeq :: [a] -> DynSeq a
toSeq [] = DS Z Nil
toSeq (x:xs) = DS (S m) (Cons x ys) where (DS m ys) = toSeq xs
```

Because singleton types are ordinary values implemented using GADTs, and because the type checking rules for pattern matching over GADTs propagate type equalities they support a number of very interesting programming paradigms. Writing a program that manipulates singleton types allows the programmer to encode operations that the type system (with its limited computation mechanism – essentially unification and solving equalities between types) cannot. Thus, typing problems that cannot be solved by the type system can be programmed by the user when necessary.

An illustrative example of this is dynamic typing. The following examples are motivated by a paper by Baars and Swierstra[3] and a paper by Hinze and Cheney[13] which both appeared at roughly the same time. We define a singleton type that reifies an interesting subset of the types classified by `*0` into the value world.

```
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]

data Dyn = exists t . Dyn (Rep t) t

print :: Dyn -> String
print (Dyn r t) = showr r t
```

```
showr :: Rep t -> t -> String
showr Int n = showInt n
showr Char c = showChar c
showr (Prod a b) (x,y) =
  "("++showr a x++","++showr b y++")"
showr (Arr a b) f = "fun"
showr (List Char) x = x
showr (List x) xs = "["++f xs++"]"
  where f [] = ""
        f [y] = showr x y
        f (y:ys) = showr x y++","++f ys
```

We define the type `Dyn` to existentially hide both a representation of a type and a value of that type. Because the `Rep` is a singleton type, we can essentially discover the *type* of the hidden value by investigating the *structure* of the `Rep` value. For example we can define a print function that can turn any dynamic value into a string.

The typing rules for GADTs propagate the precise equality qualifications necessary to type-check the patterns involving the constructors of `Rep`. Even more powerful programming patterns are possible. Suppose a function receives a `Dyn` value that it expects to be some fixed type (arbitrarily complex, as long as it can be described by `Rep`). The strategy is to perform a single dynamic type-check, and if it succeeds, to proceed by processing the dynamic value as if its type was statically known. If the dynamic type-check fails, then take some corrective action instead. We capture this pattern by the function `dyn_type_check`.

```
dyn_type_check :: Rep t -> Dyn -> (Maybe t -> a) -> a
dyn_type_check expected (Dyn r x) continuation  =
  case test expected r of
    Just Eq -> continuation (Just x)
    Nothing -> continuation Nothing
```

The key to this function is the `test` function which takes two arbitrary `Rep` types and tests them for structural equality. If they have the same structure, then they represent the same type. This fact is captured by returning a `(Maybe (Equal t type-of-x))` witness type. This is a dynamic witness (constructed at run-time, by observing the *structure* of the two `Reps`) to the static property that the type `t` is the same as the type of `x`. If the test succeeds (i.e. the `Just` clause), we can assume that `x` is classified by `t` and apply the `continuation` to the element, `x`. Defining `dyn_type_check` depends on our ability to define the function `test`. How do we do this?

9. Constructing Witness Objects.

The test function works by simultaneously investigating the structure of its `Rep`-typed parameters. Since `Reps` are singleton types, If the parameters have exactly the same structure then they must represent the same type, and this will be captured in the `Equal` witness object returned. In the code below we have used *as* patterns as a mechanism to label certain patterns. Recall, in an *as* pattern `(x@([xs],c))`, the variable `x` is bound to the whole value that matches the pattern `([xs],c)`. This is in addition to the binding of the variables `xs` and `c` to sub-components of the value. In the definition below we use the *as* patterns only for their ability to label a pattern so we can identify it uniquely in the discussion below.

```
test :: Rep a -> Rep b -> Maybe(Equal a b)
test (x1@Int) (x2@Int) = Just Eq
test Char Char = Just Eq
test (Prod s t) (Prod m n) = do { (y1@Eq)<- test s m; (y2@Eq)<- test t n; return Eq}
test (Arr a b) (Arr m n) = do { Eq <- test a m; Eq <- test b n; return Eq }
test (List a) (List b) = do { Eq <- test a b; return Eq }
test _ _ = Nothing
```

The function `test` uses Ωmega's monadic `do` notation (inherited from Haskell). In Haskell, typing the `do` notation depends upon the class system. In Ωmega, there is no class system, so Ωmega uses a more syntactic mechanism to type check `do` statements. See the Ωmega users guide[33] for details.

Consider the `Int` case. The equality qualifications attached to the constructor labeled `x1` tells us that in the scope of the pattern `a=Int`, and the one labeled `x2` tells us that `b=Int`. The prototype informs the type-checker that the right-hand-side of the equality must have type (`Equal a b`). The term that constitutes the right-hand-side is the polymorphic constructor `Eq`, with qualified type `(forall c d . (c=d) => Equal c d)`. Using this term generates the obligation that `c=d`. The Prototype declares that `(Equal a b) = (Equal c d)`. From this it follows that `a=c` and `b=c`, But under the Assumptions that `a=Int` and `b=Int` we only need to show `Int=Int` which is trivially discharged.

In the `Prod` case (and the `Arr` case which is similar) a slightly different approach is used. This approach uses the recursively returned `Equal` witness objects to witness equality of subcomponents, which are then assembled into larger witness objects. In this description we use the convention that each of the pattern-bound variables has a type that is the same as its name. Thus we see that `(Prod s t)::Rep(s,t)`, and that `(Prod m n)::Rep(m,n)` and the assumptions `a=(s,t)` and `b=(m,n)` hold in the scope of the pattern. The equality witness labeled `y1` is classified by `(Equal s m)`, and the equality witness labeled `y2` is classified by `(Equal t n)`. The type of the polymorphic constructor `Eq::(Equal z z)` then propagates the additional assumptions `s=m` and `t=n`. These 4 assumptions are sufficient to show that `a=b`.

Note, while the two `Rep`s must be completely traversed to build the witness equality, the witness equality itself has unit size. The functions that use the witness (like `dyn_type_check`), never need to traverse the witness at all. Just checking for the `Just` tag of the surrounding `Maybe` is sufficient, to know that in the arm of the case tagged by `Just`, the two types are equal. A perfect synergy between static and dynamic type checking!

Using `Rep` types is a powerful programming paradigm. We can use a variant of it to dynamically construct `Dyn` objects, by parsing them from an input stream. We illustrate this by using the Parsec[19] parsing combinators (though we omit the `Arr` case for simplicity).

```
parseDyn :: Parser Dyn
parseDyn =
  (do { n <- intLit; return(Dyn Int n)})  <|>
  (do { c <- charLit; return(Dyn Char c)})<|>
  (do { char "("; (Dyn a x) <- parseDyn
      ; char ","; (Dyn b y) <- parseDyn; char ")"
      ; return(Dyn (Prod a b) (x,y))})    <|>
  (do {xs <- brackets (many parseDyn); allsame xs})

allsame :: [Dyn] -> Parser Dyn
```

```
allsame [] = return (Dyn (List Int) [])
allsame [(Dyn r x)] = return (Dyn (List r) [x])
allsame ((Dyn r x):ys) =
  do { Dyn (List s) xs <- allsame ys; Eq <- test s r; return(Dyn (List s) (x:xs))}
```

The key is the ability to do dynamic type-checking. For integers and characters we simultaneously construct a `Rep` and its value. For pairs we just combine the pieces of the sub-components. But parsing a list is more complicated. We must check that each parsed item has the same `Rep` so that the list is uniform in the type of its element. Note how we take advantage of the `test` function to do this in the function `allsame`. There is one complication here. The string `"[]"` representing the empty list will always parse as having type `[Int]`. This is only partially mitigated by the fact that strings representing lists with at least one element will be parsed with the correct type.

Research Question. Is it possible to build `Rep`-like singleton types to represent polymorphic types? While we have tried many approaches we are not yet satisfied with the generality of any of them.

We can use singletons to build typed-terms from untyped ones.

```
data Term' = Const' Dyn | Pair' Term' Term' | App' Term' Term' -- untyped terms
data Judgment = exists z . J (Rep z) (Term z)

trans :: Term' -> Maybe Judgment
trans (Const' (Dyn r t)) = Just(J r (Const t))
trans (Pair' x y) =
  do { J a m <- trans x; J b n <- trans y; return(J (Prod a b) (Pair m n))}
trans (App' f x) =
  do { J (Arr d r) g <- trans f; J b y <- trans x
     ; Eq <- test d b; return(J r (App g y))}
```

There is an interesting pattern going on here. We have used it so often we ought to give it a name. The pattern is constructing a new datatype that existentially hides a singleton type and an element that depends on the singleton type's indexed type-parameter.

```
data Exists singleton term = exists t . Ex (singleton t) (term t)
```

Now that we recognize this pattern we can use the `Exists` type constructor instead of defining a number of similar type constructors

```
type Ans a n m = Exists (Sum n m) (Seq a)
type DynSeq a = Exists Nat' (Seq a)
data Id x = Id x
type Dyn = Exists Rep Id
type Judgment = Exists Rep Term
```

This pattern is sometimes called a *dependent sum*. The type constructor `Covert` plays a similar role.

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

toNat :: Int -> Covert Nat'
toNat 0 = Hide Z
toNat n = case toNat (n-1) of Hide b -> Hide(S b)
```

It is useful when you want to hide the indexed type parameter of a singleton type, without pairing it with another type. In the function `toNat` we use it to convert an `Int` into a `Nat'` whose index is hidden.

10. Dependently Typed Programs.

In a dependently typed program, the type of some terms are dependent on the values of other terms. Since we have made the design decision to separate values from types, we will need to use singleton types. For example, consider the family of functions that sums 1, 2, 3, ... n inputs. The first few functions in this family are:

```
0) \ x -> x     1) \ x -> \ y -> x+y     2) \ x -> \ y -> \ z -> x+y+z
```

Can we write a single, well-typed function that implements all the functions in this family? It would need a dependent type: `f :: Pi (n:Natural). Int -> sumfamily n` where `sumfamily` is a function from Natural numbers to types which meets the following specification: `sumfamily 0 = Int`, and `sumfamily n = Int -> sumfamily (n-1)`. In our hypothetical dependently-typed language:

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

Note how the type of `f` depends on the value of `f`'s first parameter. This violates our design decision separating values and types. But by using singleton types in Ωmega, we can define instead:

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

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

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

The use of singleton types is only one way we can accomplish this. We have already seen another (but similar) mechanism in the function `eval:: Term a -> a`. In `eval` we arranged for the appropriate type information to be present as a type index in the type of `Term`, and the dependence is the trivial identity dependence. The index does not necessarily have to be the index of a singleton type. In a system where types may depend on values we might have chosen `eval :: Pi (t:Term). typeof t`. Where `typeof` was a function from terms to their types. Instead, because the type of a term is such an valuable invariant of a term, we choose to encode it as an index of `Term`.

Research Question. We conjecture that these techniques are sufficient to encode any dependently-typed program. The strict separation between types and values empowers the user to specify when computation should occur – at compile-time or run-time. What mechanisms are necessary to support the programmer in these tasks? For some speculation on our part, see Section 14.

11. Types Express Properties.

Dependent types can be used to express rich properties of programs. To illustrate this we will develop a program that uses witness types to define a linear sequence datatype that is always sorted. This type witnesses that all well typed values of that type will have their elements in sorted order. We will then develop a sort program that constructs such a list. The type of the sort program encapsulates its correctness proof. To begin, we define a less-than-or-equal relation witness type `LE:: Nat ~> Nat ~> *0`, for the natural numbers, and sorted-sequence datatype that stores singleton `(Nat' n)` values in sorted order.

```
data LE a b = LeBase where a=b | exists c . LeStep (LE a c) where b = S c

data SSeq n = Snil where n = Z | exists m . Scons (Nat' n) (LE m n) (SSeq m)
```

The key to the definition of `SSeq` is that it is parameterized by the index type of its largest element, and that every cons operator stores a proof (`LE` witness) that the largest element in the tail of the list is less than or equal to the current element. An example is: `Scons (S Z) (LeStep LeBase) (Scons Z LeBase Snil)` which is classified by `SSeq (S Z)`. Note how the index of the largest element appears as the index of the type of the sequence.

Analogous to the function `test:: Nat' n -> Nat' m -> Maybe(Equal n m)`, we would like a function that given two singleton `Nat`s potentially returns a proof that the first was less than the second. It turns out to be more convenient to return one of two proofs. Either `(LE n m)` or `(LE m n)`. This is the job of the function `compare`.

```
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) = 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)
```

The `compare` function makes use of the sum type `(a + b)`. It has constructors `L::a -> (a + b)` and `R::b -> (a + b)`, and the natural mapping function `mapP:: (a -> c) -> (b -> d) -> (a + b) -> (c + d)`. The helper function `g:: LE x y -> LE (S x) (S y)` can be thought of as a lemma: $x < y \Rightarrow (x+1) < (y+1)$, which is used to obtain the desired result, from the induction step (`compare x y`).

An insertion sort is constructed by inserting the elements from an unsorted list, one at a time into a sorted list. If the insertion process maintains sorted-ness, then the resulting list will be sorted. It is straightforward to define an insert operation whose type guarantees that it returns a sorted result.

```
insert :: Nat' a -> SSeq b -> ((SSeq a)+(SSeq b))
insert z Snil = L(Scons z (magnitude z) Snil)
   where magnitude :: Nat' a -> LE Z a
         magnitude Z = LeBase
         magnitude (S x) = LeStep(magnitude x)
insert x (xs@(Scons y p zs)) =
   case compare x y of
     R q -> L(Scons x q xs)
     L q -> case insert x zs of
              (L mm) -> R(Scons y q mm)
              (R mm) -> R(Scons y p mm)
```

Inserting an element with index `a` into a sorted sequence with maximum element index `b`, could either return a sequence with maximum index `a` (if the element index is larger than the index of the list) or with maximum index `b` (if the element index is smaller than the index of the list). This is exactly the strategy of the `insert` function. In the case that the list is empty, we need to construct a proof that the index of the element is larger than zero. This is the role of the helper function `magnitude`.

Once we have `insert`, defining an insertion sort is easy. First we must express the fact that `sort` takes an unsorted list as input. We use the built-in lists of Ωmega. Because such lists must have elements with uniform type, we employ the `Covert` type constructor from Section 9. This existentially hides the index of the singleton type. We employ this "trick" twice. Once to build an actual Ωmega list with uniform elements for the input

([Nat' n] where n is hidden), and to hide the maximum index of the resulting sorted sorted sequence for the output.

```
sort :: [Covert Nat'] -> Covert SSeq
sort [] = Hide Snil
sort ((Hide x):xs) = case insert x ys of {L w -> Hide w; R w -> Hide w}
 where (Hide ys) = sort xs
```

To test our sort function, we can convert normal integers into (Covert Nat') types, construct a list of them, and then sort that.

```
test2 :: [Int] -> Covert SSeq
test2 xs = sort(map toNat xs)

x23 = test2 [0,3,1]
```

The result is exactly as expected.

```
Hide (S (S (S Z)))
     (Scons (S (S (S Z))) (LeStep (LeStep LeBase))
            (Scons (S Z) (LeStep LeBase)
                   (Scons Z LeBase Snil)))
```

12. Related Work.

The design space of languages which exploit of the Curry-Howard isomorphism is large. We have chosen a small corner of this space because it allows us to keep a particular programming style. That style includes a strict phase distinction between types and values, and a desire to minimize type annotations and other administrative work. A comparison of related work is best performed in that light.

The most important feature of GADTs 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[40], Isabelle[27]), logical frameworks (Twelf[28], LEGO[20]), proof assistants (ALF[24], Agda[7]) dependently typed languages (Epigram[21], RSP[38]). The most important consequence of this design choice 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 can be used when their 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 the two 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).

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

We use Set to be the "type of types", and the constructors must be explicitly applied to their type parameters (a and n). When 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 Sect :: *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 stated goals. Several systems[21, 2] 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.

Several systems choose a point in the design space closer to ours, where the distinction between types and values is preserved. We owe much to these works for inspiration, examples, and implementation techniques. These include Guarded Recursive Datatype Constructors[42], First-class phantom types[6], Wobbly types[17], and Silly Type Families[1]. 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, there are only minor syntactic differences between Ωmega and Guarded Recursive Datatype Constructors(GRDC) [42]. Even without kind extension, one could imagine using higher kinded type indexes like `(Tree:: *0 ~> *0)`. This is supported in Ωmega because arbitrary type equalities are allowed in the `where` clause in a `data` declaration, but is not supported in GRDC. 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[16], extensible kinds, and type functions.

Wobbly Types[17] describes Simon Peyton Jones' 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[1] 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 this paper demonstrates, these obstacles no longer hold. This interesting paper is way before its time, and is now available on the first author's web site by permission of Lennart.

The work on Refinement Types[46, 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[36] on type inference (rather than type checking) for GADTs. And Martin Sulzmann's work[37, 39] 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 germ of this idea originally appeared in 2000[41], and was well developed two years later in 2002[3, 13].

By judicious use of equality types, essentially replacing equalities in where clauses with Equal values, one can code up any GADT like structure. Consider the Term GADT redone this way.

```
data Term a = Const a
            | exists x y.Pair (Equal 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 (Eq id)
```

Programming with Equal 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[25] illustrates this on many examples. It also illustrates how important it is for the compiler to maintain the equality constraints.

The use of kinds to classify types has a long history[4, 15, 23]. Adding extensible kinds (and higher classifications) to a practical programming language like Ωmega seemed to be the natural next step. 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.

Research Question. Our primary interests is not how to type programs that use GADTs, but how to use GADTS to good effect[26, 32, 34, 35]. In particular, what other features (such as monads, new kinds, and rank-N polymorphism) magnify their effect, and what programming patterns (witness objects, singleton types, and indexing data declarations) can be used to solve recurrent problems? Will these techniques lead to more reliable and trustworthy programs?

13. Soundness.

For the Ωmega experiment to succeed witness (proof) objects in Ωmega must attest to true things. That is, regarded as a logic, Ωmega should be sound. We conjecture that the approach embodied by Ωmega can be made sound. As we migrate Ωmega from a toy interpreter to a sound logic, 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 as an effect. All computations that are not explicitly identified as potentially divergent (by their types) should terminate.
- **Consistency.** There will be types in the terminating fragment of the language that do not have any inhabitants.

The type system of Ωmega will partition the language into terminating and possibly non-terminating subsets using monads. This makes it possible to reason soundly about the terminating subset, but to include other kinds of programs for practical reasons. We will employ an explicit termination check as part of Ωmega's type checking algorithm. The validation of the correctness of that termination check will establish that all programs (without explicitly divergent types) accepted by the type checker terminate.

The design of Ωmega separates types from values by design. It allows computation at both levels. 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.

Research Question. Is it 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 possible to define type functions that cause compilation to 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.

[1] I.e. given a witness with type (Eq (a,b) (c,d)) it was 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.

As Ωmega matures we expect to implement Ωmega in Ωmega . In that implementation we expect to explicitly represent the subject reduction property of the language in its own type system.

Fundamentally, all of the enrichments provided by indexes are only refinements of structural types expressible in System F^ω or Haskell. The types classified by kind `*0` comprise a relatively conventional advanced type system. Does the introduction of new kinds, and the types classified by these kinds pose any unacceptable risks? We don't think so. Many of the risks of Ωmega are already inherent in Haskell. Consider:

```
data Eq a b = Eq (forall f . f a -> f b)
data Z
data S x
data Seq a n = Nil (Eq n Z) | forall m . Cons (Eq n (S m)) a (Seq a m)

nil :: Seq a Z
nil = Nil (Eq id)
cons :: a -> Seq a b -> Seq a (S b)
cons = Cons (Eq id)
```

The strategy (1) introduces Leibniz equality (`Eq`), and (2) defines uninhabited types for use as type indexes (`Z` and `S`), and (3) defines GADT-like structures (static length lists `Seq`) by using explicit equality witnesses as components of the constructors (`Nil` and `Cons`) to get the effect of GADTs.

We believe that any GADT, and any new kind, can be faked using these three patterns. The cost of this approach is the increased complexity when writing well typed functions over these "faked" GADTs. The programmer has to explicitly construct the appropriate *casting* functions from the Leibniz equality witnesses. The overhead, in terms of programmer productivity, can be considerable, as is illustrated in Emir Pasalic's thesis[25]. In addition, unlike the `Eq` witness from Section 9, the casting functions have real cost associated with them, as they must traverse the complete structure of the object they are casting (even though they are provably identity functions). One big lesson from Pasalic's thesis is that while much is possible using this mechanism, little is practical. The overhead is overwhelming, and rather than leading to more reliable and trustworthy programs it quickly leads to programmer distress.

The Ωmega approach also supplies guarantees not available in Haskell, not the least of which is that the kind system acts as a type system for types, and rejects ill-formed types like `(S Int)`.

14. Future work and Conclusion.

We have many additional ideas we would like to explore. Most, involve either increasing the amount help the compiler can provide to the programmer, or increasing the amount of computation that can be done at compile-time. Some of the ideas we are currently exploring include:

- **Deriving Singleton types.** Singleton types are so useful, and their structure so regular, it should be possible top derive them automatically.
- **Poly-Levelic Declarations.** Some structures are so useful, we often write programs where isomorphic instances appear at both the value and type level. Examples include things like pairs, lists, and natural numbers, but there are certainly many more examples. A programmer should declare both the types and simple functions over the types once, and these declarations get promoted to any level. For example we might write

  ```
  level n
    data Nat:: *n where
      Z:: Nat
      S:: Nat -> Nat
  ```

```
add :: Nat -> Nat -> Nat
add Z y = y
add (S x) y = S(add x y)
```

When n=0 we get the type Nat and the values Z and S, when n=1 we get the kind Nat and the types Z and S, etc.

- **Additional Constraints other than Equality.** In Ωmega the only constraints in a qualified type are equality constraints. Such constraints are solved by unification. We imagine empowering the programmer do define additional types of constraints, and mechanisms to solve them, which would be incorporated into the type checker. For example one might write:

```
constraint LE:: Nat ~> Nat ~> Prop
LE Z x = T                   -- T is the only inhabitant of Prop
LE (S x) (S y) = LE x y
LE x Z = error "no Nat (other than Z) is less than or equal to Z"

data SSeq n = Snil                     where n=Z
            | Scons (Nat' n) (SSeq m) where LE m n
```

  Now, whenever Scons is used a static constraint is propagated. When constraints must be solved, the equations for LE are used. If the error notation is ever reached the constraint can never be solved. Otherwise, constraints are propagated in the same way normal class constraints are propagated in Haskell. Clearly we may obtain types whose constraints can never be solved, but because these types are uninhabited they should cause no problems.

We hope this paper has you thinking about where functional programming languages are headed. We hope you join in the discussion. In conclusion, we believe that allowing the programmer to state properties of his program is important, and dependent programming provides important leverage in this area. But most importantly, the key to programming dependently is for the language to provide most if not all of the administrative book keeping needs. In our opinion, this is the most important lesson. When that support is available, Haskell programmers, like ducks to water, take naturally to the dependent style of programming.

References

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

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

[3] 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.

[4] H. P. Barendregt. Lambda calculi with types. In D. M. Gabbai Samson Abramski and T. S. E. Maiboum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, Oxford, 1992.

[5] Chiyan Chen and Hongwei Xi. Meta-programming through typeful code representation. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP-03)*, ACM SIGPLAN Notices, pages 275–286, New York, August 25–29 2003. ACM Press.

[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] Tim Freeman and Frank Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 268–277. ACM Press, 1991.

[13] Ralf Hinze and James Cheney. A lightweight implementation of generics and dynamics. In Manuel Chakravarty, editor, *Proceedings of the ACM SIGPLAN 2002 Haskell Workshop*, pages 90–104. ACM SIGPLAN, October 2002.

[14] Mark P. Jones. *Qualified types: Theory and Practice*. PhD thesis, Programming Research Group, University of Oxford, July 1992.

[15] 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.

[16] 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/`.

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

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

[19] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world, October 04 2001.

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

[23] 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.

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

[25] 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.

[26] 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.

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

[28] Frank Pfenning and Carsten Schrmann. 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] Franois Pottier and Nadji Gauthier. Polymorphic typed defunctionalization. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL'04)*, pages 89–98, Venice, Italy, January 2004. Superseded by [30].

[30] Franois Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and other tales. Manuscript. Submitted, February 2005.

[31] Franois Pottier and Yann Rgis-Gianas. Towards efficient, typed LR parsers. Draft paper, September 2004.

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

[33] Tim Sheard. Omega users guide, March 2005. Available from:
`http://www.cs.pdx.edu/~Omega/` .

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

[35] 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/`.

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

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

[38] 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/`.

[39] 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/` .

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

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

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

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

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

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

[46] 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. An extended example.

This extended example is an Ωmega port of Conor McBride's *First Order Unification by Structural Recursion*[22]

```
-------- MAYBE MONAD --------------------------------
return x = Just x
fail s = Nothing
bind Nothing g = Nothing
bind (Just x) g = g x

liftM  :: (a -> b)      -> Maybe a -> Maybe b
liftM2 :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
liftM  f ma    = do a <- ma;          return (f a)
liftM2 f ma mb = do a <- ma; b <- mb; return (f a b)

------- FINITE (VARIABLE) SETS & TERMS --------------
data Fin n
   = ex m. Fz         where n = S m
   | ex m. Fs (Fin m) where n = S m

data Term n = Var(Fin n) | Leaf | Fork(Term n)(Term n)

------- SUBSTITUTIONS -------------------------------
rename2sub :: (Fin m -> Fin n) -> Fin m -> Term n
rename2sub f i = Var (f i)

subst :: (Fin m -> Term n) -> Term m -> Term n
subst sub (Var x)    = sub x
subst sub Leaf       = Leaf
subst sub (Fork s t) = Fork(subst sub s)(subst sub t)

compose :: (Fin m -> Term n) -> (Fin l -> Term m)
                             -> (Fin l -> Term n)
compose f g i = subst f (g i)

------- OCCURS CHECK --------------------------------
thick :: Nat' n -> Fin (S n) -> Fin (S n) -> Maybe (Fin n)
thick n         (Fz)   (Fz)   = Nothing
thick n         (Fz)   (Fs y) = Just y
thick (S n) (Fs x) (Fz)   = Just Fz
thick (S n) (Fs x) (Fs y) = liftM Fs (thick n x y)

chk :: Nat' n -> Fin (S n) -> Term (S n) -> Maybe (Term n)
chk n x (Var y)    = liftM Var (thick n x y)
chk n x (Leaf)     = Just Leaf
chk n x (Fork s t) = liftM2 Fork(chk n x s)(chk n x t)

for :: Nat' n -> Term n -> Fin (S n) -> Fin (S n) -> Term n
for n t' x y = case thick n x y of
```

```
                Just y' -> Var y'
                Nothing -> t'

-----------------------------------------------------
-- substitution lists

data AList m n
 =         Anil   where m=n
 | ex m'. Asnoc(AList m' n)(Term m')(Fin (S m')) where m = S m'

sub :: Nat' m -> AList m n -> (Fin m -> Term n)
sub _     (Anil)        = Var
sub (S n) (Asnoc s t x) = compose(sub n s)(for n t x)

cat :: AList m n -> AList l m -> AList l n
cat xs (Anil)         = xs
cat xs (Asnoc ys t x) = Asnoc (cat xs ys) t x

data SomeSub m = ex n. SomeSub (Nat' n) (AList m n)

asnoc :: SomeSub m -> Term m -> Fin (S m) -> SomeSub (S m)
asnoc (SomeSub m s) t x = SomeSub m (Asnoc s t x)

-----------------------------------------------------
-- unification

mgu :: Nat' m -> Term m -> Term m -> Maybe (SomeSub m)
mgu m s t = amgu m s t (SomeSub m Anil)

amgu :: Nat' m -> Term m -> Term m -> SomeSub m -> Maybe (SomeSub m)
amgu m (Leaf)       (Leaf)       acc = Just acc
amgu m (Leaf)       (Fork s t)   acc = Nothing
amgu m (Fork s t)   (Leaf)       acc = Nothing
amgu m (Fork s1 t1) (Fork s2 t2) acc =
  do acc <- amgu m s1 t1 acc; amgu m s2 t2 acc
amgu m (Var x) (Var y) (SomeSub _ Anil) = Just (flexFlex m x y)
amgu m (Var x) t       (SomeSub _ Anil) = flexRigid m x t
amgu m s       (Var x) (SomeSub _ Anil) = flexRigid m x s
amgu (S m) s t (SomeSub n (Asnoc sub r z)) =
  do sub <- amgu m (subst (for m r z) s)
                   (subst (for m r z) t)
                   (SomeSub n sub)
     return (asnoc sub r z)

flexFlex :: Nat' m -> Fin m -> Fin m -> SomeSub m
flexFlex (S m) x y = case thick m x y of
               Just y' -> SomeSub m (Asnoc Anil (Var y') x)
               Nothing -> SomeSub (S m) Anil
```

```
flexRigid :: Nat' m -> Fin m -> Term m -> Maybe (SomeSub m)
flexRigid (S m) x t = do
               t' <- chk m x t
               Just (SomeSub m (Asnoc Anil t' x))
```