# Languages of the Future

Tim Sheard*
work performed at
OGI School of Science & Engineering, Oregon Health & Science University
author's current affliation
Portland State University
sheard@cs.pdx.edu

## Abstract

This paper explores a new point in the design space of formal reasoning systems - part programming language, part logical framework. The system is built on a programming language where the user expresses equality constraints between types and the type checker then enforces these constraints. This simple extension to the type system allows the programmer to describe properties of his program in the types of *witness* objects which can be thought of as concrete evidence that the program has the property desired. These techniques and two other rich typing mechanisms, rank-N polymorphism and extensible kinds, create a powerful new programming idiom for writing programs whose types enforce semantic properties.

A language with these features is *both* a practical programming language *and* a logic. This marriage between two previously separate entities increases the probability that users will apply formal methods to their programming designs. This kind of synthesis creates the foundations for the languages of the future.

## 1 Introduction

Today's languages have two glaring problems. These problems are the *semantic gap* and the *temporal gap*.

- **The Semantic Gap.** 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. Languages which can narrow this gap are sorely needed
- **The Temporal Gap.** Systems are configured with new knowledge at many different times compile-time, link-time, run-time. We express this knowledge with an amazing mish mash of systems and ad-hoc machinery – *autoconf*, *make*, *macros*, *configuration files*, *scripting languages*, and *embedded interpreters*. Such approaches create huge impedance mismatches, and are often hard to understand, but failure to address the temporal gap often leads to unacceptable performance bottlenecks. Languages which could handle all these issues in a uniform manner are sorely needed.

If we are ever to build systems that we can trust on a large scale, we must develop programming languages that narrow these gaps. The programming languages of the future will have the following properties.

- They will allow programmers to describe and reason about semantic properties of programs from within the programming language itself, mainly by using powerful type systems. But, the languages will be *designed* to interoperate with other external reasoning or testing systems as well.
- The languages will be within reach of the majority of programmers. Using the reasoning capability of the language will not be too time consuming, nor will the learning curve for learning how to use such features be too high.
- They will be practical, supporting all the capabilities we now expect in a programming language. But, they may organize these capabilities in new ways that better control potentially unsafe features. They will use static analyses to separate powerful but risky features from the rest of the program, and will clearly mark the boundaries between the two. They will spell out the obligations required to control the risk, and support and track how these obligations can be met.
- They will be efficiently implementable, but perhaps in new and novel ways. Rather than relying on a strict compile-time/run-time distinction to perform a single heroic optimization, they will provide a flexible hierarchy of *stages* from within the programming language. Staging will deal uniformly with notions of compile-time, link-time, run-time, and run-time code generation. This will allow the computation system to take advantage of important contextual information no matter when it becomes available. The staging separation will also track semantic properties across stages. It will be possible to know that a stage $i$ program always builds a stage $i+1$ program with some known property $p$.

In this paper we explore a new point in the design space of programming languages and formal reasoning systems: the development of the language Ωmega. Ωmega is *both* a practical programming language *and* a logic. Ωmega also addresses the temporal gap by the use of explicit staging.

The sometimes irreconcilable goals of being both a programming language and a logic, are made possible by embedding the Ωmega logic in a type system based on *equality qualified* types[9]. This

*Supported by NSF CCR-0098126.


*OOPSLA 2004* October 24-28, 2004, Vancouver, B.C., Cananda.

is done by making a very small (backward compatible) change to the notion of Algebraic Datatypes as supported in languages such as O'Caml and Haskell. We call these new types *Generalized Algebraic Datatypes* (GADTs). They support the construction, maintenance, and propagation of semantic properties of programs using powerful old ideas about types (the Curry-Howard Isomorphism) in surprisingly easy to understand new ways.

The semantic gap between formal tools and implementation languages prevents the application of formal methods to software design on all but the most important applications. It is necessary to narrow this gap so that these techniques can be applied to a much broader set of applications. One way to reason about program properties is to use Curry-Howard isomorphism. This states that programs are proofs of the properties expressed in their types. It is often hard for the programmers to conceptualize how they can put this abstraction to work. GADT's make this isomorphism concrete – proofs are real objects that programmers can build and manipulate without leaving their own programming language. Such proofs can express important semantic properties of their programs. We believe that this increases by orders of magnitude the probability that programmers will construct programs that they actually reason about, and this will make measurable differences in the quality of the code produced. It is not that programmers cannot reason about their programs; rather, it is that they find the barriers to entry so high that they would rather not.

Theorem provers and logical frameworks have many of the same goals, but we believe there are qualitative differences between them and our work. First, Ωmega is designed foremost to be a programming language. It supports practical programming features such as input/output and side-effects, but uses its type system to cleanly separate these potentially dangerous features from the core language of the logic.

Second, Ωmega uses a single computational model for both its logic and its programming. It uses a strict functional model with monads [46, 45, 44] to separate effects from pure computation. This model suffices to describe both programs and properties. Contrast this with logical frameworks where programs are purely functional and the logic employs prolog style back chaining (Elf), or higher order pattern matching (Twelf). A similar dichotomy arises in LCF style theorem provers such as Coq. In such systems, programs must be extracted from proofs, which are themselves constructed in highly unnatural ways using tactics and proof combinators. We believe that this two model paradigm is unnatural, and that the single model of Ωmega is easier to learn and use by ordinary programmers. We discuss this in more detail in Section 7.

Third, Ωmega tries to maintain a strict distinction between values, types, and kinds. Since these classifications usually indicate a phase distinction between compile-time and run-time, understood by most programmers, we believe this makes it easier for programmers to use Ωmega than systems which do not make these distinctions.

Fourth, Ωmega incorporates staging annotations. In Coq and other related systems, proofs *correspond* to programs. More efficient implementations can often be extracted from proofs by a form of type erasure. In Ωmega proofs *are* programs (with equality qualified types), and unlike Coq[41], and Isabelle[27] where type erasure is fixed and inflexible, type erasure in Ωmega is implemented by the use of explicit staging. The conjunction of staging and logical systems provides a powerful new tool. By using staging, extraction of efficient programs from proofs is under the control of the programmer, and can be targeted at *any* object-language. Staging can also be used to perform specialization and partial evaluation.

Fifth, GADTs allow Ωmega to reflect representations of its types into the value world. These reflections exactly mirror the type world, but are first class values. Each reflected value has a unique (singleton) type. The reflection mechanism ensures that only sound computations can be performed on reflected values. A reflected value can be reified back into the type world, and the system can rely on its soundness. This allows programmers to write programs that perform type-level computations in the value world for type checking problems that are too hard for the type checker (with its limited computation mechanism - usually based upon unification) to figure out. This allows the programmer to construct programs that effortlessly slip between static and dynamic property checking, and allow the programmer to capture a wide variety of properties, both logical (semantic), and physical (resource usage) in the types of programs.

While formal reasoning systems are very good at what they do, they were not designed to be programming languages. These tools are too expressive. They trade usability for expressiveness. There is something to be gained by being selective, giving up some expressiveness in order to maintain the pragmatic properties of a system. Powerful tools are very useful and have their place in system design, but there is a missing point in the continuum of tools between practical and formal, and Ωmega is designed to fill this gap. By doing so wisely, much is to be gained, in terms of ease of use, a more gradual learning curve, and increased interoperability with other systems.

## 2 How Types Capture Properties

An important role of type systems in programming languages is to guarantee the property that programs do not use data (including functions) in inappropriate ways. But types can also be used to ensure much more sophisticated properties. There are many examples of this in the literature. Types have been used to ensure the safety of low level code such as Java Byte Code[36, 5] or typed assembly language[23, 24]. These systems use types to model the shape of the stack or register bank to ensure that low level code sequences are used properly (e.g. no stack underflow). Types have also been used to model information flow[31, 43, 25] to ensure security properties of systems. Types have been used to track resource control, such as the possibility of non-termination [20], or to place upper bounds on the time consumed by a computation[11, 42]. Types have been used as a means of removing dynamic error tests – for example, to enforce data structure invariants[51] (such as ensuring red-black trees are well formed) or to make code more efficient by removing unnecessary run-time array bounds checks[52]. Finally, types have been used to track access control, which allows removing (or minimizing) stack inspection overhead as a means of managing capabilities[47, 6].

We have modelled the important ideas in each of these systems using Ωmega. While the properties of these systems could have been modelled by a formal system such as a logical framework (Twelf[30]) or theorem prover (Coq[41] or Isabelle[27]), the properties would be a meta-logical property of the program and external to the implementation. In Ωmega they are a property of the implementation, which is enforced by Ωmega's type checker. Rather than model an existing application in a formal system, or use a formal system to build a model of an as-yet- unimplemented application and then derive or generate an implementation from this model, in Ωmega, we implement and reason in a single paradigm. We illustrate this with simple examples later in the paper.

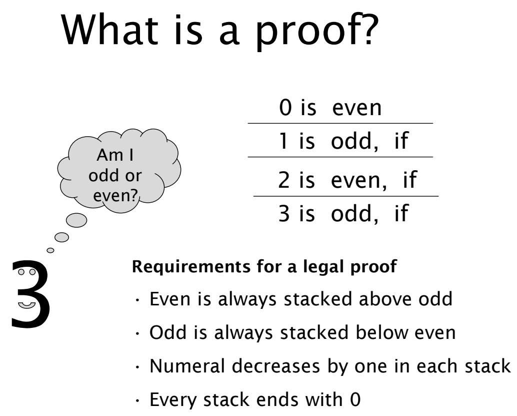


**Figure 1. Proofs are logical arguments which use well-formed reasoning.**

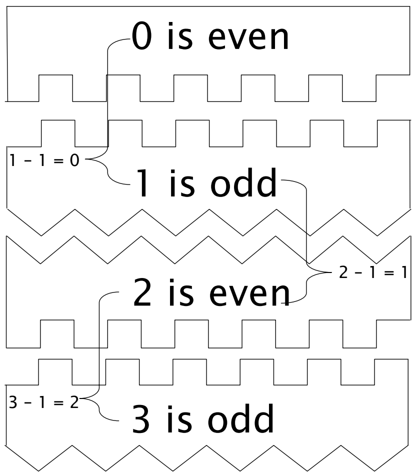


**Figure 2. Proofs are data that fit together in precise ways.**

We have coined a new slogan for the process of designing reliable systems in this way: *Mostly types – just a little theorem proving.* We argue that many properties that can be modeled in a theorem prover or logical framework, can also be modelled more straightforwardly in a programming language whose type system has been strengthened in just a few simple ways. This allows properties of systems to be modelled in a more light-weight manner, yet still be completely formal. Generalizing the notion of algebraic datatypes, adding type checking for rank-N polymorphism, extensible kinds, and staging support makes this light-weight formality possible. Programmers already familiar with the use of a theorem prover or logical framework will find that many of the powerful ideas behind these tools have been moved to a practical programming language and have become more widely applicable. Thus, we can save the power and frustration of using a theorem prover for when we really need it.

## 3 What is a proof?

What is a proof? Consider the question asked in Figure 1. How can we tell if the integer 3 is odd or even? Given the "fact" that 0 is even, we can reason that 1 is odd, and then that 2 is even, and finally that 3 is odd. Such arguments follow strict rules about how they are constructed. The notion that we can "construct" proofs leads us to a powerful analogy – proofs are just data structures that are constructed in precise, well-formed ways.

We illustrate this in Figure 2. Here each step is a "lego" block with different shape "snaps". Odd steps have triangular teeth on the bottom, and square teeth on top. Even steps have square teeth on the bottom, and triangular teeth on top. Like a jigsaw puzzle, this prevents steps from being "snapped" together in bad ways. In addition to the protection provided by the puzzle piece protocol, the invariant that the numeral being tested decreases by one as we ascend the stack must also be ensured. The top of the proof stack is a fact. Notice how it has no teeth on top. It is from simple "facts" (like 0 is even) that proofs get started.

We capture this puzzle like behavior by using types. We provide an abstract interface to the creation of well-formed odd/even proofs. Let `Z` be the fact that 0 is even. And let `E` construct a proof that `m+1` is even when snapped underneath a proof that `m` is odd, and finally `O` creates odd proofs by snapping them onto even proofs. This is captured by the interface below:

```
Z:: Even 0
E:: Odd m -> Even (m+1)
O:: Even m -> Odd (m+1)
```

So to construct a proof that 3 is odd we simply snap together the right pieces `O(E (O Z)) :: Odd (1+1+1+0)`. Note how the types of the functions that comprise the interface to the odd/even proof abstraction prevent us from constructing proofs of invalid facts. It is this property of well defined proof abstractions that make them so valuable. How will we go about designing such abstractions? We can build them by generalizing the well understood abstraction of algebraic datatypes.

## 4 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:

```
-- 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
```

Each definition introduces a new structured datatype (`Color`, `Address`, `Person`), and a set of constructors. Some are constants (`Red`, `Blue`, `Green`), others are constructor functions (`MakeAdress`, `Teacher`, `Student`). Types are used to prevent the construction of ill-formed data. This is done by giving each constructor a type which ensures that it can only be "snapped" together in well formed ways. For example:

```
Red :: Color
Blue :: Color
Green :: Color
MakeAddress :: Number -> Sreet -> Town -> Address
Teacher :: [Class] -> Person
Student :: Major -> Person
```

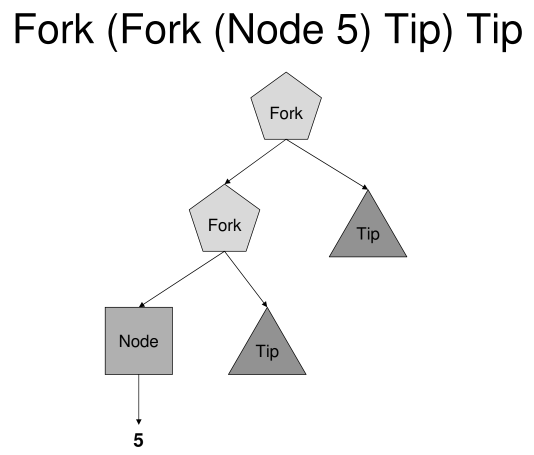


**Figure 3. Constructor functions provide an abstract interface to heap data. Every shape represents an internal node in the tree constructed by a different constructor.**

Two valuable extensions to this mechanism are the ability to parameterize a `data` definition to construct polymorphic data structures, and the ability to construct recursive data structures. For example:

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

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 constants (`Tip`) are given polymorphic types. This enables the same constructor functions to construct trees of many different types.

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

Values of the new datatype are constructed solely by the application of these constructors. For example, the tree illustrated in Figure 3 is constructed by the following expression `(Fork (Fork (Node 5) Tip) Tip)`. Constructors abstract away from explicit heap allocation and the use of pointers. These mechanisms are hidden from the programmer and are built into the constructor function implementations that are automatically generated by the compiler.

Data stored in algebraic datatypes is accessed by the use of pattern matching, which allows abstract, high-level (yet still efficient) inspection of the values stored inside the structure. Pattern matching, like constructor functions, abstract pointers and heap allocation, and give a declarative feel to data access. We illustrate this be defining the `sum_of` function which adds up all the values stored in tree whose "slots" are filled with integers.

```
sum_of :: Tree Int -> Int
sum_of Tip = 0
sum_of (Node x) = x
sum_of (Fork m n) = sum_of m + sum_of n
```

**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*.

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

By generalizing algebraic datatypes so they are no longer subject to this restriction we can construct proof-like objects, and the type system of the language will enforce that only well-formed proofs are ever constructed. We can encode the odd/even abstraction as a three constructors (`Z`,`E`, and `O`) from two different types (`Even` and `Odd`) which have non-polymorphic ranges.

```
Z::  Even 0
E::  Odd m -> Even (m+1)

O::  Even m -> Odd (m+1)
```

The only complication with this approach is *what to make of the type parameters to* `Even` *and* `Odd`. These appear to be integers rather than types.

## 5 New Kinds

Kinds are similar to types in that, while types classify values, kinds classify types. We indicate this by the *classifies* relation (`::`). For example: `5 :: 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` has kind `Nat`, and `S` has kind `Nat ~> Nat`. The type `S` is a type constructor, so it has a higher-order kind. We indicate this using the classifies relation as follows:

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

The classification `Nat::*1` indicates that `Nat` is at the same "level" as `*0` — they are both classified by `*1`. 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 4.

From this discussion we see that the integer arguments to the type constructors `Even` and `Odd` are really the natural numbers implemented at the type level by the kind declaration for `Nat`.

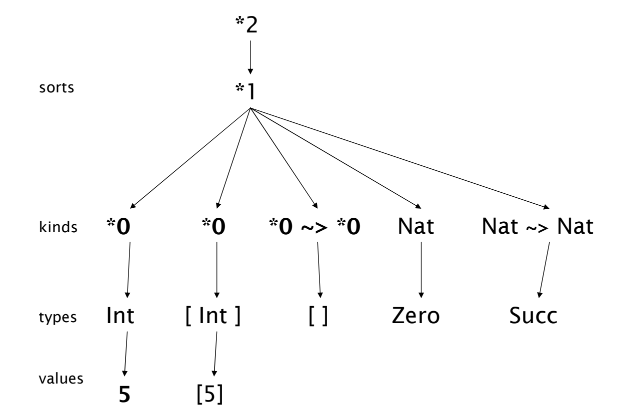


**Figure 4. The classification hierarchy. An arrow from $a$ to `b` means $b$::$a$. Note how only values are classified by types that are classified by `*0`, and how type constructors (like `[]` and `S`) have higher order kinds.**

```
Z::Even 0                 Z::Even Z
E::Odd m -> Even (m+1)    E::Odd m -> Even (S m)

O::Even m -> Odd (m+1)    O::Even m -> Odd (S m)

O(E(O Z))::Odd 3          O(E(O Z))::Odd(S(S(S Z)))
```

In the rest of this paper when we use Arabic numerals at the type level, we are using them simply as a notational convenience. For example we may write: `(Odd 3)` instead of `(Odd (S(S(S Z))))`.

## 6 Equality Qualified Types

Removing the range restriction from algebraic datatypes can be explained in terms of *equality qualified types*. We provide two different ways to introduce a new type using a `data` declaration. Both support the definition of algebraic datatypes without the range restriction. The first is by the use of *explicit* constructor function declaration, and the second is by the use of *equality restrictions*. We illustrate both forms below for the Even/Odd proofs.

```
-- explicit constructor function declaration
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)

-- equivalent declaration using equality restrictions
data Even n
  = Z where n=Z
  | exists m . E (Odd m) where n=S m

data Odd n =
  exists m . O (Even m) where n=S m
```

The explicit declaration form follows from our earlier discussion. Just list 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 into the second form that explains the range restriction in terms of the well studied and understood idea of qualified types.

In the equality qualified syntax we use the original syntax for `data` declaration, but now allow every constructor to be followed by a where clauses which lists a set of type equalities that must hold for that constructor.

Constructor functions (`Z`, `E`, and `O`) construct elements of `data` types `Even` and `Odd`. The type of a constructor function is described in the `data` declaration. For example, the clause in the `Even` declaration: `exists m.E (Odd m) where n=S m` introduces the `E` constructor function. Without the `where` qualification, the constructor function `E` would have type `(E::Odd m -> Even n)`. *Equality Qualification* (indicated by the `where` in the clauses for `Z`, `E`, and `O`) and *existential quantification* (indicated by `exists` in the clauses for `E`, and `O`) help encode semantic properties. The `where` *qualifies* `E`'s type, in effect saying `(E:: Odd m -> Even n)` *provided* `n=S m`. We capture this formally by writing `E::(forall n m.(n=S m)=> Odd m -> Even n`. The equations behind the *fat arrow* (`=>`) are equality qualifications. Since `n` is a universally quantified type variable, there is only one way to *solve* the qualification `n=S m` (by making `n` equal to `S m`). Because of this unique solution, `E` also has the type `(forall m.Odd m -> Even (S m))`. This type guarantees that `Cons` can only be applied in contexts where `n=S m`. This is the same type we would have written using the explicit constructor declaration method.

The existential quantification of the type variable `m` names the intermediate predecessor of the sub-proof of `E`, which if not introduced in this way would appear as an unbound type variable. Every declaration in the *explicit form* can be translated into one in the *equality qualified form*. So the first can be defined as a shorthand for the second. The first form is more concise and sometimes easier to use, but the second form can be explained semantically in terms of the well known and well studied idea of qualified types. Equality qualified types are a relatively new feature in the world of programming languages, and were only recently introduced by Hinze and Cheney[9]. Ωmega is based upon this work.

## 7 An Introduction to Ωmega

In this section we introduce Ωmega. We use a simple application which has a semantic invariant captured by the type system of Ωmega. The example is sequences of elements with the semantic property that the length of the sequence is encoded in its type. For example the sequence $[a_1, a_2, a_3]$ has type $(Seq\ a\ 3)$, and the type of the *Cons* operator that adds an element to the front of a sequence would be $a \rightarrow Seq\ a\ n \rightarrow Seq\ a\ (n+1)$. A map function 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.

The type of the append operator would be $Seq\ a\ n \rightarrow Seq\ a\ m \rightarrow Seq\ a\ (n+m)$. In order to type such functions it is necessary to do arithmetic at the type level. In Figure 5 is an Ωmega program that captures this specification. The code introduces a new type (`Seq`), two new functions (`map` and `app`), and a new type function (`plus`). Equality qualified types can encode powerful static invariants about data. For example the type of `map` provides a proof that `map` returns a list of the same length as its input list.

**Tracking equality constraints.** 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*. Obli-

```
data Seq a n
  = Nil where n = Z
  | exists m . Cons a (Seq a m) where n = S m

-- equivalently in explicit constructor format
-- data Seq:: *0 ~> Nat ~> *0 where
--   Nil::Seq a Z
--   Cons:: a -> Seq a m -> Seq a (S m)

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

{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 p xs ys)
```

**Figure 5. An Ωmega encoding of lists whose types record their lengths.**

gations are generated by the type-checker when (a) the program constructs data-values with constructors that contain equality constraints; or (b) an explicit type signature in a definition is encountered; or (c) when type functions are used (see **Type functions in Ωmega** below).

We will explain how these are used by considering the functions and types defined in Figure 5. There we provide the declaration for `Seq` in both formats. The explicit format is more direct for the programmer, but the equality qualified format is easier to use when explaining the typing rules.

Consider type-checking the expression `Nil`. Using the equality qualified form, the constructor `Nil` is assigned the type `forall a n.n=Z => Seq a n`, thus `Nil::Seq a n` but also generates the obligation `n=Z`. Since `Nil` is polymorphic in `a` and `n`, the type variable `n` can be instantiated to `Z`. Instantiating `n` to `Z` makes the equality constraint obligation Z=Z, which can be trivially discharged by the type checker. After discharging this constraint the type of `Nil` is exactly the type provide in the explicit form.

*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 was defined using a `where` clause, the type equalities in the `where` clause are added to the current set of assumptions in the scope of the pattern. These assumptions can be used to discharge obligations. The equality qualified form is especially helpful in explaining the type checking of pattern matching.

For example, consider type checking the definition of `map`. Recall `map::(a->b)->Seq a n->Seq b n`. In the first equation, the pattern `Nil` of type `Seq a n` introduces the assumption `n=Z` because of `Nil`'s qualified type `forall a n.n=Z => Seq a n`. On the right-hand-side, the use of the `Nil` constructor generates the obligation that `n=Z`. The assumption generated by the pattern on the left is used to discharge obligations incurred on the right.

Now, consider the second equation `map f (Cons x xs) = Cons (f x) (map f xs)`. We know `map`'s type, so the second pattern `(Cons x xs)` must have type `(Seq a n)`. This implies `x::a` and `xs::Seq a m` provided `n = S m`. This equation is added to the list of assumptions when typing the right-hand-side of the second equation: `Cons (f x) (map f xs)`. This right-hand-side should have type (`Seq b n`) (the range of `map`'s type). Since `xs::Seq a m` we can compute `(map f xs)::Seq b m`, Since `x::a` we can compute `(f x)::b`, thus it appears `(Cons (f x) (map f xs)):: Seq b (S m)`. But given the equality constraint `n=S m` the right hand side has type `Seq b n` as expected.

**Type functions in Ωmega.** The append function should have type `Seq a n -> Seq a m -> Seq a (n+m)`. In order to type such functions it is necessary to do arithmetic at the type level. This means defining functions over types. In Ωmega we do this by writing equations over types. We surround type-function application by braces (e.g. `{plus x x y}` to distinguish it from type-constructor application (e.g. `List Int`). We also define the type-function `plus` and the value-function `append` of static sequences. The `app` can be typed using reasoning similar to the reasoning used when type-checking the `map` function. An additional complication is the use of the type function `plus` to rewrite type-level terms into normal form when discharging obligations. To ensure that type-checking is tractable we require type-functions to be exhaustive, and confluent.

**An observation about the type parameters of Ωmega type constructors.** The second parameter of the type constructor `Seq` plays a qualitatively different role than type parameters in other data structures. 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, `Tree` s 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 `n` in `(Seq a n)`. Here there are no sub components of type `n`. Instead, the parameter `n` is used to stand for an abstract property (the statically known length of the list). The `where` qualifications restrict the legal instances of `n`. Type parameters used in this way are sometimes called index types[50, 52], and will play an important role in what follows.

# 8 Comparing Formal Reasoning Systems

Ωmega uses a single computational model (strict functional rewriting) for both its logic and its programming. We claim that this is easier for programmers to learn and to use than the dual computational model found in some other systems. To illustrate this point we compare Ωmega to two other systems with similar goals: Coq and Twelf.

In Figure 6 we see a similar encoding of natural numbers at the type level, and an encoding of sequences with encoded lengths. In Coq the definition of `plus` is defined by structural induction over `nat` types, but the definition of append is given by a series of commands (`Introduction`, `EApply`, `Simpl` etc.) that guide the Coq theorem prover to construct a proof object with the given type. The append function is then extracted (not shown) from this proof object. In the Twelf encoding the plus function and the append function are encoded as logic programs.

The big advantage of the Ωmega approach is that the program *is* the logic. There is no translation between programming notation to some external reasoning tool. Second, there is no need to switch gears when reasoning about the system. Rather than thinking in terms of our implementation programming language, in Coq we must think in terms of proof tactics, and in Twelf (given that the vast majority of programs are not written in Prolog) we must think in terms of logic programs.

**Coq encoding**

```
Inductive nat : Set := Z : nat | S : nat -> nat.

Definition plus : nat->nat->nat :=
Fix plus
  {plus [n:nat] : nat->nat :=
     [m:nat]Cases n of
             Z => m
           | (S p) => (S (plus p m))
           end}.
Inductive  Seq [A:Set] : nat -> Set :=
 Nil : (Seq A Z)
|Cons : (n:nat; x:A; xs : (Seq A n))(Seq A (S n)).

Definition app [A:Set] : (m,n:nat)
  (Seq A m) -> (Seq A n) -> (Seq A (plus m n)).
Intros.  Induction H. EApply H0. Simpl.
Apply (Cons A (plus n0 n) x HrecH). Defined.
```

**Twelf encoding**

```
elem : type.
e1 : elem.

nat : type.
z : nat.
s : nat -> nat.

plus : nat -> nat -> nat -> type.
base : plus z Y Y.
step : plus (s X) Y (s Z)
       <- plus X Y Z.

seq : nat -> type.
nil  : (seq z).
cons : elem -> (seq A) -> (seq (s A)).

app : (plus A B C) -> (seq A) ->
        (seq B) -> (seq C) -> type.
app_1 : app base nil X X.
app_2 : app (step P) (cons X XS) YS (cons X ZS)
          <- app P XS YS ZS.
```

**Figure 6. Coq and Twelf programs for comparison to Ωmega.**

# 9 Witness Objects

GADT's can model relations between types, other than equality, by defining witness types. A witness is a value with a parameterized type whose existence witnesses a relationship between its 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. 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 w x y
  = Base where w=Z , x=y
  | exists m n . Step (Sum m x n)
             where w=S m, y=S n
```

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 length.

# 10 Example: Type-Safety & Static Scoping

We now turn to a richer example: modelling a simple imperative *While language* with semantic properties of static scoping and type safety[26, 28]. Every while-program represented as an Ωmega data structure is a proof that every variable in that program refers to some binding site (static scoping), and that the program is also well typed. The power of Ωmega is that modelling these static semantic properties requires approximately the same amount of time and intellectual effort one uses to model context free syntactic properties using other means. In addition any Ωmega program that manipulates a while-program data structure, is guaranteed to maintain these properties. Ωmega programs that do not maintain the scoping and typing are statically determined to be ill-typed and are thus rejected.

In Figure 7 we introduce data structures to represent the While language. The `data` declarations introduce three new parameterized types `V`, `Exp` and `Com` for variables, expressions, and commands. These are *type constructors*, and an actual element of the new types will have types like `(V (Int,Bool) Bool)`, `(Exp (Int,Bool) Int)`, or `(Com (Int,Bool))`. We interpret `(Exp s t)` as an expression with type `t` in store `s`. The type of a store captures the types of the variables currently in scope. A similar interpretation is given to variables `(V s t)`. Commands don't have result types, but are interpreted in the store `(Com s)`. The declarations also introduce *constructor functions* `Z`, `S`, `IntC`, `BoolC`, etc. whose types are given as comments in Figure 7. Readers familiar with type systems will notice that the types of the constructor functions look a lot like typing judgments. We have used the equality constrained types to encode and reason about these inference rules *in the programming language*.

**Manipulating while-programs.** In Figure 8 a small interpreter for the While-language is given. Expressions are interpreted by the function `eval::Exp s t -> s -> t`. The function `eval`, given a term of type `(Exp s t)` producers a function from `s` to `t`. `eval` gives meaning to the term. Given `store::s`, a data structure which stores values for the expression's variables, then we can produce the value of the expression by applying `eval` to the expression and `store`. The type of the store models the types of the reachable variables in the object-program. Variables are integers (using a de Bruijn-like notation), and stores are nested pairs. The nested pairs have the following shape `(0,(1,(2, ...)))` where the `0`, `1`, and `2` indicate the index of the variable that "reaches" to the corresponding location in the nested pair. Because of the natural number-like definition of the type `(V s t)` we see that `(Var Z)` models the variable with index 0, `(Var (S Z))` models the variable with index 1, and `(Var (S (S Z)))` models the variable with index 2, etc. Thus if the type of the store is `(Int,(Bool,a))` then variable with index 0 has type `Int` and the variable with index 1 has type `Bool`.

```
data V s t
  = exists m . Z where s = (t,m)               -- x0                     V (t,m) t
  | exists m x . S (V m t) where s = (x,m)      -- xn                     V m t -> V (x,m) t

data Exp s t
  = IntC Int where t = Int                     -- 5                      Int -> Exp s Int
  | BoolC Bool where t = Bool                  -- True                   Bool -> Exp s Bool
  | Plus (Exp s Int) (Exp s Int) where t = Int   -- x + 3                  Exp s Int -> Exp s Int -> Exp s Int
  | Lteq (Exp s Int) (Exp s Int) where t = Bool  -- x <= 3                 Exp s Int -> Exp s Int -> Exp s Bool
  | Var (V s t)                                -- x                      V s t -> Exp s t

data Com s
  = exists t . Set (V s t) (Exp s t)           -- x := e                 V s t -> Exp s t -> Com s
  | Seq (Com s) (Com s)                        -- { s1; s2; }            Com s -> Com s -> Com s
  | If (Exp s Bool) (Com s) (Com s)            -- if e then x else y     Exp s Bool -> Com s -> Com s -> Com s
  | While (Exp s Bool) (Com s)                 -- while e do s           Exp s Bool -> Com s -> Com s
  | exists t . Declare (Exp s t) (Com (t,s))   -- { int x = 5; s }       Exp s t -> Com (t,s) -> Com s
```

**Figure 7. Typed, statically scoped, abstract syntax for the *While language*. The left hand column illustrates the Ωmega code that introduces data structures that represent the new object-language, and the middle column (following the comment token `--`) suggests a concrete syntax that the abstract syntax represents. The right hand column gives the type of the constructor function as described in the text below.**

Under this interpretation it is easy to understand the functions `update`, `eval`, and `exec`. Consider: `(update (S Z) False (12,(True,0))`. This should return a new nested pair where the location of the index ((S Z) which is 1) has been replaced by `False` giving `(12,(False,0))`. This proceeds by `(update (S Z) False (12,(True,0))` ⟶ `(12,update Z False (True,0))` ⟶ `(12,(False,0))`. Note how pattern matching chooses the correct clause to execute.

In a similar fashion the `eval` function when applied to a variable `(Var` $i$`)` "extracts" the $i^{th}$ value from a nested pair. `(eval (Var (S Z)) (12,(True,0))` ⟶ `(eval (Var Z) (True,0))` ⟶ `True`. The execution function for commands `(exec::Com s -> s -> s)` is a store transformer, transforming the store according to the assignments executed in the command.

Since the properties of the object-programs are captured in their types, respecting these types ensures that the meta-programs maintain the properties of the object programs. For example given that the meta-level variables `x` and `sum` are defined by `sum = Z` (the variable with index 0) and `x = S Z` (the variable with index 1) , observe:

```
prog::Com (Int,(Int,a))
prog = Seq (Set sum (Int 0))
           (Seq (Set x (Int 1))
                (While (Lteq (Var x) (Int 5))
                  (Seq (Set sum (Plus (Var sum)(Var x)))
                       (Set x (Plus (Var x)(Int 1))))))
-- { sum = 0;
--   x = 1;
--   while (x <= 5)
--    { sum = sum + x;
--      x = x + 1; } }
```

The term `prog` has a meta-level type that states that it is well-typed at the object-level, only if the object-level store has an `Int` at indexes 0 and 1. If one tries to create an ill-typed object-level term a static type checking error occurs. For example consider the command `(if x then x := 0 else x := 1)` where the variable `x` needs to be typed as both an `Int` and a `Bool`.

```
badIf = If (Var x) (Set x (IntC 0)) (Set x (IntC 1))

In the expression: Set x (IntC 0)
the result type: Com (a,(Int,b))
was not what was expected: Com (a,(Bool,c))
   Int   does not unify with    Bool
```

**Possible Enhancements.** Enhancing object-languages with type safety can be accomplished in two dimensions: a richer language *or* a richer type system. We have done both. We have also modelled several different styles of language semantics other than the big-step style given for the While-language. One of our most interesting semantics consisted of a typed small step semantics. Since this small step semantics is typed, it amounts to a machine checked subject reduction proof[49].

## 11 Staging and Efficient Implementations

Staged programs proceed in stages. Each stage "writes" a program that is executed in the next stage. Practical examples of staged systems include run-time code generation, dynamic compilation, and program generators. Staging is the key technology that supports efficient implementations without interpretive overhead.

Staging is an programming language interface to code generation. We have built two large sophisticated systems that implement staging. MetaML[33], a system with run-time code generation, and Template Haskell[34], a system with compile-time code generation (think macros, quasi-quotes, and type safety). In Figure 11 we use the staging mechanism of Ωmega. It consists of the annotations brackets (`[| _ |]`) and escape (`$( _ )`). Brackets introduce a new code template and specify that the expression inside the brackets should be generated as a program for the next stage. Within brackets, escape specifies a hole within a template. The escaped expression is executed (resulting in a piece of code), and the resultant code is spliced into that hole. Staging makes a perfect complement to equality qualified types for two reasons. First, many applications can be encoded as domain specific languages (DSLs). Such languages can be given meaning by writing a simple interpreter (like the `eval` and `exec` functions from Figure 8). Staging an interpreters produces an efficient compiler as the interpretive overhead or traversing the abstract syntax is removed. This is illustrated in the top of Figure 11 for the `Exp` fragment of the while-language.

Second, staging can implement program extraction from proofs. Both Coq and to some extent Isabelle support program extraction from proofs. These features are limited because the target languages are hardwired and the generated programs must conform to the type system of the target language. This often requires discarding important information about the source program, or run time passing of static information. If we consider the `app` function from

```
update::(V s t) -> t -> s -> s
update Z n (x,y) = (n,y)
update (S v) n (x,y) = (x,update v n y)

eval::Exp s t -> s -> t
eval (IntC n) s = n
eval (BoolC b) s = b
eval (Plus x y) s = (eval x s) + (eval y s)
eval (Lteq x y) s = (eval x s) <= (eval y s)
eval (Var Z) (x,y) = x
eval (Var (S v)) (x,y) = eval (Var v) y
```

```
exec::(Com st) -> st -> st
exec (Set v e) s = update v (eval e s) s
exec (Seq x y) s = exec y (exec x s)
exec (If test x1 x2) s =
  if (eval test s) then exec x1 s else exec x2 s
exec (While test body) s = loop s
  where loop s = if (eval test s)
                 then loop (exec body s)
                 else s
exec (Declare e body) s = store
  where (_,store) = (exec body (eval e s,s))
```

**Figure 8. Interpreters for the While-language. These functions illustrate pattern matching over constructor functions, and semantics preserving meta-functions. All of `update`, `eval`, and `exec` manipulate while-programs in a way that respects their semantic properties. In fact, because all while-programs are well typed these interpreters are tagless[39], and they return values whose types correspond to the types of the while-programs.**

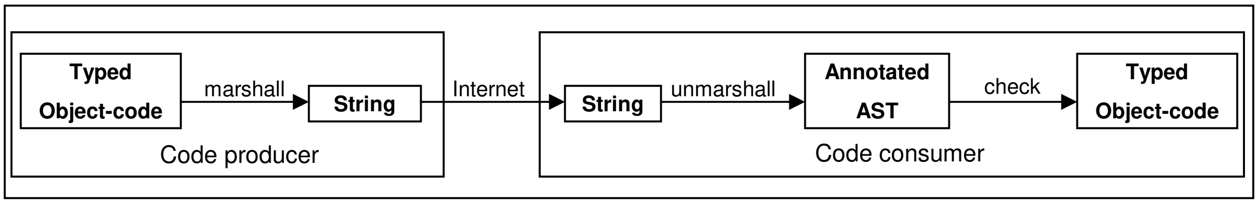


**Figure 9. Proof carrying code process**

```
data TyAst = I | B | P TyAst TyAst

data ExpAst
  = IntCA Int
  | BoolCA Bool
  | PlusA ExpAst ExpAst
  | LteqA ExpAst ExpAst
  | VarA Int TyAst

-- Equality Proofs and Type representations
data Eq a b = EqProof where a=b

data TypeR t
  = IntR where t = Int
  | BoolR where t = Bool
  | exists a b . PairR (TypeR a) (TypeR b)
                 where t = (a,b)

match::TypeR a -> TypeR b -> Maybe (Eq a b)
match IntR IntR = succeed EqProof
match BoolR BoolR = succeed EqProof
match (PairR a b) (PairR c d) =
  do { EqProof <- match a c
     ; EqProof <- match b d
     ; succeed EqProof }
match _ _ = fail "match fails"

-- Judgments for Types
data TJudgment = exists t . TJ (TypeR t)
```

```
checkT::TyAst -> TJudgment
checkT I = TJ IntR
checkT B = TJ BoolR
checkT (P x y) =
   case (checkT x,checkT y) of
     (TJ a, TJ b) -> TJ(PairR a b)

-- Judgments for Expressions
data EJudgment s = exists t . EJ (TypeR t) (Exp s t)

checkE::ExpAst -> TypeR s -> Maybe (EJudgment s)
checkE (IntCA n) sr = succeed(EJ IntR (IntC n))
checkE (BoolCA b) sr = succeed(EJ BoolR (BoolC b))
checkE (PlusA x y) sr =
  do { EJ t1 e1 <- checkE x sr
     ; EqProof <- match t1 IntR
     ; EJ t2 e2 <- checkE y sr
     ; EqProof <- match t2 IntR
     ; succeed(EJ IntR (Plus e1 e2))}
checkE (VarA 0 ty) (PairR s p) =
  do { TJ t <- succeed(checkT ty)
     ; EqProof <- match t s
     ; succeed(EJ t (Var Z))}
checkE (VarA n ty) (PairR s p) =
  do { EJ t' (Var v) <- checkE (VarA (n-1) ty) p
     ; TJ t <- succeed(checkT ty)
     ; EqProof <- match t t'
     ; succeed(EJ t' (Var (S v)))}
```

**Figure 10. Implementing the check function for the proof carrying code example.**

```
x = Z
y = S Z
e1 = Lteq (Plus (Var x)(Var y)) (Plus (Var y) (IntC 1))

data Store s = M (Code s)
  | forall a b . N (Code a) (Store b)  where s = (a,b)

test e = [| \ (x,(y,z)) ->
  $(eval2 e (N [|x|](N[|y|](M[|z|])))) |]

eval2::Exp s t -> Store s -> Code t
eval2 (IntC n) s = lift n
eval2 (BoolC b) s = lift b
eval2 (Plus x y) s = [| $(eval2 x s) + $(eval2 y s) |]
eval2 (Lteq x y) s = [| $(eval2 x s) <= $(eval2 y s) |]
eval2 (Var Z) (N a b) = a
eval2 (Var (S v)) (N a b) = eval2 (Var v) b

-- test e1 --->  [| \ (x,(y,z)) -> x + y <= y + 1 |]
```

```
app3::Sum n m p -> Code(Seq a n) ->
        Code(Seq a m) -> Code(Seq a p)
app3 Base xs ys = ys
app3 (Step p) xs ys =
  [| case $xs of Cons z zs -> Cons z $(app3 p [|zs|] ys) |]

test2::Sum u v w -> Code (Seq a u -> Seq a v -> Seq a w)
test2 witness = [| \ xs ys -> $(app3 witness [|xs|] [|ys|]) |]

-- test2 (Step (Step Base)) --->
--   [| \ xs  ys  ->
--        case xs of
--          (Cons z  zs ) ->
--              Cons z (case zs of
--                          (Cons w ws) -> Cons w ys) |]
```

**Figure 11. Illustrating Staging, removal of interpretive overhead (top), and witness removal (bottom).**

Figure 6 as a proof (because it takes a witness `Sum` type as well as two lists) staging can remove the witness in an early stage, resulting in a new piece of code which can rely on all the (now) static information encoded in the witness. Note how once given the witness `(Step (Step Base))` the staged function `app3` can unroll the loop. So not only is the witness removed in the second stage, but the resulting program is no longer even recursive!

The ability to control extraction is important. Two different programs extracted from the same proof object may have very different physical properties (i.e. heap space usage). Staging allows users to extract programs in a manner that fits their needs.

## 12 Example: Proof Carrying Code

Peter Lee, on his web site states[22]: *Proof-Carrying Code (PCC) is a technique by which a code consumer (e.g., host) can verify that code provided by an untrusted code producer adheres to a predefined set of safety rules ... The key idea behind proof-carrying code is that the code producer is required to create a formal safety proof that attests to the fact that the code respects the defined safety policy. Then, the code consumer is able to use a simple and fast proof validator to check, with certainty, that the proof is valid and hence the foreign code is safe to execute.*

In Figure 9 we illustrate how this might be implemented using Ωmega. The code producer produces code whose safety policy is embedded in the type of the object-code as we have illustrated in the previous section. The producer than marshalls (pretty prints) this code into some flat untyped representation that can be transported over the Internet (a `String` in the figure). On the consumer side, the consumer unmarshalls (parses) this string into an untyped annotated abstract syntax tree. The check is a dynamic (i.e. at run-time) attempt to reconstruct the typed object-code (a static property) from the annotated untyped AST. If this succeeds then the consumer has a proof that the object code has the desired safety property, since all well typed object-programs have the safety property. The only difficult step in this process is the reconstruction of the typed object-code from the untyped annotated AST. In order to describe how this is done we introduce additional features of Ωmega, polymorphic kinds and representation types. We apply these features to the dynamic construction of the statically typed `Exp` datatype from the while-program example (Figure 8).

In Figure 10 we define two untyped algebraic datatypes `TyAst` and `ExpAst` that we will use as our annotated abstract syntax types. The type `TypeR` is a representation type. It reflects objects that live in the type world (`Int`, `Bool`, and pairs) into the value world. Note how `IntR::(TypeR Int)` is a value, but its type completely distinguishes what value it is. This notion has been called *singleton types*[37, 32], but we think *representation types* is a more appropriate name. Writing a program that manipulates representation types allows the programmer to encode operations that the type system (with its limited computation mechanism – essentially solving equalities between types) cannot. *It cannot be over-emphasized how important this ability is.* Typing problems that cannot be solved by the type system can be programmed by the user when necessary.

We choose to represent `Int`, `Bool` and pairs because these types either appear as type indexes to `Exp` and `Com` or describe the shape of the store as a nested pair. The key to dynamic reconstruction of static type information is the `Eq` data type. The `Eq` type constructor has a polymorphic kind (`Eq::forall (k:*1) (k1:*1) . k ~> k1 ~> *0`). This kind means that the arguments to `Eq` can range over any two types classified by `k` and `k1` that are themselves classified by `*1`. This includes types like `Int` and `Bool`, as well as type constructors like `Tree` and `List`.

The constructor function (`EqProof::forall (k:*1) (u:k) (v:k).(u = v) => Eq u v)`) is a first-class (dynamic) witness to the fact that the static types `u` and `v` are equal. Equality witnesses can be created in a static context where `u` is equal to `v` then passed around as data to a new context where this information is needed. One way to create these witnesses is the use of the function `match::forall u v.TypeR u -> TypeR v -> Maybe(Eq u v)`. The function `match` dynamically tests whether two representation types are equal. If they are, rather than return a boolean value, it returns either a successful equality witness or it returns a failure. The witness can be used in a pattern matching context to guard an expression with this new piece of static information (that `u=v`). For example, given that `x` has the type `Eq u v`, in the case expression: `(case x of { Eq -> ...  })`, the case arm indicated by `...` can be type checked under the static assumption that `u=v`.

The standard typing rules for equality qualified types provide this mechanism. There is nothing new here, only a new way of using the old techniques. The datatypes `EJudgment` and `TJudgment` are forms of `TypeR` and `Exp` that use existential types to hide some of the type indexes to those type constructor functions. `EJudgment` also includes a representation of the type `t`.

The functions `match`, `checkT`, and `checkE` are examples of partial functions. They might succeed, producing some result `ans`, but they also might fail. In Ωmega this is indicated by a result type `(Maybe ans)`. They are programmed using the `do` notation which makes it

```
kind State = Open | Closed

prog2::Com (Int,(Int,a)) Open Open
prog2 =
 Seq (Set sum (Int 0))
     (Seq (Set x (Int 1))
     (While (Lteq (Var x) (Int 5))
         (Seq (Set sum (Plus (Var sum)(Var x)))
         (Seq (Writef (Var x))
              (Set x (Plus (Var x) (Int 1)))))))

data Com st x y
  = forall t . Set (V st t) (Exp st t) where x=y
  | forall a . Seq (Com st x a) (Com st a y)
  | If (Exp st Bool) (Com st x y) (Com st x y)
  | While (Exp st Bool) (Com st x y) where x = y
  | forall t . Declare (Exp st t) (Com (t,st) x y)
  | Openf where x = Closed, y = Open
  | Closef where x = Open, y = Closed
  | Writef (Exp st Int) where x = Open, y = Open
```

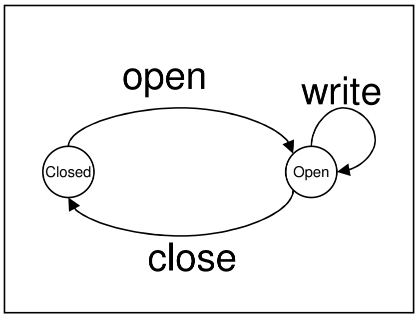


**Figure 12. The While-language augmented with commands for manipulating a file, and a DFA illustrating the protocol.**

easy to program partial functions that are comprised of sub computations that might also fail. A sequence of partial computations `do { p1 <- e1; ... ; pn <- en }` succeeds only if all the $e_i$ succeed. If any of them fails then the whole sequence fails. If the $e_i$ succeeds with a structured data object, then the $p_i$ can be used to pattern match against the result if it is successful. If the $e_i$ is successful but the object returned doesn't match against the $p_i$ then the whole sequence fails as well.

We explain one clause of the definition of `checkE`. Consider `checkE (PlusA x y) sr = ...` First, recursively check the subterm of the annotated AST, `x`. This returns a judgment encapsulating a typed term `(e1::Exp s _a)` and a representation of its type `(t1::TypeR _a)` where `_a` is an existentially quantified type variable. Test if this representation matches `IntR`. If it succeeds the witness `(EqProof::Eq Int _a)` is pattern matched and the rest of the computation can proceed under the static assumption that `_a` is equal to `Int`. In a similar fashion check and then test `y`, and finally succeed with a new judgment.

**Possible Enhancements.** This technique can be extended to the full While-language including the `Com` language. In that case, the judgment for commands must include representations for stores in the way that the judgment for expressions contained representations for types. The same techniques can be used to infer well typed object-code terms from untyped abstract syntax trees without annotations, but the details become more complicated. The reflection of the type world into the value world is a powerful idea. It lets the user dynamically construct objects with static properties that the static type system may not be able to infer with its limited computational mechanism.

## 13 Example: A Language with Temporal Safety Properties

Many systems depend upon communication occurring according to a temporal protocol. For example a file must be opened before it can be written to. Once opened, a file shouldn't be opened again until after it has been closed. A closed file should never be written to. Such protocols are naturally expressed as finite state automata. The DFA in Figure12 captures this protocol precisely.

A language can express and enforce such protocols quite naturally using its type system. To illustrate this we have augmented the While-language with commands for opening, closing , and writing to a single file (we discuss removing this restriction later).

In Figure 12 we have defined a new kind `State` with types `Open` and `Closed`, and augmented the command data structure with three new constructor functions: `Openf`, `Closef`, and `Writef`. The `Com` type now takes two additional type parameters. Interpret the type `(Com st x y)` as a command in store `st`, starting execution in state `x` and ending in state `y`. The types of the new constructors enforce the protocol: `(Openf::Com st Closed Open)`, `(Closef:: Com st Open Closed)`, and `(Writef::Exp st Int -> Com st Open Open)`. The type of a command such as `prog2` from Figure 12 describes precisely in which states of the protocol the command resides. Commands with polymorphic starting and ending states, essentially carry a proof that they do no IO at all!

**Possible Enhancements.** It is easy to imagine richer protocols with DFA's with more than two states. Accommodating such protocols simply requires enriching the `State` kind, and adding new commands for each transition. If the host language has a notion of typed procedures it isn't necessary to add new constructor functions to `Com` for each transition in the DFA. Languages with multiple protocols, or with more than 1 file can be accommodated by specifying the starting and ending state parameters of `Com` be structured types with more than one component.

## 14 Example: Multi-Level Security

Our next example concerns a language with multi-level security domains. A multi-level security language is meant to ensure confidentiality of information stored at higher levels of the security hierarchy. In such a language data is partitioned into security domains, for example a two level domain might have two distinct levels *High* and *Low*.

The key semantic property is to insure that the value of data at higher levels never influences the value of data at lower levels. This is tricky because control flow decisions, predicated on high security information, can cause information to leak to lower levels. The example below has this problem:

```
{ high int x;
  low int y;
  if (x==0)
     then y := 0
     else y := 1
}
```

To reason about confidentiality we need an object-language in which we can reason about information flow. In Figure 13 we define such a language based on similar languages from the literature [43, 31].

```
Domain :: *1
kind Domain = High | Low  -- High,Low::Domain

D::Domain ~> *
data D t
  = Lo where t = Low     -- Lo::D Low
  | Hi where t = High    -- Hi::D High

data Dless x y
  = LH where x = Low, y = High
  | LL where x = Low, y = Low
  | HH where x = High, y = High

data P x y = P

data V s d t
  = forall s0 d0 . Z (D d)
      where s = P (D d,d0) (t,s0)
  | forall a b t1 d1 . S (V (P a b) d t)
      where s = P (d1,a) (t1,b)

eval :: Exp (P a s) d t -> s -> t
exec :: (Com d (P a st)) -> st -> st
```

```
Exp :: * ~> Domain ~> * ~> *
data Exp s d t
  = Int Int where t = Int
  | Bool Bool where t = Bool
  | Plus (Exp s d Int) (Exp s d Int) where t = Int
  | Lteq (Exp s d Int) (Exp s d Int) where t = Bool
  | forall d2 . Var (V s d2 t) (Dless d2 d)

Com :: Domain ~> * ~> *
data Com d st
  = forall t d1 d2 .
      Set (V st d2 t) (Exp st d1 t)
          (Dless d1 d2) (Dless d d2)
  | Seq (Com d st) (Com d st)
  | If (Exp st d Bool) (Com d st) (Com d st)
  | While (Exp st d Bool) (Com d st)
  | forall t d2 a b .
      Declare (D d2) (Exp st d2 t)
              (Com d (P (D d2,a) (t,b)))
      where st = P a b

update :: (V (P a s) d t) -> t -> s -> s
update (Z d) n (x,y) = (n,y)
update (S v) n (x,y) = (x,update v n y)
```

**Figure 13. Security Domains**

The `kind` declaration in Figure 13 introduces a new kind, `Domain`, and two new types, `High` and `Low`. The `data` declaration for `D` introduces a new type constructor with an interesting kind: `(D::Domain ⇝ *)`. Like other `data` declarations its also introduces new values `Hi` and `Lo`. The type `D` reflects the structure of the kind `Domain` into the value world, and the type of `Hi` and `Lo` are indexed by the types (`High` and `Low`) they represent: `(Lo::D Low)` and `(Hi::D High)`.

The security language is closely related to the While-language. The main difference is the introduction and use of domains. This necessitates a change in the way we type stores. In the While-language the type of a store was a nested tuple encoding the types of the variables in scope. In the security language, the types of the variables is not enough – we must also encode the `Domain` of each variable. This is the role of the type constructor `P` (think of `(P x y)` as a special kind of pair). In the While-language a command typed as `(Com (Int,(Bool,a)))` would be typed as `(Com (P (D High,(D Low,b)) (Int,(Bool,a))))` in the security language. The type parameter to `Exp` and `Com` describing stores is now a `P` pair. The second component of the pair is exactly as in the While-language, and the first component of the pair is a parallel structure (with the same nesting shape as the second) but storing representations of the `Domain` of variables rather than their types.

The interpretation of a command with type `(Com d s)` is a command in store `s` executing in a control thread in domain `d`. A similar interpretation applies to expressions with types `(Exp s d t)` except that a expression also returns a value of type `t`. Security in the language is enforced by the `Dless` witnesses in `Var` and `Set` constructors. Consider: `(Var::V s d2 t -> Dless d2 d -> Exp s d t)`, a variable expression is well formed only if the domain of the variable (`d2`) is less than the thread of execution (`d`). Information can flow from `Low` variables into `High` threads, but not the other way around. For the assignments constructor function we have `(Set::V s d2 t -> Exp s d1 -> Dless d1 d2 -> Dless d d2 -> Com s d)`. The thread of the expression being assigned (`d1`) must be less than the domain of the variable being assigned to (`d2`). Anyone can assign to `High` variables, but only expressions in `Low` threads can assign to `Low` variables. In addition the thread of the assignment command (`d`) must be less than the thread of the variable (`d2`). This prevents the problem illustrated above of control flow predicated on `High` information being used to leak information into `Low` variables.

Given a semantics for this language (similar to the `eval` and `exec` commands for the While-language) it is easy to state and prove that the type system prevents adverse information flow. The proof is cast as a separation argument. Given a a well-typed command `(c::Com d (P ds st))` then its meaning `((exec c)::st -> st)` is a function from stores to stores, and values of low variables in the output store never depend on the values of the high variables in the input store.

# 15 Related Work

Ωmega's design has been influenced by work on Logical Frameworks[16, 30, 29], Inductive Families[14, 10], Refinement Types[15, 12], Practical Dependent Typing[1, 2], and Guarded Recursive Datatype Constructors[52, 8]. But the work on Equality Types[9] has allowed us to express many of these ideas as a simple extension (based upon qualified types) to the familiar notion of Algebraic Datatypes. By combining type inference with type checking for arbitrary rank polymorhism[19, 21, 35] and staging annotations[7, 40, 34, 38]; we have created a powerful new way to embed properties of programs in their types.

Expressing that two types are equal in a manner controllable by the programmer is the key to embedding semantic properties of object-programs. The first work expressing equality between types in a programming language 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[48], and was well developed two years later in 2002[3, 17]. Programming with 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 has some problems for which no solution is known[1]. Using type equality became practical with the introduction of equality qualified types by Hinze and Cheney[9]. The implementation of Ωmega is based on this key idea. We know that a type system built on top of equality constrained types is sound because of their work.

The use of kinds to classify types has a long history[4, 18, 23]. Adding extensible kinds (and higher classifications) to a practical programming language like Ωmega was a natural next step. Duggan makes use of kinds in his work on dynamic typing[13] in a manner reminiscent of our work, but the introduction of new kinds is tied to the introduction of types.

## 16 Conclusion

Ωmega is descended from functional programming languages – Its syntax and type system are similar to Haskell, but it adds several new features that make it possible to use algebraic datatypes to build proof-like objects as illustrated above. This approach to combining reasoning and programming in a single system makes it of interest to all programmers. Ωmega opens intriguing possibilities for the design, exploration, and implementation of programs with semantic properties. We believe exploring this point in the design space of programming languages and reasoning systems makes is an important step in the direction towards the programming languages of the future. Our path in this exploration is closer to the world of programming languages than the path of many other reasoning systems. We see this as a positive benefit and conjecture that programming languages of the future will be built along similar lines.

Ωmega supports a reflective mechanism that enables intensional analysis of reflected types, and thus allows programmers to write tactic level proof scripts at the value level on these reflections. The tactics can then be reflected back into the type system in a sound manner. Staging can be used to build efficient implementations by exploiting contextual invariants, it can also be used to extract efficient programs from proof like objects. Our sincere hope is that a programming language with these features can lead to more reliable programs.

## 17 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] 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] P. Bertelsen. Semantics of Java Byte Code. Technical report, Dep. of Information Technology, Technical University of Denmark, March 1997.


[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.


[6] Fréde'ric Besson, Thomas de Grenier de Latour, and Thomas Jensen. Secure calling contexts for stack inspection. In *Proceedings of the Fourth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-02)*, pages 76–87, New York, October 6–8 2002. ACM Press.

[7] Cristiano Calcagno, Eugenio Moggi, and Tim Sheard. Closed types for a safe imperative MetaML. *Journal of Functional Programming*, 13(12):545–572, May 2003.

[8] 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.

[9] James Cheney and Ralf Hinze. Phantom types. Available from `http://www.informatik.uni-bonn.de/~ralf/publications/Phantom.pdf.`, 2003.

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

[11] Karl Crary and Stephanie Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POLP-00)*, pages 184–198, N.Y., January 19–21 2000. ACM Press.

[12] 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.

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

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

[15] 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.

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

[17] 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.

[18] 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.

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

[20] J. Launchbury and R. Paterson. Parametricity and unboxing with unpointed types. *Lecture Notes in Computer Science*, 1058:204–??, 1996.

[21] Didier Le Botlan and Didier Rémy. $ML^F$: raising ML to the

power of system F. *ACM SIGPLAN Notices*, 38(9):27–38, September 2003.

[22] Peter Lee. Proof-carrying code. Available from http://www-2.cs.cmu.edu/~petel/papers/pcc/pcc.html.

[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] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. ACM SIGPLAN Workshop on Compiler Support for System Software, 1999.

[25] P. Ørbæk and J. Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6):557–591, November 1997.

[26] Emir Pasalic, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP-02).*, pages 218–229, Pittsburgh, PA., October 4–6 2002. ACM Press.

[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] Emir Pašalić, Tim Sheard, and Walid Taha. DALI: An untyped, CBV functional language supporting first-order datatypes with binders (technical development). Technical Report CSE-00-007, OGI, 2000. Available from http://www.cse.ogi.edu/PacSoft/.

[29] 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.

[30] 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.

[31] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

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

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

[34] T. Sheard and S. Peyton-Jones. Template meta-programming for haskell. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, pages 1–16. ACM, 2002.

[35] Vincent Simonet. An extension of HM(X) with bounded existential and universal data-types. *ACM SIGPLAN Notices*, 38(9):39–50, September 2003.

[36] R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *25th Annual ACM Symposium on Principles of Programming Languages*, pages 149–160, January 1998.

[37] Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–227, Boston, Massachusetts, January 19–21, 2000.

[38] Walid Taha. A sound reduction semantics for untyped CBN mutli-stage computation. Or, the theory of MetaML is non-trivial. In *2000 SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00)*, January 2000.

[39] Walid Taha, Henning Makholm, and John Hughes. Tag elimination and jones-optimality. *Lecture Notes in Computer Science*, 2053:257–??, 2001.

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

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

[42] Joseph C. Vanderwaart and Karl Crary. Foundational typed assembly language for grid computing. Technical Report CMU-CS-04-104, Carnegie Mellon University, 2004.

[43] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, December 1996.

[44] Philip Wadler. Comprehending monads. *Proceedings of the ACM Symposium on Lisp and Functional Programming, Nice, France*, pages 61–78, June 1990.

[45] Philip Wadler. The essence of functional programming (invited talk). In *19'th ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992.

[46] Philip Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi*, volume 118 of *NATO ASI series, Series F: Computer and System Sciences*. Springer Verlag, 1994. Proceedings of the International Summer School at Marktoberdorf directed by F. L. Bauer, M. Broy, E. W. Dijkstra, D. Gries, and C. A. R. Hoare.

[47] D. S. Wallach and E. W. Felten. Understanding java stack inspection. In *1998 IEEE Symposium on Security and Privacy (SSP '98)*, pages 52–65, Washington - Brussels - Tokyo, May 1998. IEEE.

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

[49] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.

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

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

[52] 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.