Combining Programming Languages and Logical Reasoning Systems
Project Summary

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. While many reasoning tools are built on the Curry-Howard isomorphism, it is often hard for the programmers to conceptualize how they can put this abstraction to work. We propose the design of a language that makes this important 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 actually construct programs that they 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.

Intellectual Merit: In order to make this vision real, we have chosen to explore a new point in the design space of formal reasoning systems. We propose the use of a programming language with a type system in which the user expresses equality constraints between types, which the type checker then enforces. 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. The addition of two other type extensions, rank-N polymorphism and extensible kinds, creates a powerful new programming idiom for writing programs whose types enforce semantic properties.

This idiom enforces a coding style which experienced users of theorem provers may find tedious – the programmer must be explicit about many things the theorem prover does automatically. But the automation hides crucial details that make using the theorem prover hard for beginners. By making these issues explicit, we ease the user into accepting the need for a reasoning system. We further argue, that the use of a reflection mechanism, can re-automate many of these tasks. So, we can have the best of both worlds.

We propose that a language with these features is *both* a practical programming language *and* a logic. This marriage between two previously separate entities further 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. We further propose that a language with these features makes an ideal meta-language that can be used to combine and reason about multiple layers of system design. Such a meta-language can play an important role in scripting and connecting more powerful tools (such as logical frameworks, theorem provers, generic analysis frameworks, and model checkers) when needed to further enhance system trust.

Broader Impact: Defective software has a tremendous societal cost. First are the obvious up-front costs of applying a continual stream of patches to installed software, combating viruses, and productivity lost when work has to be done again or performed in a sub-optimal manner. Second are the opportunity costs of defective software: projects that are never attempted, or whose scope is significantly reduced, because the cost of software failure is too high to contemplate. Finally, there are the well-documented costs of major software disasters. The research outlined in this proposal lays the foundation for building software that works.

As a graduate-only institution, OGI is in a unique position to promote and advance the introduction of new ideas into the work force. Our students are by and large employed professionals, uniquely motivated by the problems that they have experienced. Our close connections with industry mean not only that we are acutely aware of the problem of faulty software, but also that we have access to the industrial-strength tools that are being developed to solve them. The effort required to leverage these tools was one of the prime motivations for the research proposed. OGI has a tradition teaching research oriented topics in advanced graduate level courses, and in building and maintaining widely distributed tools.

1 Introduction

The ultimate goal of the proposed research is to construct tools and processes that can support the building of large software systems that are secure, reliable, and that have predictable properties. Achieving this goal is difficult because large systems are necessarily complex, and complex systems are hard to reason about. To control complexity we build systems in layers. Without a means of ensuring that important semantic properties of software are preserved across layer boundaries, any benefit (in terms of controlling complexity) gained by layering may be more than offset by the resultant loss in predictability, reliability and trust.

Fortunately, there are many mechanisms for specifying properties of systems (models, first order logic, higher order logic, modal logics), and there are many tools to prove that such specifications are sound (theorem provers like PVS, Isabelle, HOL98, and ACL2; and logical frameworks like Elf and Twelf). The problem is that the semantic gap between these formal tools and the languages in which the applications are implemented is huge. This gap prevents the application of formal methods to software design on all but the most important applications. If we are ever to build systems that we can trust on a large scale, we must develop programming languages that narrow this semantic gap. We propose research into the design of the programming languages of the future. Such languages 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 vast 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.

The goals of this proposal are: (1) To take the first steps in the design of the programming languages of the future. (2) To demonstrate that reasoning capabilities can be built into a practical programming language by strengthening the type system in ways that are easy for the programmer to understand. (3) To apply such a programming language to applications that require a heightened levels of predictability and trust.

What we are proposing. As a step in this direction, we propose to explore a new point in the design space of formal reasoning systems: the development of the language Ωmega. Ωmega is *both* a practical programming language *and* a logic. These sometimes irreconcilable goals are made possible by embedding the Ωmega logic in a type system based on *equality qualified* types. This design supports the construction, maintenance, and propagation of semantic properties of programs using powerful old ideas about types in novel new ways.

How is this different from previous work? Theorem provers and logical frameworks have many of the same goals, but we believe there are qualitative differences between them and the proposed work.

First, Ωmega is a practical 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 to separate effects from 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 3.

Third, Ωmega incorporates several powerful extension mechanisms. In Coq and other related systems, proofs *correspond* to programs. In Ωmega proofs *are* programs (with equality qualified types). More efficient implementations can often be extracted from proofs by a form of type erasure. Unlike Coq[61], and Isabelle[39] 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. A second extension mechanism is Ωmega's ability to reflect representations of its types into the value world and to perform arbitrary computations on these representations in a type safe manner. Because the logic of Ωmega is embedded in its type system, the sound reflection mechanism supports extension of Ωmega's logic to deal with a wide variety of properties, both logical (semantic), and physical (resource usage).

Why now? We have already developed a preliminary version of Ωmega. Its design has been heavily influenced by a set of recent advances in the programming language community. The ability to combine type inference with type checking and arbitrary rank polymorhism[22, 25, 53]; the semantics of staged computation systems[7, 60, 49, 57]; and the use of simplified form of dependent typing called *indexed types*[70, 9, 10] have combined to create a powerful new way to embed properties of programs in their types.

Why here? The P.I. is a pioneer in staged computation and meta-programming systems. In 2 previous NSF supported projects *Type Safe Program Generators* (CCR-9625462, 10/96-03/00), and *Heterogeneous Meta Programming Systems* (CCR-0098126 10/01-06/04), the P.I. and his students and collaborators have been instrumental in the design and implementation of staged systems. This proposal is the logical next step of applying these ideas to reason about real world systems.

Why this approach? Ωmega is not just another functional programming language. Ωmega is clearly descended from Haskell. Its syntax and type system are similar, but it has features that Haskell does not. It is strict, it has polymorphic kinds, kind extension, staging, and types qualified by equality predicates. But it also drops some features of Haskell. This was done to simplify its semantics so that it is easier to reason about. The features dropped include laziness, which can be simulated by staging; and the class system, which can be simulated by Ωmega's richer types. 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 progress towards the goals outlined above.

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. Types have been used to ensure the safety of low level code such as Java Byte Code[55, 3] or typed assembly language[33, 34]. 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[45, 63, 35] to ensure security properties of systems. Types have been used to track resource control, such as the possibility of non-termination [24], or to place upper bounds on the time consumed by a computation[11, 62]. Types have been used as a means of removing

value		type		kind		sort
5	::	Int	::	*0	::	*1
				Nat	::	*1
		Z	::	Nat	::	*1
		Succ	::	Nat $\sim\succ$ Nat	::	*1
		Seq	::	*0 $\sim\succ$ Nat $\sim\succ$ *0	::	*1
		Sum	::	Nat $\sim\succ$ Nat $\sim\succ$ Nat $\sim\succ$ *0	::	*1
Nil	::	Seq α Z	::	*0	::	*1
Cons	::	$\alpha \rightarrow$ Seq α n $\rightarrow$ Seq α (S n)	::	*0	::	*1
Base	::	Sum Z n n	::	*0	::	*1
Step	::	Sum m n o $\rightarrow$Sum(S m) n (S o)	::	*0	::	*1

Figure 1: Classification of values(Nil,Cons,Base,and Step), types (Z,Succ,Sum, and Seq), and kinds (Nat) defined in Figure 2

dynamic error tests – for example, to enforce data structure invariants[69] (such as ensuring red-black trees are well formed) or to make code more efficient by removing unnecessary run-time array bounds checks[70]. Finally, types have been used to track access control, which allows removing (or minimizing) stack inspection overhead as a means of managing capabilities[65, 4].

As far as the proposer can tell from the literature, each of these systems was built using a general purpose programming language. While the properties of these systems could be modelled by a formal system such as a logical framework or theorem prover such as Coq[61], Isabelle[39], or Twelf[41], the properties are a meta-logical property of the program and external to the implementation. In Ωmega they could be a property of the implementation, which could thus be enforced by the programming language. 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, we can both implement and reason in a single paradigm with Ωmega.

While formal reasoning systems are very good at what they do, they were not designed to be programming languages. These tools are too expressive. There is something to be gained by being selective, choosing features wisely, and maintaining 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.

We have coined a new slogan for the process of designing trustworthy systems: *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. Adding rank-N polymorphism, equality qualified types, extensible kinds, and staging support makes this light-weight formality possible. Programmers who use languages like O'Caml, Standard ML, or Haskell will find these extensions familiar. For these programmers the learning curve will be small. Those 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 An Introduction to Ωmega

In this section we introduce Ωmega by comparing it to other different formal reasoning systems: Coq, and Twelf. We claim that Ωmega's single computational model makes it easier to state and maintain semantic properties of programs than using either of the other two. To be concise, we use a simple example, but our experience has shown the results to be similar in much larger examples as well. 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 $a \rightarrow Seq\ a\ n \rightarrow Seq\ a\ (n + 1)$. The type of the append operator would be

```
kind Nat = Z | S Nat

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
```

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

app :: Sum n m p -> Seq a n -> Seq a m -> Seq a p
app Base Nil ys = ys
app (Step p) (Cons x xs) ys = Cons x (app p xs ys)
```

Ωmega 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.
```

Coq 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.
```

Twelf encoding

Figure 2: Comparison of three formal systems encoding lists whose types record their lengths.

$Seq\ a\ n \rightarrow Seq\ a\ m \rightarrow Seq\ a\ (n+m)$. In order to type such functions it is necessary to do arithmetic at the type level. In Figure 2 this is done in three different formal systems. The first encoding is in our preliminary version of Ωmega. The Ωmega example introduces two new types (`Sum` and `Seq`), a new function (`app`), and a new `kind` (`Nat`). The new kind `Nat` introduces two new type constructors `Z` and `S` which encode the natural numbers at the type level.

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). `*0` is the kind that classifies all types that classify values (things we actually can compute). `*0` is classified by `*1`, etc. We sometimes write `*` as a shorthand for `*0`. There is an infinite hierarchy of classifications. 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. In Figure 1 We illustrate the relationship between values, types, kinds, and sorts introduced in Figure 2.

Constructor functions (`Nil`, `Cons`, `Base`, and `Step`) construct elements of `data` types. The type of a constructor function is described in the `data` declaration. For example, the clause in the `Seq` declaration: `exists m.Cons a (Seq a m) where n = S m` introduces the `Cons` constructor function. Without the `where` qualification, the constructor function `Cons` would have type `(Cons::a -> Seq a m -> Seq a n)`. *Equality Qualification* (indicated by the `where` in the clauses for `Nil`, `Cons`, `Base`, and `Step`) and *existential quantification* (indicated by `exists` in the clauses for `Cons`, and `Step`) help encode semantic properties. The `where` *qualifies* `Cons`' type, in effect saying `(Cons::a -> Seq a m -> Seq a n)` *provided* `n=S m`. We capture this formally by writing `Cons::(forall a n m.(n=S m)=>a -> Seq a m -> Seq a 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, `Cons` also has the type `(forall a m.a -> Seq a m -> Seq a (S m))`. This type guarantees that `Cons` can only be applied in contexts where `n=S m`. Existential quantification of the type variable `m` names the intermediate length of the sublist of `Cons`, which if not introduced in this way would appear as an unbound type variable.

Equality constrained types are a relatively new feature in the world of programming languages, and were only recently introduced by Hinze and Cheney[10]. We can use the mechanism to model relations between types, other than equality, by defining witness types. A witness is a value constructed by the constructor functions (like `Base` and `Step`) of some `data` definition (like `Sum`). The type of such a value encodes the property. The very existence of the witness implies that the property must be true. Witnesses to untrue properties cannot be constructed since such values would be ill-typed. A value of type `(Sum m n o)` witnesses the ternary arithmetic relation `m+n=o`.

Ωmega's types are used to enforce the property that the length of appending two lists is the sum of the length of the two lists appended `(app::Sum n m p -> Seq a n -> Seq a m -> Seq a p)`. The first argument to `app` is a witness to the crucial property. Consider the first clause defining the append function `app Base Nil ys = ys` – how is this typed? We know `app`'s type, so the first argument `Base` must have type `(Sum n m p)`, and the second argument `Nil` must have type `Seq a n`, and the third argument `ys` must have type `(Seq a m)`. The right-hand-side of the equation should then have type `(Seq a p)`. But, since the right-hand-side is the same as the second argument, this clause appears ill-typed. In short we write:

$$\{\mathit{Base} :: \mathit{Sum}\, n\; m\; p,\ \mathit{Nil} :: \mathit{Seq}\, a\; n,\ \mathit{ys} :: \mathit{Seq}\, a\; m\} \vdash \mathit{ys} :: \mathit{Seq}\, a\; p$$

The key to type checking this clause, is to recognize that the constructor functions `Nil` and `Base` have equality qualified types. In particular when they were constructed it must have been the case that `n=Z` (from Nil) and that `n=Z` and `m=p` (from Base). So the complete typing judgment becomes:

$$\{\mathit{Base} :: \mathit{Sum}\, n\; m\; p,\ \mathit{Nil} :: \mathit{Seq}\, a\; n,\ \mathit{ys} :: \mathit{Seq}\, a\; m,\ n = Z,\ m = p\} \vdash \mathit{ys} :: \mathit{Seq}\, a\; p$$

which is easily shown to be true.

The propagation and solving of equality qualifications is handled by the compiler and type checker. The user is simply required to introduce equalities by using the `where` clause in `data` definitions, and stating the type of the function by giving its type signature (i.e. `app::Sum n m p -> Seq a n -> Seq a m -> Seq a p`) and the compiler does the rest. If a type signature is not supplied, the compiler will attempt to infer a Hindley-Milner polymorphic type for the function. Hindley-Milner inference for `app` would fail since it uses polymorphic recursion. The important thing to note is that Ωmega uses a combination of type inference and type checking. The presence of type signatures indicates that a function should be type checked. We do not believe that supplying type signatures for such functions is overly burdensome. Since the types encode properties of the object-language, the user ought to know what type his functions have, since it corresponds to the properties he is trying to model. If the function type checks, then the user has a proof that the program has the property described by the equalities between types.

3.1 A Comparison of Formal Reasoning Systems

We now come to the comparison part of this section. In the Coq and Twelf encodings in Figure 2 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

```
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 3: 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.

programs are not written in Prolog) we must think in terms of logic programs.

To be fair, we point out two caveats to the above arguments we address later. First, in Ωmega we must implement the `Sum` witness in a logical style. This style is closer to Twelf's logical style than Coq's functional style, so in Ωmega it appears we must think logically rather than functionally (at least at the type level). This is a consequence of the mechanism used to solve equality constraints. Second, (this will probably only make sense to those familiar with Coq) we could have defined append as a set, rather than a proposition, and then defined it by induction as we did in Ωmega. Had we done so we could no longer extract an efficient program from this definition. By combining the programming language and the logic we can address both these issues. In Section 8 we discuss removing the relational bias from the type system, and in Section 6 we discuss extracting efficient programs.

4 A Type-Safe and Statically-Scoped While Language

We now turn to a richer example: modelling a simple imperative *while* language with semantic properties of static scoping and type safety[38, 40]. 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 3 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 3. 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*.

```
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 4: 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[59], and they return values whose types correspond to the types of the while-programs.

An observation about the type parameters of Ωmega type constructors. The parameters of type constructors in the while-language play a qualitatively different role than type parameters in other data structures. Consider the declaration for a binary tree datatype:

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

In this declaration the type parameter `a` is used to indicate that there are sub components of `Tree`s that are of type `a`. In fact, `Tree`s are polymorphic. Any type of value can be placed in the "sub component" of type `a`. The type of the value placed there is reflected in the `Tree`'s type. Contrast this with `(Com s)`. Here there are no sub components of type `s`. Instead, the parameter `s` is used to stand for an abstract property (the types of the statically reachable object-variables). The `where` qualifications restrict the legal instances of `s`. Type parameters used in this way are sometimes called index types[68, 70].

Manipulating while-programs. In Figure 4 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. In this model variables are modeled by integers (using a de Bruijn-like notation), and stores are modelled by 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`.

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))` $\longrightarrow$ `(12,update Z False (True,0))` $\longrightarrow$ `(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))` $\longrightarrow$ `(eval (Var Z) (True,0))` $\longrightarrow$ `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` and `x = S Z`, observe:

```
prog :: Com (Int,(Int,a))
```

```
prog = Seq (Set sum (Int 0))                        -- { sum = 0;
        (Seq (Set x (Int 1))                        --   x = 1;
        (While (Lteq (Var x) (Int 5))               --   while (x <= 5)
          (Seq (Set sum (Plus (Var sum)(Var x)))    --    { sum = sum + x;
          (Set x (Plus (Var x)(Int 1))))))          --      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[67].

5 Extrapolating to a Realistic Example

The Border Gateway Protocol (BGP) is the protocol used to exchange path routing information between different autonomous systems (i.e. sub networks administrated by different companies such as ISP's or backbone suppliers). BGP is a dynamic routing mechanism as the path information evolves over time. BGP tries to meet two sets of competing demands. On one hand, the protocol must reliably establish paths between network addresses over a constantly changing network configuration. On the other hand, it must allow the administrators of each autonomous system to make decisions about when and how to advertise paths that meet their entrepreneurial goals (i.e. preferring routes over their own networks or those of their partners with which they have financial agreements). System administrators meet both sets of goals by using the BGP protocol to write policy functions. The construction of bad policy functions (either by accident or by malicious intent) has dire consequences. Such policies can break the network (making some addresses unreachable), or cause large financial losses. There has been much recent interest in devising domain specific policy languages (DSPL) that guarantee network integrity (such as eventual convergence to stable or optimal paths) while allowing the freedom to craft policies to meet entrepreneurial goals. The guarantees supported by these DSPLs are generally based upon mathematical models of path vector protocols[19, 54]. One such model, proposed by Sobrinho[54] models path vector protocols by an algebra, and shows if the algebra has certain necessary properties then any network protocol built with that algebra will have some desired properties. The key to applying these techniques is to design DSPL's such that every DSPL program is guaranteed either (1) to fit within some known algebra (easy but restrictive) or (2) can be described by some as yet unknown algebra that also has the desired property. The latter is harder, as the algebra often has to be constructed and then proved to have the required property, but is more flexible. Constructing such an algebra requires sophisticated algorithms, and proving its properties relies on using automated decision procedures such as BDD's or SAT-solvers.

Ωmega can be invaluable in this process by modeling the syntax and static semantics (type system) of the DSPL as shown in the previous section. Ωmega can then be used as a programming language to derive a program specific algebra. Ωmega can then be used to prove that the constructed algebra supports the necessary properties. If it does the DSPL program is compiled, or if it does not, Ωmega can reconstruct an error message from the failure trace to provide a domain specific error message explaining why the DSPL program is not safe. Ωmega's use of property encoding types will disallow certain classes of semantic errors, and catch other kinds of errors earlier.

```
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 5: Illustrating Staging, removal of interpretive overhead (top), and witness removal (bottom).

6 Staging Supports 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 [14, 42, 26, 29], dynamic compilation [5, 6, 18, 17], and program generators[30]. Staging is the key technology that supports efficient implementations without interpretive overhead. In 2 previous NSF supported projects *Type Safe Program Generators* (CCR-9625462, 10/96-03/00), and *Heterogeneous Meta Programming Systems* (CCR-0098126 10/01-06/04), the proposer has reported on the design[60], use[47, 48], semantics[57], type systems[7, 32, 51], implementation[31, 49, 8], and open problems[50] of meta-programming systems.

Staging is an programming language interface to code generation. We have built two large sophisticated systems that implement staging. MetaML[47], a system with run-time code generation, and Template Haskell[49], a system with compile-time code generation (think macros, quasi-quotes, and type safety). In Figure 5 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 4). 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 5 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 hard-wired 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 Figure 2 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`

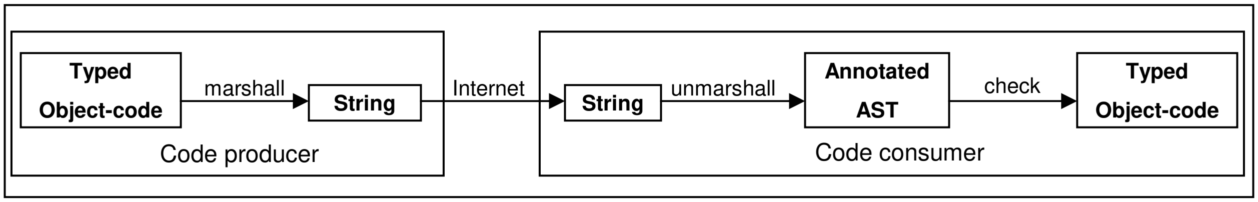

Figure 6: Proof carrying code process

`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.

7 Example: Proof Carrying Code

Peter Lee, on his web site states[27]: *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 6 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 4).

In Figure 7 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*[56, 46], 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`.

```
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 7: Implementing the check function for the proof carrying code example.

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 existentially hide some of the type indexes to those type constructor functions. `EJudgment` also encapsulates a representation of the type `t` that it encapsulates.

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 easy to program partial functions that are comprised of sub computations that might also fail. A sequence of partial computations `do` { p_1 `<-` e_1`; ... ;` p_n `<-` e_n } 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. We believe 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 dynanmically construct objects with static properties that the static type system may not be able to infer with its limited computational mechanism.

8 Research Plan

The research proposed falls into three broad categories: theoretical, implementation, and applications. We plan to enhance our preliminary design based upon our theoretical investigations, build a robust implementation incorporating useful meta-programming features, and develop a corpus of *design patterns* of the use of Ωmega as a means of documenting its use. We also plan to teach the use of Ωmega in our graduate level degree program in High Assurance Software. We discuss our plan in each of these areas in the following sub sections.

8.1 Theoretical Issues

The features of rank-N polymorphism, equality qualified types, and polymorphic kinds are all well studied. The knowledge of how to incorporate them into a programming language in a safe manner is a very recent accomplishment. Ωmega is a synthesis of these and other ideas. This synthesis leads to some theoretical questions. In this section we discuss the recent work and theoretical issues yet to be resolved.

Rank-N Polymorphism. Polymorphism is a powerful technique. Combining it with parametricity[43, 44, 64] allows types to be used to express interesting properties of programs. Rank-N polymorphism supports functions (and data structures) that take polymorphic functions as arguments (as opposed to Hindley-Milner polymorphism where all polymorphism is at the outer level of a type). System F[16] captures rank-N polymorphism but it requires type annotations on every binding site just to do type checking, and in most interesting cases type inference is undecidable. Discovering a practical mechanism that supports rank-N polymorphism, requiring type annotations only when rank-N types are desired, and performs Hindley-Milner type inference elsewhere, in a manner that is understandable by humans is a very recent accomplishment[25, 53]. Ωmega's support of rank-N polymorphism is based upon an unpublished paper *Practical type inference for arbitrary-rank types* [22] by Simon Peyton Jones and Mark Shields. This paper was particularly useful as it explained in practical terms how to build such a type checker – in particular how to set up the subsumption relation that describes which types are more polymorphic than others.

Research issues still to be resolved revolve around the use of rank-N polymorphism and Higher-order Type constructors. The current system supports rank-N types (as described in [22]) only on the back-end of the arrow type constructor (for example: `(forall a .  a -> a) -> T Int`). This is sufficient to model the examples we have done so far but extending this to arguments of other type constructors (i.e. `T Int (forall a .  a -> a) Bool`) requires extending the notion of subsumption to type constructors other than the type arrow. We hope to find inspiration in other work [25, 53] in this area.

Equality Qualified 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 based 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`

```
kind Nat = Z | S Nat

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

```
{Plus Z y} = y
{Plus (S x) y} = S {Plus x y}

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

Figure 8: Type functions restricted to equality qualifications

`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[66], and was well developed two years later in 2002[1, 20]. 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[10]. 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 work by Hinze and Cheney[10]. What happens with the addition of rank-N polymorphism, extensible and polymorphic kinds, and staging?

Logical Soundness. The soundness of the type system ensures that well-typed programs do not go wrong at run-time. But, this is not enough. We need logical soundness as well. When a type indicates a program has a property, this really must be the case. It is possible to spoof a property when the semantics of the language includes non-terminating computations. Divergent computations can often be given any type. In this situation, the type of a program may only indicate that it contains a divergent computation, rather than having the desired property indicated by the type. Thus it becomes important to track non-termination. As we see it, there are two possible approaches to tracking non-termination. First, use the type system itself to track non-termination as suggested in the work of Launchbury and Paterson[24]. The second approach is to use a separate termination analysis on the definition of every function along the lines as is done in ACL/2[23]. It may even be possible to combine the two approaches. *Ensuring logical soundness is the biggest research challenge posed by this proposal.*

Polymorphic Kinds. The use of kinds to classify types has a long history[2, 21, 33]. Adding extensible kinds (and higher classifications) to a practical programming language like Ωmega was a natural next step. Research in this area revolves around use of polymorphic kinds, in particular the use of types that have polymorphic recursive kinds. This occurs when an `data` type definition is given a polymorphic kind signature, and the type being defined is used in its own definition at more than one instance of its (polymorphic) kind.

Removing the Relational Bias in the Type System. We believe strongly that a functional approach to specifying properties is easier to teach and to learn and has a much smaller learning curve than a relational approach, especially for beginners. Unfortunately, while Ωmega is functional, the approach we have outlined so far relies on a relational model at the type level. An example illustrates this best. In Figure 2 we defined polymorphic sequences whose lengths are recorded in their types. Thus a term with type `(Seq Int (S Z))` is a sequence of integers whose length is one. In order to define an append function on this type, we needed to encode addition on types of kind `Nat`. We did this using a relational approach. The value of type `(Sum w x y)` is a witness that `w + x = y`. This relational approach was made necessary by the equality qualified type system. The mechanism for solving equality qualifications depends crucially on congruence laws. For example, from the equality `(Tree x)` = `(Tree y)` we can conclude that `x` = `y`. Such congruence laws must hold for all type constructors. Thus if we allowed functions to be defined at the type level we could express the type of `app` as `Seq a n -> Seq a m -> Seq a (Plus n m)`, but we'd lose the congruence laws since we cannot conclude `Z` = `(S z)` from `(Plus Z (S Z)) = (Plus (S Z) Z)`.

We believe it is still possible to remove the relational bias, and retain the congruence laws by putting

[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.

some restrictions on the formation of types. This is illustrated in Figure 8. Here we separate type constructors (like `Tree` and `Seq`) from type functions (like `Plus`) by enclosing applications of type functions in braces. Congruence holds for type constructors only. We restrict type function applications to the qualifications of types (behind the fat arrow (`=>`) only). We allow type functions to be defined by pattern matching equations, just as we define functions at the value level, but we use the brace notation to indicate type function application. Such equations are only used to simplify equations before solving. We have only begun to think about how to implement such a system. We must also prove the soundness of the type system under such a drastic change, and prove the decidability of solving equations that derive from the use type function application.

8.2 Implementation

We have built a prototype of Ωmega as a proof of concept. It is just an interpreter. A robust implementation would include a compiler and additional meta-programming features. Features such as parsing[28] and freshness[15] seem too important not to include in a language designed to be a meta-language.

Freshness. Some object languages that have binding constructs which have a notion of alpha equivalence (i.e. the renaming of local variables does not matter). In such a case, a good meta-language should not distinguish between two alpha equivalent object-programs. In our examples in the previous sections we finessed this problem by using de Bruijn indices to represent binding. Other new powerful techniques built on the notions of freshness and the permutation of names have recently been defined using a Fraenkel-Mostowski universe of sets with atoms[52]. This kind of system has been shown to provide a good, syntax-independent mathematical model of fresh bindable names and α-conversion [15]. Adding freshness to Ωmega will make it a more useful meta-language. Does this have any effect on the properties we desire for Ωmega? We are particularly concerned about the effects freshness will have on the inclusion of staging.

Practical features. Because the logic of the system is embedded in the types, proof failures manifest themselves as type checking errors. An interactive type exploration mechanism could allow users to explore the internal state of a failed type checking run to debug their proofs. Other features such as a foreign function interface are also necessary if Ωmega is ever to become a practical language as we desire. Other practical concerns such as efficiency and useable error messages also pose significant engineering challenges.

8.3 Applications

A handbook of examples illustrating how to apply Ωmega to paradigmatic object-languages, or how to use Ωmega to build bridges to other tools may be the most valuable artifact of the research. Examples of programming paragons include the use of representation types to move computation into the value world, the use of de Bruijn like indices to handle binding, the use of data types as witnesses of relationships between types, and the use of the `Eq` type (Section 6) to make dynamic tokens of static relationships. One can think of these as Meta-programming patterns. Other important patterns we would like to investigate include:

Property preserving transformation. Formal properties of languages can be mapped across layers written in different object languages by the use of property preserving transformation. In this scheme important properties are captured in two layers (often using different mechanisms in each layer) and the translation between layers preserves the property. An example motivated by the paper *From System F to Typed Assembly Language*[33] would be to introduce a typed assembly object-language, and demonstrates how the while language can be mapped into the assembly language in a way that preserves its semantic properties.

Efficient Implementation of DSLs Using Staging. Staging supports runtime code generation[60, 47]. By the use of staging and typed object languages we can create object language implementations that have neither interpretive overhead, nor tagging overhead [59, 58]. We believe it is possible to use these techniques to implement DSLs with all the usual features such as functions and procedures, data structures, pattern matching[37, 36], and polymorphism, as well as unusual features such as modal [12, 13] types.

Generic Programming. Generic programming is the ability to write one program to operate over data structures of many different types. By exploiting type representations (like `TypeR` in Figure 7) Ωmega supports generic programs such as printing (`print::TypeR t -> t -> String`) and equality functions (`equal::TypeR t -> t -> t -> Bool`). By using staging we can program generators (`equalGen :: TypeR t -> Code(t -> t -> Bool)`) that build efficient implementations without interpretive overhead.

Dynamic Typing. The need for programs that interact across a network is becoming more and more important. Programs which can perform dynamic type checks on untrusted input once, and then run in a type-safe mode thereafter will be more efficient[51] and safe than those that use continuous dynamic type checking. The use of type representations as dynamic witnesses of static properties provide a wide range of freedom in building systems which use a combination of static and dynamic checking [1].

9 Broader Impacts

The OGI School of Engineering is in a unique position to make a broad impact in the design of trustworthy software. The feature that distinguishes OGI's CSE department from other leading departments with similar levels of research activity is our close connection with industry. This is reflected not only in industrial support for our research, but also in our student body. Many of our Ph.D. students come to us from industry, and the majority of the students in our MS program are either part-time students currently employed in industry, or full- time students who have worked for some years in industry and have decided to continue their studies at a higher level. As such our students are motivated by the real problems of commercial software development, and are open to trying new and better ways of developing software.

At OGI we have recently proposed to redesign the Master of Science in Computer Science degree (as taught at OGI) to focus on constructing reliable and secure software (NSF education proposal 0417615). We are not proposing adding a few new courses, that would necessarily be electives taken by a minority of students, but a thorough re-design of our whole masters curriculum. The new curriculum will ensure that *every* student is familiar with the ideas needed to construct reliable and secure software, and has mastered the principles behind them (discrete mathematics, logic, modeling, model checking, theorem proving, programs as data, type theory, and systematic testing). Many of these key ideas are language based mechanisms, and we envision Ωmega as an integral tool for teaching them. Our industrial collaborators are making available to us powerful verification tools as well as compelling examples of how these they can be used to change the nature of software development, and Ωmega can be profitably used to script these tools. We plan to use our preliminary version of the Ωmega interpreter in our graduate course *The Design and Development of Domain Specific Languages* in the spring of 2004.

10 Conclusion

We have proposed to explore a new point in the design space for formal reasoning systems. Our choice is closer to the world of programming languages than many other reasoning systems. We see this as a positive benefit and conjecture that systems built along the proposed lines will be more widely used and thus lead to better and more trustworthy software.

The logic of the system is embedded in the type system. Semantic properties of programs, which before could only be expressed at a meta-logical level (and were thus necessarily external to the world of the programmer) can now be expressed in the programming language. While the proposed system leads to a proof construction style that is more explicit than in other systems, we believe this is an asset for programmers who are new to formal reasoning about software. The explicit nature of the proofs localizes failed proof attempts, and makes it easier to reason about the failure. , The explicit nature of the programs also minimizes the size of the state that needs exploration when failure occurs.

Advanced users can employ a reflective mechanism that enables intensional analysis of reflected types, and thus allows them 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. We conjecture this can lead to a system with the best features of both worlds.

References

[1] 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.

[2] 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.

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

[4] 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.

[5] R. G. Burger. *Efficient Compilation and Profile-Driven Dynamic Recompilation in Scheme*. PhD thesis, Indiana Univ. Computer Science Dept., 1997.

[6] R. G. Burger and R. K. Dybvig. An infrastructure for profile-driven dynamic recompilation. In *Proc. International Conf. on Computer Languages*, pages 240–249. IEEE, 1998.

[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] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. A bytecode-compiled, type-safe, multi-stage language. Technical report, Rice University, 2002.

[9] 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.

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

[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 temporal logic approach to binding-time analysis. In E. Clarke, editor, *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey*, pages 184–195. IEEE Computer Society Press, July 1996.

[13] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *23rd Annual ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 258–270, St. Petersburg Beach, January 1996.

[14] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A language for efficient, machine-independent dynamic code generation. In *Proc. POPL '96*, pages 131–144. ACM, 1996.

[15] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.

[16] Jean-Yves Girard. *Proofs and types*, volume 7 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 1989.

[17] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248:147–199, 2000.

[18] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proc. PLDI '99*, pages 293–304, 1999.

[19] Timothy G. Grifin, Aaron D. Jaggard, and Vijay Ramachandran. Design principles of policy languages for path vector protocols. In *Prooceedings of the annual meeting of the ACM Special Interest Group on Data Communication (SIGCOMM'03)*, pages 61–72. ACM, August 2003.

[20] 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.

[21] 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.

[22] 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".

[23] M. Kaufmann and J. Moore. Design goals of acl. Technical Report 101, Computational Logic, Inc., August 1994.

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

[25] 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.

[26] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *Proc. PLDI '96*, pages 137–148, 1996.

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

[28] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, Departement of Computer Science, Universiteit Utrecht., 2001.

[29] M. Leone and P. Lee. Dynamic specialization in the Fabius system. *ACM Computing Surveys*, 30, 1998.

[30] Michael Lowry, Andrew Philpot, Thomas Pressburger, and Ian Underwood. Amphion: Automatic programming for scientific subroutine libraries. *NASA Science Information Systems Newsletter*, 31:22–25, 1994.

[31] MetaOCaml Hompage. Available online from `http://cs-www.cs.yale.edu/homes/taha/metaocaml/`.

[32] E. Moggi, W. Taha, Z. Benaissa, and T. Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.

[33] 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.

[34] 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.

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

[36] Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from `ftp://cse.ogi.edu/pub/tech-reports/README.html`.

[37] Emir Pasalic and Tim Sheard. Meta-programming with typed object-language representations. Technical report, OGI, 2004.

[38] 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.

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

[40] 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/.

[41] 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.

[42] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM TOPLAS*, 21:324–369, 1999.

[43] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

[44] John C. Reynolds. An introduction to logic relations and parametric polymorphism. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 155–156. ACM, January 1993.

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

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

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

[48] T. Sheard, Z. Benaissa, and E. Pasalic. Dsl implementation using staging and monads. In *Second Conference on Domain-Specific Languages (DSL'99)*, Austin, Texas, October 1999. USEUNIX.

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

[50] Tim Sheard. Accomplishments and research challenges in meta-programming. In Walid Taha, editor, *Proceedings of the Workshop on Semantics, Applications and Implementation of Program Generation (SAIG'01)*, volume 2196 of *LNCS*, pages 2–44, Berlin, September 2001. Springer Verlag. Invited talk.

[51] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic typing through staged type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 289–302, January 1998.

[52] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Proc. ICFP 2003*, pages 263–274. ACM, 2003.

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

[54] João Luís Sobrinho. Network routing with path vector protocols: Theory and applications. In *Prooceedings of the annual meeting of the ACM Special Interest Group on Data Communication (SIGCOMM'03)*, pages 49–60. ACM, August 2003.

[55] 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.

[56] 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.

[57] 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.

[58] Walid Taha. Tag elimination - or - type specialisation is a type-indexed effect. In *APPSEM Workshop on Subtyping & Dependent Types in Programming. Ponte de Lima Portugal.*, July 2000. Available online at http://www-sop.inria.fr/oasis/DTP00/Proceedings/proceedings.html.

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

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

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

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

[63] 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.

[64] PL Wadler. Theorems for free! In MacQueen, editor, *Fourth International Conference on Functional Programming and Computer Architecture, London*. Addison Wesley, 1989.

[65] 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.

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

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

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

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

[70] 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.