Playing with Type Systems:
Automated assistance in the design of programming languages
Revision 2

Tim Sheard

Portland State University

sheard@cs.pdx.edu

Abstract

We introduce the programming language Ωmega, and illustrate how it can be used as a general purpose meta-language. Ωmega has a sophisticated type system built on top of Generalized Algebraic Datatypes and an extensible kind system. It allows users to encode object-level types in the meta-level data structures that represent object-level terms. We show Ωmega can be used to explore language designs interactively, by constructing both static and dynamic semantics as Ωmega programs. These programs can be seen as partially checked proofs of soundness of the system under exploration. As a large example we present a new type system for MetaML, that is simpler than previous type systems.

1. Introduction.

It has become common practice when designing a new language to create both a static semantics (type system) and a dynamic semantics and to prove the soundness of the type system with respect to the dynamic semantics. This process is often exploratory. The designer has an idea, the approach is analyzed, and hopefully the consequences of the approach are quickly discovered. Automated aid in this process would be a great boon.

The ultimate goal of this exploratory process is a type system, a semantics, and a proof. The proof witnesses the fact that *well-typed programs do not go wrong* [17] for the language under consideration. Approaches include a subject reduction proof in the style of Wright and Felleisen[43] on a small step semantics, or by using denotational approaches. In either case, proofs require an amazing amount of detail and are most often carried out by hand, and are thus subject to all the foils of human endeavors. It has long been a personal goal of the author to develop a generic meta-language that could be used for exploring the static and dynamic semantics for new object-languages[21, 28] that could aid in the generation of such proofs. Many others have had similar desires and the development of systems like Twelf[23] and Coq[41] attest to the broad appeal of these ideas. But, while the author owes much to these systems for inspiration, he has always desired a system whose use was closer in style to the use of a programming language than existing systems. Hence the development of the language Ωmega. This paper reports on our experience using the language Ωmega[29, 30, 31, 32] as just such a meta-language. In this paper we show that:

- Much of the work of exploring the nuances of a type system for a new language can be assisted by using mechanized tools – a generic meta-language. We call this exploration playing with type systems.
- Such tools need not be much more complicated than your favorite functional language (Haskell), and are thus within the reach most language researchers. After all playing should be child's work.
- The automation helps language designers visualize the consequences of their design choices quickly, and thus helps speed the design process.
- The artifacts created by this exploration, while not quite proofs in the full sense, are checked by machine, and are hence less subject to error than pencil and paper proofs constructed by hand.

In addition to the broad goals about meta-language design, the author has also devoted much energy to searching for a simple elegant type system for a multi-stage language such as MetaML. Such a system should not throw away too many good programs[18, 40, 36] (thus losing much of its usefulness) or be overly complex[3, 37, 35]. After many attempts, the search has been fruitful. Thus, this paper has two purposes. First, it demonstrates the efficacy of the language Ωmega as a first step towards the design of a general purpose meta-language. Second, it illustrates that a simple elegant type system for a multi-stage language is possible. The approach is so general, that we believe it can be applied to any multistage language, and as such is of wide spread interest to the program generation community.

The paper is divided into three parts. First, in Section 2 we introduce the language Ωmega. Second, in Section 3 we discuss the multi-staged language MetaML, and explain why it is so hard to develop a sound type system for a staged language. In this section we provide several small programs that illustrate the multiple causes of soundness errors in a staged language. Third, in Section 4 we play with type systems. In this section we introduce our static and dynamic semantics for a multi-stage language as an Ωmega program. We demonstrate that the ability to explore issues is greatly enhanced by the machine assistance supplied by Ωmega, and that the artifacts produced consist of machine checked proofs.

2. The Ωmega meta-programming language.

Haskell makes a fair meta-language. Its support for abstraction and first class functions make it an excellent tool for defining object-language programs as Haskell data, and for defining meta-language manipulations as Haskell functions. Unfortunately, its type system is too weak to be a true generic meta-programming system. In particular Haskell lacks the facilities to define and enforce type-systems for object-language programs represented as meta-language data. To overcome this deficiency we have defined the language Ωmega.

- Ωmega is based on Haskell so that it is easy for Haskell programmers to learn.
- We started by adding some features and removing others. To make our first attempt simple we have removed the class system of Haskell, and made Ωmega strict but pure. We have tried hard to keep all the other features of Haskell not affected by these choices intact.
- We added small (backward compatible) features that support programs that use a form of refinement or dependent types. We tried hard not to disturb the functional programming style – in particular the phase distinction between values and types. The features we added are Generalized Algebraic Datatypes (GADTs) and an extensible kind system.
- We built a non-trivial implementation and have programmed up a wide array of examples from the literature.

2.1 Generalized Algebraic Datatypes

The key to this style of programming is the use of Generalized Algebraic Datatypes (GADTs), a generalization of the normal Algebraic Datatypes (ADT) available in functional programming languages such as Haskell, ML, and O'Caml. Implementing GADTs in a functional language requires only a small, backward compatible, change to the ADT notion and is easily understood by functional programmers. A language with GADTs can support several closely related concepts such Refinement Types[11, 46, 8], Guarded Recursive Datatype Constructors[44], Inductive Families[7, 10], First-class phantom types[5], Silly Type Families[1], and Equality Qualified Types[29, 32]. There are many examples of the usefulness of such concepts in the recent literature[2, 4, 12, 20, 24, 26, 45].

ADTs generalize other forms of structuring data such as enumerations, records, and tagged variants. For example, in Ωmega (and in Haskell) we write:

```
 -- An enumeration type
data Major = CompScience | English | Chemistry

type Number = Int
type Title = String
 -- A record structure
data Class = MakeClass Number Title

-- A tagged Variant
data Person = Teacher [Class]  | Student Major
```

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

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

This declaration defines the polymorphic `Tree` type constructor. Example tree types include `(Tree Int)` and `(Tree Bool)`. In fact the type constructor `Tree` can be applied to any type whatsoever. Note how the constructor functions (`Fork`, `Node`) and constants (`Tip`) are given polymorphic types.

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

When we define a parameterized algebraic datatype, the syntactic formation rules of the `data` declaration 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*. We can remove this restriction *semantically* by strengthening the type system of the language. The restriction is *syntactically* removed by the following mechanism. In a `data` declaration, Rather than supplying the type arguments to the type being defined, the user supplys an explicit kind declaration; and rather than leaving the range of the constructor functions implicit, the user replaces the enumeration of the constructors and the type of their domains with a full explicit typing of each constructor. The only restriction being that the range must be some instance of the type being defined. For example we could define the `Tree` type as follows using the new syntax.

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

It is not necessary to use the new syntax, since `Tree` meets the restriction. As we will see below, there exist many useful types that do not. Removing the restriction requires new type checking rules that are beyond the scope of this paper, but which have been well studied[13, 44, 5].

2.2 Representing Object-Programs with Types as Data

This simple extension allows us to build datatypes representing object-programs, whose meta-level types encode the object-level types of the programs represented. A very simple object-language example with types is:

```
data Term :: *0 ~> *0 where
  Const :: a -> Term a
  Pair :: Term a -> Term b -> Term (a,b)
  App :: Term(a -> b) -> Term a -> Term b
```

Above we introduced the new type constructor `Term`, which is a representation of a simple object-language of constants and pairs. `Term` is a type constructor. It must be applied to another type to create a valid new type of kind `*0` (star-zero). We say `Term` is classified by the kind `*0 ~> *0`. The kind `*0` classifies all types that classify computable values. For more on kinds, see Section 2.3. No restriction is placed on the types of the constructors except that the range of each constructor must be a fully applied instance of the type being defined, and that the type of the constructor as a whole must be classified by `*0`. Note how the range of `Pair` is a non-trivial instance of `Term`.

As promised, `Terms` are a typed object-language representation, i.e. a data structure that represents terms in some object-language. The meta-level type of the representation (`Term a`), indicates the type of the object-level term (a). This is made possible only by the flexibility of the GADT mechanism. Using typed object-level terms, it is impossible to construct ill-typed term representations, because the meta-language type system enforces this constraint.

```
ex1 :: Term Int
```

```
ex1 = App (App (Const (+)) (Const 3)) (Const 5)

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

Attempting to construct an ill-typed object term, like `(App (Const 3) (Const 5))`, causes a meta-level (Ωmega) type error.

Another advantage of using GADTs rather than ADTs is that it is now possible to construct a tagless[21, 39, 38] interpreter directly. An interpreter is a function that takes a term to the value which is the meaning of that term. In a language without GADTs terms are untyped, and values are some universal value domain like: `data V = IntV Int | PairV V V | FunV (V -> V)`, and the interpeter is said to be tagged, since each value must be tagged with one of the constructors of the universal domain (such as `IntV`, `PairV`, or `FunV`). See [20] for a detailed discussion of this phenomena. But using GADTs none of this is necessary. Consider:

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

Here the universal domain is not necessary, and the tagless interpreter has the structure of a denotational semantics. Because the `eval` function is total and well-typed at the meta-level, it also implies that the object-level semantics (defined by `eval`) is also well-typed. As long as `eval` is total, *every* well-typed object level term evaluates to a well-formed value.

While we worked hard to make this look like Haskell programming, there are some key differences. First, the prototype declaration (`eval :: Term a -> a`) is required, not optional. Functions which pattern match over GADTs can be type checked, but type inference is much harder (see [33] for work on how this might be done). Functions that don't pattern match over GADTs can have Hindley-Milner types inferred for them (see [13] for how this mixture of type-checking and type-inference is done). Requiring prototypes for only some functions should be familiar to Haskell programmers because polymorphic-recursive functions already require prototypes[14].

2.3 Values, Types, Kinds, and Sorts

In Haskell, values are classified by types. In a similar fashion, types can be classified by kinds, and kinds can be classified by sorts. We indicate clasification as a relation using the infix symbol(`::`). We say (`::`) is overloaded because the same notation can be used in all three contexts: *value*`::`*type*, and *type*`::`*kind*, and *kind*`::`*sort*.

Some concrete examples at the value level include: `5::Int`, and `[True]::[Bool]`. We say `5` is classified by `Int`. At the type level: `Int::*0` , `Term:: *0 ~> *0`. We say `Int` is classified by star-zero, and `Tree` is classified by star-zero to star-zero. `*0`, and `*0 ~> *0` are kinds. At the kind level both: `*0:: *1` and `(*0 ~> *0):: *1`. Here, `*1` is a sort.

The kind `*0` is interesting because it classifies all types that classify values (things we actually can compute). For example, `Int:: *0`, and `[Int]:: *0`, but not `Tree:: *0`, because there are no values of type `Tree` (but there are values of type `Tree Int`).

New kinds are introduced by the *kind* declaration that also introduces the type constructors that produce types classified by that kind, just as *new types* are introduced by the *data* declaration along with the value constructors that produce values classified by that type. The `data` and `kind` declarations introduce similar structures, but at different levels in the type hierarchy. Two interesting kind declarations, of interest in this paper follow:

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

value name space		type name space				
value		type		kind		sort
5	::	Int	::	*0	::	*1
[2]	::	[Int]	::	*0	::	*1
		[]	::	*0 ⇝ *0	::	*1
				Nat	::	*1
		Z	::	Nat	::	*1
		S	::	Nat ⇝ Nat	::	*1
		Nat'	::	Nat ⇝ *0	::	*1
Z	::	Nat' Z	::	* 0	::	*1
S	::	Nat' m → Nat' (S m)	::	*0	::	*1
		Term	::	*0 ⇝ * 0	::	*1
Const	::	a → Term a	::	*0	::	*1
				Row	::	*1 ⇝ *1
		RNil	::	Row k	::	*1

Figure 1. Values, types, kinds, and sorts, are the first few levels of the infinite classification hierarchy. As is done in Haskell, there are two name spaces. One name space, names values, and the other name space, names types, kinds, sorts etc. Two objects can have the same name if they live in different name spaces. This form of overloading is often exploited in Ωmega (as well as in Haskell) programs. Note that `Z` and `S` are defined in both the value name space and the type name space.

The `Nat` declaration introduces the kind `Nat` and two new type constructors `Z` and `S` which encode the natural numbers at the type level. 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 a kind classified by the sort `*1`. Both `Nat` and `*0` are kinds at the same "level" — they are both classified by `*1`.

```
kind Row (x :: *1) = RNil | RCons x (Row x)
```

`Row`s are list like structures at the type level, which are constructed by the type constructors `RNil` and `RCons`. For example `(RCons Int (RCons Bool RNil))` and `(RCons Z RNil)`. Such types are kinded by `Row`. The kind `Row` is a higher order kind, and is classified by `(*1 ~> *1)`. Thus `Row` must be applied to a kind to be well formed (the argument indicates the kind of the types stored in the row). For example if we classify the examples above we get:

```
(RCons Int (RCons Bool RNil)):: Row *0
(RCons Z RNil):: Row Nat
```

Both `*1` and `(*1 ~> *1)` are kinds classified by the sort `*2`.

We illustrate the relationship between values, types, kinds, and sorts by example in a table found in Figure 1. Note that the table has empty slots. Not all types classify values. For example, there are no values of type `[]` (list) or `Z`, but there are values of type `[Int]`. The same holds at the kind level. Not all kinds classify types. For example, there are no types classified by `Row`, but `RNil` could be classified by `(Row Int)`. Note the different kinds of arrows (→ and ⇝). The first is used to classify functions at the value level. The second is used to classify type constructors. In Ωmega programs we write `->` for →, and `~>` for ⇝.

2.4 Singleton Types

It is sometimes useful to build representations of types at the value level. Such representations are called *singleton types* if they are encoded by a type constructor whose argument indicates the type being represented. For example consider:

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

Values classified by the type `(Nat' a)` are reflections of the *types* classified by the *kind* `Nat`. The value constructors of the `data` declaration for `Nat'` mirror the type constructors in the `kind` declaration of `Nat`, and the type index of `Nat'` is equal to the type reflected. For example, the value `S(S(S Z))` is classified by the type `Nat'(S(S(S Z)))`. We say that `Nat'` is a *singleton type* because there is only one element of any singleton type. For example, only `S (S Z)` inhabits the type `Nat' (S (S Z))`. As discussed in Figure 1, we exploit the separate name spaces for value and types by using the same names for the type constructors of kind `Nat` (`S` and `Z`) and the constructor functions of `data` type `Nat'` (`S` and `Z`) to emphasize the close relationship between `Nat` and `Nat'`.

We will find natural numbers at the type level (and their `Nat'` reflections at the value level) to be so useful we introduce some syntactic sugar for constructing such types (or values). For example, `#0 = Z`, and `#1 = (S Z)`, and `#2 = (S (S Z))` etc. We also support the syntax `#(n + x) =` $(S_1(S_2 \ldots (S_n\ x)))$. This syntactic sugar is analogous to the use of square brackets to describe lists in addition to the use of the constructor `(:)`.

2.5 Tags and Labels

Many object languages have a notion of name. To make representing names in the type system easy we introduce the notion of Tags and Labels. As a *first approximation*, consider the finite kind `Tag` and its singleton type `Label`:

```
kind Tag = A | B | C

data Label:: Tag ~> *0 where
  A:: Label A
  B:: Label B
  C:: Label C
```

Here, we again deliberately use the value/type name space overloading first discussed in Figure 1. The names `A`, `B`, and `C` are defined in both the value and type name spaces. They name different, but related objects in each space. At the value level every `Label` has a type index that reflects its value. I.e. `A::Label A`, and `B::Label B`, and `C::Label C`. Now consider a countably infinite set of tags and labels. We can't define this explicitly, but we can build such a type as a primitive inside of Ωmega. At the type level, every legal identifier whose name is preceded by a back-tick (`‘`) is a type classified by the kind `Tag`. For example, the type `‘abc` is classified by `Tag`. At the value level, every such symbol `‘x` is classified by the type `(Label ‘x)`.

2.6 Rows, Records, and HasType.

The kind `Row` classifies list-like data structures at the type level. The kind `HasType` classifies pairs at the type level.

```
kind HasType = Has Tag *0
```

It aggregates a `Tag` and any type classified by `*0`. For example, `(Has ‘a Int)::HasType`. We can construct lists (at the type level) of such pairs using the type constructors of `Row`. For example,

```
(RCons (Has ‘a Int) (RCons (Has ‘b Bool) RNil))
  ::Row HasType
```

Note, `(RCons (Has ‘a Int) (RCons (Has ‘b Bool) RNil))` is a type, and that `(Row HasType)` is a kind. Such a type can be thought of as classifying records at the value level. We can define such records within Ωmega as follows:

```
data Rec:: Row HasType ~> *0 where
  RecNil :: Rec RNil
  RecCons:: Label s -> t ->
            Rec r ->
            Rec (RCons (Has s t) r)
```

We can construct records by using the constructor functions `RecNil` and `RecCons`. Such values have types (`Rec` *r*) where `r` is classified by `(Row HasType)`. For example, consider

```
r1 :: Rec(RCons (Has ‘x Int)
                (RCons (Has ‘a Bool) RNil))
r1 = RecCons ‘x 5 (RecCons ‘a True RecNil)

r2:: Rec u -> Rec(RCons (Has ‘x Int)
                        (RCons (Has ‘a Bool) u))
r2 x = RecCons ‘x 5 (RecCons ‘a True x)
```

It is interesting to note that we have managed to express a simple form of Wand's (or Remy's) row-polymorphism[42, 27] in Ωmega just by using kinds. We have found `Row` and `HasType` so useful we have built special syntactic sugar for printing them. For example, `Rec(RCons (Has ‘x Int) (RCons (Has ‘a Bool) RNil))` prints as `Rec {‘x:Int,‘a:Bool}`. The syntactic sugar for `Row` and `HasType` replaces `RCons` and `RNil` with squiggly brackets, and replaces `Has` with colon. A type classified by `Row` whose (ultimate) tail is not `RNil` (i.e. a type variable) prints with a trailing semi-colon. For example,
`Rec(RCons (Has ‘x Int) (RCons (Has ‘a Bool) w))` prints as `Rec {‘x:Int,‘a:Bool; w}`.

2.7 An Object Language with Binding.

Rows and records allow us to define object-languages with binding structures that track their free variables in their meta-level types. The object-language `(Lam env t)` represents the simply typed lambda calculus.

```
data Lam:: Row HasType ~> *0 ~> *0  where
  Var      :: Label s -> Lam (RCons (Has s t) env) t
  Shift    :: Lam env t ->
              Lam (RCons (Has s q) env) t
  Abstract :: Label a ->
              Lam (RCons (Has a s) env) t ->
              Lam env (s -> t)
  Apply    :: Lam env (s -> t) ->
              Lam env s -> Lam env t
```

The first index to `Lam`, `env` is a Row tracking its variables, and the second index, `t` tracks the object-level type of the term. For example, a term with variables `x` and `y` might have type `Lam {‘x:Int, ‘y:Bool; u} Int`. This is made possible by the use of `Row` and `HasType` in the GADT representing lambda terms.

The key to this approach is the typing of the object-language constructor functions for variables and lambda expressions. Consider the `Var` constructor function. To construct a variable we simply apply `Var` to a label, and its type reflects this. For example, here is the output from a short interactive session with the Ωmega interpreter.

```
prompt> Var ‘name
(Var ‘name)::
forall a (u:Row HasType) . Lam {‘name:a; u} a

prompt> Var ‘age
(Var ‘age)::
forall a (u:Row HasType) . Lam {‘age:a; u} a
```

Variables behave like Bruijn indices. Variables created with `Var` are like the natural number 0. A variable can be lifted to the next natural number by the successor operator `Shift`. To understand why this is useful consider that the two examples have different names in the same index position. The two variables would clash if they were both used in the same lambda term. To shift the position of variable to a different index, we use the `Shift:: Lam u a -> Lam {v:b; u}` a constructor. Rather than counting with natural numbers (as is done with de Bruijn indices) we "count" with rows, recording both its symbolic name and its type. Here is how we could define two variables `x` and `y` for use in the same environment.

```
x :: Lam {‘x:a; u} a
x = Var ‘x

y :: Lam {u:a,‘y:b; v} b
y = (Shift (Var ‘y))
```

The type system now tracks the variables in an expressions.

```
z :: Lam {‘x:a -> b,‘y:a; u} b
z = (Apply (Var ‘x) (Shift (Var ‘y)))
```

Finally, and of great interest, we can build a well-typed evaluator for the GADT `Lam`.

```
evalLam :: Lam env t -> Rec env -> t
evalLam (Var _) (RecCons _ y _) = y
evalLam (Shift x) (RecCons _ _ rs) = evalLam x rs
evalLam (Abstract l _ body) rs =
   \ x -> evalLam body (RecCons l x rs)
evalLam (Apply f x) rs =
   (evalLam f rs) (evalLam x rs)
```

We declare that the type of our evaluation function is as follows: (`Lam env t -> Rec env -> t`). We can interpret this to mean that every well typed `Lam` term with type `t` under an environment with shape `env`, can be given meaning as a function from a record with shape `env` to `t`. The function `evalLam` is a denotational semantics. It provides a meaning for every well formed lambda term.

In essence, the well-typing of the evaluation function is one of three parts that comprise a proof of soundness of the type system with respect to the semantics. The other two parts are proofs of totality and compositionality.

- **Totality.** To ensure that every term is mapped to a well-typed value, we must ensure that `evalLam` is total. That is, it terminates for every well-typed lambda term. Every well-typed `Lam` term matches one of the clauses of `evalLam`, and every recursive call of `evalLam` is called on a smaller subterm of the original argument, so every call will terminate with a value if the input term is finite and the meta-language (in this case Ωmega) is strongly normalizing. Note that the input to `evalLam` is a meta-language term, so if the meta-language is strongly normalizing no infinite inputs are possible. The key is a strongly normalizing meta-language.
- **Compositionality.** The meaning of every term is composed only from the meaning of its subterms.

While neither of these two are currently enforced by Ωmega, we have plans to provide such support in the future.

Other kinds of proofs are also possible. Proofs in style of Wright and Felleisen[43] work on a small step semantics defined in terms of substitution. Ωmega can also be used to define languages in this way. See our paper *Meta-programming with Built-in Type Equality*[32] which was presented at the Logical Frameworks and Meta-Languages Workshop to see how this can be done.

2.8 The bottom line.

The ability to define GADTs, and the ability to define new kinds, creates a rich playground for those wishing to explore the design of new languages. These features, along with the use of rank-N polymorphism (which we will illustrate by example later in the paper) make Ωmega a better meta-language than Haskell. In order to explore the design of a new language one can proceed as follows:

- First, represent the object-language as a type indexed GADT. The indexes correspond to static properties of the program.
- The indexes can have arbitrary structure, because they are introduced as the type constructors of new kinds.
- The typed constructor functions of the object-language GADT define a static semantics for the object language.
- Meta programs written in Ωmega, manipulate object-language represented as data, and check and maintain the properties captured in the type indexes by using the meta-language type system. This lets us build and test type systems interactively.
- A dynamic semantics for the language can be defined by (1) writing either a denotational semantics in the form of an interpreter or evaluation function, or by (2) writing a small step semantics in terms of substitution over the term language. In either case, the type system of the meta-language guarantees that these meta-level programs maintain object level type-safety.

3. The MetaML language.

MetaML is a homogeneous, manually annotated, run-time code generation system. In MetaML we use angle brackets (`< >`) as quotations, and tilde (`~`) as the anti-quotation. We call the object-level code inside a pair of angle brackets, along with its anti-quoted holes a *template*, because its stands for a computation that will build an object-code fragment with the shape of the quoted code. In MetaML the angle brackets, the escapes, the `lift`s, and the `run` operator are staging annotations. They indicate the boundaries within a MetaML program where the program text moves from meta-program to object-program. The staging annotations in MetaML are placed manually by the programmer and are considered part of the language. In MetaML the staging annotations have semantic meaning, they are part of the language definition, not just hints or directions to language preprocessors. A simple example using templates follows:

```
-| val x = <3 + 2> ;
val x = <3 %+ 2> : <int>

-| val code = <show ~x> ;
val code = <%show (3 %+ 2)> : <string>
```

In this example we construct the object-program fragment `x` and use the anti-quotation mechanism to splice it into the object-program fragment `code`. Note how the definition of `code` uses a template with a hole (the escaped expression `~x`). MetaML also statically scopes free variable occurrences in code templates. This is called *cross-stage persistence*. Variables defined in earlier stages are available for use in later stages.

```
-|  fun f x y = x + y - 1;
val f = fn  : int -> int -> int

-| val z = <f 4 5>;
val z = <%f 4 5> : <int>

-| let fun f x y = not x andalso y
   in run z end;
val it = 8 : int
```

When printing code with a lexically scoped variable, the pretty printer places the percent-sign (`%`) in front of `f` in the code template `z` to indicate that this is a statically bound object-variable (not a free object-variable). Note how the free variable `f` in the code template `z` refers to the function `f:int -> int -> int`, which was in scope when the template was defined, and not the function `f:bool -> bool -> bool` which was in scope when `z` was run. Variables in MetaML are lexically scoped even across code brackets. Because code may be spliced or run in a context far removed from the scope where it was defined, this lexical scoping becomes important.

The `run` operator in MetaML transforms a piece of code into the program it represents. It is useful to think of `run` as indicating the composition of run-time compilation with execution. In the example below, we first build a generator (`power_gen`). Apply it to obtain a piece of code (`power_code`). Run the code to obtain a function (`power_fun`). And then apply the function to obtain an answer (`125`).

```
-| fun power_gen m =
     let fun f n x =
             if n = 0 then <1>
                      else <~x * ~(f (n-1) x)>
     in <let fun power x = ~(f m <x>) in power end> end;
val power_gen = fn  : int -> <int -> int>

-| val power_code = power_gen 3;
val power_code =
<let fun power x = x * x * x * 1 in power end>
: <int -> int>

-| val power_fun = run power_code;
val power_fun = fn  : int -> int

-| power_fun 5;
val it = 125 : int
```

The introduction of staging annotations such as, run, bracket, and escape considerably complicates the type system for any staged language. Most type systems disqualify bad programs that have a small class of errors. This class of errors includes, use of a variable in a scope where the variable is not declared, and use of a value in a manner inconsistent with its type – i.e. using an integer where a float is expected, or applying something that is not a function to an argument. Once staging annotations are added to a language the class of errors becomes measurably larger. New problems include tracking the stage at which a variable is declared (as well as its scope), the execution of code with free variables, and tracking alpha-renamed variables. Of course all the old problems still exist. We illustrate these three new problems with simple examples below.

- **Correct use of variables.** The staged type system helps the user construct well formed object-programs. One of its most useful features is that it tracks the level at which variables are bound. Attempts to use variables at a level lower than the level at which they are bound makes no sense. For example:

```
-| fun id x = x;
val id = Fn  : 'a  -> 'a

-| <fn x => ~(id x) - 4>;
Error: The term: x  Variable bound in stage 1
                    used too early in stage 0
```

  In the above example `x` is a stage 1 variable, but because of the anti-quotation it is used at stage 0. This is semantically meaningless and should be reported as a type error.

- **Running code with free variables.** In MetaML we use `run` to move from one stage to the next. Because it is legal to use the anti-quotation under a lambda-binding, there is the possibility that `run` may be applied to variables that will not be bound until some later stage. For example:

```
-| val bad = <fn x => ~(run <x>) + 4>;
```

  will cause an error because the object-variable `x` will not be bound until the whole piece of code is run, and then finally applied.

- **Tracking Alpha-renaming.** The third problem is very subtle. It was described in Walid Taha's thesis as *the puzzle*.

```
-| val puzzle =
   <fn a => ~((fn x => <x>)(fn y => <a>)) 0>;
```

  Using the (small step) substitution semantics for a staged language given in the thesis, the expression `(run puzzle) 5` should evaluate to `<5>`, but the implementation of MetaML raised an error. The subtlety comes because bound variables in object code in MetaML are generative. I.e. the object variable `a` in the puzzle will be alpha renamed to some new variable name, say `a_97`. The generative approach is essential to avoid inadvertent name capture in generated code. This name gets stored in the environment of the closure implementing `(fn y => <a>)`, this closure is then embedded inside the code value created by the template `<x>`, creating the object code `<fn a_97 => %x 0>` where the `%x` is a cross-stage persistent value linked to the closure. This value will only be executed when the larger enclosing code `<fn a_97 => %x 0>` is run. When the entailing function `(fn a_97 => %x 0)` is applied (to say `5`), the variable `a_97` becomes bound in the environment to `5`. Somehow this binding needs to be propagated inside the cross-stage persistent function represented by `%x`. Doing this considerably complicates the implementation, and inadversely affects its efficiency.

The kind of errors described above do not occur in normal (unstaged) programs. They complicate the semantics, the type systems, and the implementations meta-programming systems. There has been much thought put into devising type-systems which will disallow such programs [18, 3, 37, 35]. The author finds all of these systems lacking in some degree or other, even though he had a hand in devising some of them.

There has been much debate about the third example. Is it a good program? If so then the implementation must be much more complicated than it might otherwise be. If it is not a good program, this raises two questions. First, how do we get a type system to disqualify it? And, second, how do we know we won't be disqualifying important programs that real users want to write? The second objection is an easier one to answer. Had the user performed the explicit beta reduction inside the escape, i.e. written:

```
-| val puzzle = <fn a => ~(<fn y => <a>>) 0>;
```

None of the problems associated with this example would occur. So the challenge is to come up with a type system that rejects the original program but not this one. In our opinion, this is a superior solution to complicating the implementation to accommodate the original program (which we believe no one would ever deliberately write).

4. MetaML as an Ωmega program.

In this section we play with types. We explore, in Ωmega, several different formulations for a MetaML type system. We will discard several of our attempts, as our exploration points out their deficiencies. This section is meant to illustrate how Ωmega is useful for exploring language design issues. Those interested in the final MetaML type system may skip ahead to Section 5.

We will develop a type system for an abstraction of MetaML that includes all of MetaML's important features. It has a standard lambda calculus fragment, and a staging fragment. The staging fragment includes brackets, escape, run, and cross-stage-persistence.

As we did in Section 2.7 with the Lam GADT, we represent the lambda fragment with tagged de Bruijn style variables. The staging fragment is more problematic. How do we deal with the multiple levels in a staged expression? The standard solution[37, 36] is to use a level-indexed family of expressions. In Ωmega we would extend the Lam GADT as follows:

```
data Lam:: Nat ~> Row HasType ~> *0 ~> *0  where
  Var      :: Label s ->
              Lam n (RCons (Has s t) env) t
  Shift    :: Lam n env t ->
              Lam n (RCons (Has s q) env) t
  Abstract :: Label a ->
              Lam n (RCons (Has a s) env) t ->
              Lam n env (s -> t)
  Apply    :: Lam n env (s -> t) ->
              Lam n env s ->
              Lam n env t
  Bracket  :: Lam n env t -> Lam (S n) env t
  Escape   :: Lam (S n) env t -> Lam n env t
```

Note, how a natural number index is added to Lam, and that how the constructor for bracket lifts the index of a term, and how the constructor for escape drops the index of a term. It is at this point in the design exploration that the automation inherent that the meta-language begins to payoff. A well-typed term built with the constructors of Lam is a derivation of the type of that term. By entering simple expressions at the Ωmega prompt we get the Ωmega type system to check the well-formedness of the derivation, and to display the type of the term. For example, by typing: `Bracket (Apply (Var ‘f) (Shift (Var ‘x)))` we get the following result:

```
Lam #(1+u) {‘f:a -> b,‘x:a; v} b
```

Right away, an error becomes obvious in our formulation. Shouldn't an expression that is bracketed have a code type? We need typing rules some thing like the following for escape and bracket.

```
  Bracket  :: Lam n env t -> Lam (S n) env (Code t)
  Escape   :: Lam (S n) env (Code t) -> Lam n env t
```

But, what then is Code? Some thought leads to the following definition.

```
data Code t = exists env . Code (Lam Z env t)
```

Code is just a Lam term at level zero (Z) which can be typed in some environment[1]. Now, let's observe the type of the same term:

```
Bracket (Apply (Var ‘f) (Shift (Var ‘x)))
:: Lam #(1+u) {‘f:a -> b,‘x:a; v} (Code b)
```

Almost, but the environment index, `{‘f:a -> b,‘x:a; v}`, does not indicate the stage at which a variable is bound. This can be fixed by making the environment be a row of triples (rather than pairs). We introduce the kind HasT to encode triples, and adjust the definition of Var, Shift, and Abstract.

```
kind HasT = H3 Tag Nat *0
```

[1] Which environment actually matters, but will discuss this in more detail later.

```
data Lam:: Nat ~> Row HasT ~> *0 ~> *0  where
  Var      :: Label s -> Nat’ n ->
              Lam n (RCons (H3 s n t) env) t
  Abstract :: Label a ->
              Lam n (RCons (H3 a n s) env) t ->
              Lam n env (s -> t)
  Shift    :: Lam n env t ->
              Lam n (RCons (H3 s m q) env) t
```

When a variable is used, it must be applied to a singleton (`Nat’ n`) to indicate at what level it was defined. Consider again the type of the example term (adjusted to include level information on the variables).

```
Bracket (Apply (Var ‘f #0) (Shift (Var ‘x #0)))
:: Lam #1 {‘f^#0:(a -> b), ‘x^#0:a; u} (Code b)
```

We have (again) introduced some syntactic sugar. When displaying a type of kind HasT, we display `(H3 ‘tag #n typ)` as `(‘tag^#n:typ)`. A new problem arises if the two variables come from two different stages. Assume ‘f comes from stage 0, and ‘x from stage 1, as it might in the MetaML term `(fn f => <fn x => f x>)`. This makes ‘f a cross stage persistent value. We replace `(Var ‘x #0)` with `(Var ‘x #1)`. Observe the types of f and x, and the constructor Apply:

```
Var ‘f #0 :: Lam #0 {‘f^#0:a; u} a
Var ‘x #1 :: Lam #1 {‘x^#1:a; u} a
Apply :: Lam u v (a -> b) -> Lam u v a -> Lam u v b
```

As we intended, the levels of the two terms now differ. The term f is at level 0, and the term x is at level 1. The application of f to x `(Apply (Var ‘f #0) (Shift (Var ‘x #1)))` will be ill-typed, because Apply requires that the level of the two terms be the same. The problem here is similar to the problem of constructing a term with two different variables with different names but with the same de Bruijn indices. We solved that problem be introducing the Shift operator. We can solve the level problem by similar means. Introduce a new constructor of Lam terms that raises the level of a term. We call this constructor Cross, because it used when we want cross-stage persistence.

```
 Cross    :: Lam n env t -> Lam (S n) env t
```

Using Cross on f, the term is now well typed:

```
Apply (Cross (Shift (Var ‘f #0))) (Var ‘x #1)
:: Lam #1 {‘x^#1:a, ‘f^#0:(a -> b); u} (Code b)
```

The term is a second level term. This is indicated in the level index (#1). It mentions two variables f defined at level #0 with type `(a -> b)`, and x defined at level #1 with type b. The type of the term is `(Code b)`. We have succeeded in describing well-formed object-level terms using the type system of the meta-language!

Unfortunately, while an interesting exercise, this particular path is hard to extend. The problem comes from having variables from all levels in a single environment. Rhetorically, should it be necessary for the variables f and x to have different deBruijn indices? While executing in the second stage, the environment including the binding for f (as well as the need for f) will be long gone. The single environment approach also causes many problems when doing proofs. It is necessary to construct lemmas which talk about projections over environments. A projection projects only those variables defined a single stage. The key to a simple and elegant type system for a staged language is to break from tradition. Do not use an level-indexed term, and do not use a single environment.

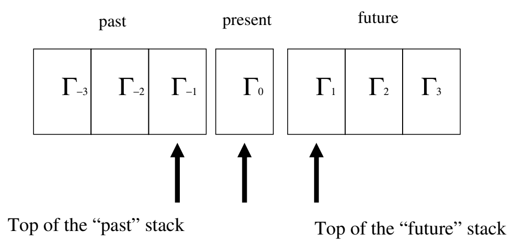

Figure 2. A typing judgment contains a "sliding band" of type contexts. The band is "implemented" using a single type context, and two stacks of type contexts. The depth of the stacks is unbounded, since a multistage program can have an arbitrary number of stages.

5. Sliding Bands of Contexts

The key to typing staged terms is to think about the environment as a "sliding band" of type contexts. The lambda fragment binds and looks up variables in the "present" stage. Staging operations slide the pointer. The structure of the environment is illustrated in Figure 2.

Capturing this structure in the object-level type system as a meta-level GADT is an interesting, though not too difficult exercise. We start by introducing a new GADT for terms `Exp`. Like `Lam`, it has multiple indexes. We write `(Exp past now future t)` where `past` and `future` are stacks of contexts, `now` is the current environment, and `t` is the type of the term.

For mnemonic purposes we would like the past stack to "grow" on the right, and the future stack to "grow" on the left. To do this we use meta-level pairs (i.e. `(a,b)`). Unfortunately, the type constructor for pairs only allows types of kind `*0` as arguments, and environments (as we have defined them) have kind `(Row HasType)`. Fortunately we also defined the type constructor `Rec` (for constructing records) which when applied to a `(Row HasType)` has kind `*0`. To push a context `(r::Row HasType)` on to a right growing stack `(past::*0)` we write `(past,Rec r)`, and to push a context `r` on to a left growing stack `future` we write `(Rec r,future)`. Keeping these explanations in mind study the GADT for expressions below:

```
data Cd n f t = Cd (forall p . Exp p n f t)

data Exp:: *0 ~> Row HasType ~> *0 ~> *0 ~> *0 where
  -- The lambda fragment
  V:: Label s -> Exp p (RCons (Has s t) env) f t
  Sh::  Exp p env f t ->
        Exp p (RCons (Has s q) env) f t
  Abs:: Label x ->
        Exp p (RCons (Has x s) n) f t ->
        Exp p n f (s -> t)
  App:: Exp p n f (t1->t) ->
        Exp p n f t1 ->
        Exp p n f t

  -- the staging fragment
  Br::  Exp (p,Rec n) c f t ->
        Exp p n (Rec c,f) (Cd c f t)
  Esc:: Exp p b (Rec n,f) (Cd n f t) ->
        Exp (p,Rec b) n f t
  Csp:: Exp p b (Rec n,f) t ->
        Exp (p,Rec b) n f t
  Run:: (forall x . Exp p n (Rec x,f) (Cd x f t)) ->
        Exp p n f t

  -- pairs and constants
  Pair:: Exp p n f s ->
         Exp p n f t ->
         Exp p n f (s,t)
  Const:: t -> Exp past now future t
```

The type `Cd` is similar to the type `Code` from Section 4. It is nothing more than a restricted form of `Exp`, and is the meta-level type used to implement object level terms with code type. In the examples below we often contrast `Lam` terms from Section 4 with `Exp` terms from this section and the distiction between `Code` (used in `Lam` terms) and `Cd` (used in `Exp` terms) is important.

The `Exp` type is how we represent object-level MetaML terms in Ωmega. The lambda fragment of the language is completely standard. It basically ignores (by keeping constant) the first and third indexes which are the stacks of contexts. Its approach to variables is the same as in the `Lam` object-language, using `Sh` to shift variables into increasing de Bruijn indices. The staging fragment is reminiscent of the level indexed `Lam` terms. But rather than changing the level index, these constructors manipulate the two context stacks. Let's compare the bracket constructor from the two approaches.

```
Bracket :: Lam u v a ->
           Lam #(1+u) v (Code a)

Br      :: Exp (a,Rec u) v b c ->
           Exp a u (Rec v,b) (Cd v b c)
```

While the `Lam` bracket constructor increments the level index, the `Exp` bracket constructor shifts the top context on the past stack (`Rec u`) into the now context, and shifts the current context (`v`) onto the top of the future stack.

```
Escape :: Lam #(1+u) v (Code a) ->
          Lam u v a

Esc    :: Exp a u (Rec v,b) (Cd v b c) ->
          Exp (a,Rec u) v b c

Cross  :: Lam u v a ->
          Lam #(1+u) v a

Csp    :: Exp a u (Rec v,b) c ->
          Exp (a,Rec u) v b c
```

A similar (but inverse) relationship holds for the escape operator. The cross-stage persistence operator, rather than changing the level of a term, cause its argument to reach into the top most past context to compute its type. Lets observe the type of our example from the previous section in both systems.

```
Apply (Cross (Shift (Var ‘f #0))) (Var ‘x #1)
:: Lam #1 {‘x^#1:a, ‘f^#0:(a -> b); u} b

App (Csp(Sh(V ‘f))) (V ‘x)
:: Exp (a,Rec {u:b,‘f:c -> d; v}) {‘x:c; w} e d

App (Csp (V ‘f)) (V ‘x)
:: Exp (a,Rec {‘f:c -> d; v}) {‘x:c; w} e d
```

Notice how the environment of the two stages is now completely separate. Both `f` and `x` could live at the same de Bruijn index since the two environments are disjoint. The third term illustrates what happens when we de Bruijn shift neither variable. If we level shift the last expression by wrapping the whole thing inside a bracket we obtain.

```
Br(App (Csp (V ‘f)) (V ‘x))
:: Exp a {‘f:c -> d; v}
         (Rec {‘x:c; w},e)
         (Cd {‘x:c; w} e d)
```

Notice how the Br shifts the top context in the past stack into the current environment, and makes the type of the resulting expression have code type.

The last staging fragment constructor is Run. The typing rule for run is intimately tied with the declaration for code. As in the level indexed example, code is should be nothing more than an expression at level 0 typed in some environment.

```
data Cd n f t = Cd (forall p . Exp p n f t)
```

A term is at level 0 if it has no embedded escapes or cross-stage persistent sub-terms. We capture this by requiring that the past environment is universally quantifiable. To better understand this requirement, consider a term with an embedded escape versus one without.

```
App (Esc (V ‘f)) (Const 3)
:: Exp (a,Rec {‘f:Cd u b (Int -> c); v}) u b c

App (V ‘f) (Const 3)
:: Exp a {‘f:Int -> b; u} c b
```

Notice the rich structure of the past stack

```
(a,Rec {‘f:Cd u b (Int -> c); v})
```

when the escape is present, and that it is simply a universally quantified variable (a) when there are no escapes. It seems polymorphism is as useful as counting levels! Finally we can explain the type for the run construct.

```
data Exp:: *0 ~> Row HasType ~> *0 ~> *0 ~> *0 where
  ...
  Run:: (forall x . Exp p n (Rec x,f) (Cd x f t)) ->
        Exp p n f t
```

A piece of code can be run if it has no free variables. We use the same trick for testing for the lack of free variables, as we did for testing for the lack of escapes: polymorphism. A term can be run only if has code type (Cd x f t), and the current environment (x) is universally quantifiable.

Reflect for a minute on what we have just done. The meta-level types of the constructors for the object-level terms Exp specifies a complete type system for object-level terms for a multi-stage language with all the features of bracket, escape, run, and cross-stage persistence. The correspondence between typing rules and the types of the object-level constructors is one-to- one. This is illustrated in Figure 3.

5.1 An Evaluation Function.

The last step is to show that the evaluation of a well typed term does not go wrong. There are several ways to describe what evaluation means. In this section we define an interpreter just as we did for Lam in Section 2.7, we introduce an evaluation function with type: Exp past now future t -> Rec now -> t. Every well typed Exp term with type t under an environment with shape now is given meaning as a function from a record with shape now to t.

```
eval :: Exp past now future t -> Rec now -> t
eval (Const n) env = n
eval (V a)  (RecCons b y x) = y
eval (Sh e) (RecCons b x y) = eval e y
eval (App f x) env = (eval f env) (eval x env)
eval (Abs a e) env = \ v -> eval e (RecCons a v env)
eval (Pair x y) env = (eval x env, eval y env)
eval (Br e) env = Cd (bd (EnvZ env) e)
eval (Run e) env = case eval e env of
                     Cd x -> eval x RecNil
```

As always, once the type system is right, the eval function is simplicity itself. Lets consider the cases one at a time.

- **(Const n).** Constants evaluate to themselves.
- **(V a).** The value of every (unshifted) variable is stored as the first element of the record storing the environment. There is no need to check that the label a and b match, because the type system ensures that they do.
- **(Sh e).** A shifted expression ignores the first element of the record. Simply evaluate the unshifted expression in the rest of the record.
- **(App f x).** Evaluate the function part of the application. This must be a function. Apply it to the result of evaluating the argument part of the application.
- **(Abs a e).** The result of an abstraction must be a function, so we return a meta-level lambda abstraction. The body of the object-level term is well typed in an environment which maps its formal parameter to the first element of the record. So simply evaluate the body in a new record with the meta-level lambda bound value as an additional element at the front.
- **(Pair x y).** Evaluate each part of a pair, and build a pair with the result.
- **(Br e).** Build an expression that is polymorphic in the past with the function bd, then make code out of it with the constructor for code Cd. We discuss building code with bd in the next section.
- **(Run e).** The expression e must evaluate to code whose underlying Exp is typable in any environment. Thus, evaluate e, extract the code x, and evaluate that in the empty record. Any record will do, RecNil is a convenient one.

5.2 Building code.

To evaluate a code template, that is an Exp whose top-level constructor is Br, we must walk over the term removing embedded escapes and cross-stage persistent terms. Inside a bracket these two kinds of subterms are the only ones that refer to values in the current (or past) environments. We must embed these values in the code produced now, because when the resulting code is evaluated, the values in the current environment will no longer be accessible. To understand this, recall how run evaluates code in the empty environment. If we don't capture the values of variables from lower levels when the code is built, it will be too late when the code is run.

Escapes are removed by splicing in the code obtained by evaluating the term inside the escape. This is how template holes are spliced in. Cross-stage persistent term are removed by evaluating them in the current environment, and lifting the value obtained up to the code level. If all the escapes and all the cross-stage persistent terms are removed, the resulting expression will be polymorphic in the past stack. This is captured by the type of the build function: (bd :: Env a z -> Exp a n f t -> Exp z n f t). Note how the past index (z) of the Exp in the range of bd is unrelated to the type of the past stack (a) of the input Exp.

The bd function walks over an expression, copying it except when it reaches an escape or cross-stage persistent sub-term. Notice how only the cases for Esc and Csp do anything interesting.

```
data Env:: *0 ~> *0 ~> *0 where
  EnvZ:: a -> Env (b,a) c
  EnvS::  Env a b -> Env (a,c) (b,c)
```

$$\frac{(P,\Gamma);(\Gamma^{f});(F)\vdash e:\tau}{(P);\ (\Gamma);\ (\Gamma^{f},F)\vdash <e>\ :\ <\Gamma^{f};F;\tau>}\ \text{brackets} \qquad \frac{(P);\ (\Gamma^{p});\ (\Gamma,F)\vdash e\ :\ <\Gamma;F;\tau>}{(P,\Gamma^{p});\ (\Gamma);\ (F)\vdash \sim e:\tau}\ \text{escape}$$

Figure 3. Compare the type inference rules with the type of the constructors `Br` and `Esc`.

```
bd :: Env a z -> Exp a n f t -> Exp z n f t
bd env (Const n) = Const n
bd env (V z) = V z
bd env (App x y) = App (bd env x) (bd env y)
bd env (Abs a e) = Abs a (bd env e)
bd env (Pair x y) = Pair (bd env x) (bd env y)
bd env (Br e) = Br(bd (EnvS env) e)
bd env (Run e) = Run(bd env e)
bd (EnvZ env) (Esc e) =
  case eval e env of
    Cd x -> x
bd (EnvS r) (Esc e) =
  case bd r e of
    Br x -> x
    y -> Esc y
bd (EnvZ env) (Csp e) = Const(eval e env)
bd (EnvS r) (Csp e) = Csp(bd r e)
```

Not all escapes or cross-stage persistent can be removed. Those embedded inside more than the original surrounding brackets refer to values available when the code being built will be run. For example, when evaluating a term like <f ~x <g ~y>> we rebuild the term (f ~x <g ~y>). Only the first level escape `~x` should be evaluated and then spliced in the rebuilding process. In order to know which escapes (and cross-stage persistent terms) to process, brackets must be counted as `bd` crawls over a term. Counting brackets is the role of the `Env` parameter to `bd`. The number of brackets inside the original pair (that was removed in the `Br` case of `eval`) can be determined by the number of `EnvS` constructors wrapped around the current environment. Thus `(EnvS (EnvS (EnvZ env)))` means the `bd` is processing a subterm inside of two additional sets of brackets., while `(EnvZ env)` means zero additional sets of brackets.

The only interesting cases inside `bd` are the `(Br e)`, `(Esc e)`, and `(Csp e)` cases. For the `(Br e)` case simply rebuild the subterm, but wrap an extra `EnvS` term around the environment to record the fact. For the `(Esc e)` case, if the environment is `(EnvZ env)` then the subterm `e` should be evaluated, and the resulting code term spliced in. If the environment is `(EnvS env)` then at least one extra pair of brackets surrounds this term. Simply rebuild it (remembering to count down by one by removing the `EnvS`) and wrap an `Esc` around the result. An optimization is possible here. Consider a subterm inside a bracketed term like ~<e>, this could be replaced by `e`. We recognize this by comparing the rebuilt subterm. If it is itself a bracketed term, then we can apply the rule, and don't need to wrap the extra escape around the returned result. The cases for `Csp` are almost identical except we use `Const` to lift a value into a piece of code when the environment count is zero.

5.3 Well-typed MetaML programs do not go wrong

In Section 2.7 we showed that a denotational semantics for the lambda calculus was sound by showing that it was well-typed, total, and compositional.

Showing that well-typed programs do not go wrong for MetaML is more complicated. First, our evaluator is not a denotational semantics, because it is not compositional. The problematic case is the evaluation of a `Run` term, where the `eval` function is called twice.

```
eval (Run e) env = case eval e env of
                     Cd x -> eval x RecNil
```

The second call to `eval` is on a term that is not a subterm of the original term. Note if evaluating `e` from a term `(Run e)` causes an infinite sequence of `Run` subterms, the evaluator would not be total.

Can we argue the non-compositional evaluator is sound? I.e. that every well-typed term is mapped to a meta-level value with that type. To show this we must answer the question: *What's a well-typed term in MetaML?* In MetaML only level 0 terms can be evaluated i.e. As discussed in Section 4, level 0 terms have no escapes at level 0, and a term is at level 0, if and only if, its is polymorphic in its past. Thus we need argue soundness for terms polymorphic in their past.

Showing *totality* is problematic for two reasons. The first is the possibility of evaluation leading to an infinite sequence of `Runs` inside `Runs`. We are currently working on a solution in which the nesting level of `Run` is encoded as an additional type index to `Exp`. We hope to report on this in the final paper.

The second cause of non-totality is non-exhaustiveness. The function `eval` is not defined on terms built with the `Esc` or `Csp` constructors. Fortunately, code polymorphic in its past cannot contain either `Esc` or `Csp` at level 0. This leads us to the following strategy, first define a version of `eval`, called `eval0`, that can only be applied to terms polymorphic in their past.

```
eval0 :: (forall p . Exp p now future t) ->
         Rec now -> t
eval0 exp env = eval exp env
  where eval::Exp past now future t -> Rec now -> t
        eval (Const n) env = n
        ...
        eval (Run e) env =
           case eval e env of
             Cd x -> eval0 x RecNil

        bd::Env a z -> Exp a n f t -> Exp z n f t
        bd env (Const n) = Const n
        ...
```

By defining `eval` and `bd` as local functions of `eval0` It is impossible to apply `eval` to terms not polymorphic in their past, since all access to `eval` is through `eval0` which requires polymorphic terms. All recursive calls to `eval` and `Bd` (except the second one in the `Run` case in `eval`) are applied to strict subterms of the original term, so if the original terms was polymorphic, so must all these sub-terms be polymorphic. The problematic second call to `eval` in the `Run` case which is applied to the return result of applying `eval` to a subterm, can be replaced with `eval0`, because expressions inside the `Cd` constructor must also be level 0 terms. Thus both `eval` and `eval0` are total, in that they will never be called on a term for which they are not defined. While not a complete proof, we have made subtantial progress. In the next section we demonstrate that the type system distinguishes the subtle staging errors discussed in Section 3.

6. Evaluating the type system.

Section 3 introduced three programs that highlight subtle typing issues for multi-stage programs. We now evaluate the novel multi-stage type system discussed in the previous section based on how it performs on these issues.

- **Correct use of variables.** The following program exhibits a level-mismatch error, and is incorrect.

```
<fn x => ~(id x)>;
```

This program makes use of the variable x at a stage prior to the stage in which it is bound. What about the Ωmega encoding of this program?

```
Br (Abs 'x (Esc (App (Abs 'y (V 'y)) (V 'x))))
```

The interpreter gives this term the following type:

```
Exp a {'x:Cd {'x:b; u} c d; v}
      (Rec u,c)
      (Cd u c (b -> d)))
```

The variable `'x` shows up in two different environments at two different stages. Each environment in the sliding band of type contexts is a separate name space, so the two occurrences of `'x` in the Ωmega encoding aren't the same variable, as they are in the MetaML program. So while this is a "type correct" program in the Ωmega encoding, it isn't an encoding of `<fn x => ~(id x)>`.

This brings up an important point. The cross-stage persistence construct does not appear in MetaML, so there must be some syntactic sugar in the parsing stage which translates MetaML syntax into the `Exp` data-structures. This preprocessing step counts brackets, and assigns to every variable a static level at its point of definition. At each point of use, a number of cross-stage persistent annotations (`Csp`) are inserted. The correct number is the difference between the level at which a variable is defined, and the level at which it is used. If this number is negative, the program is ill-typed. So this kind of error is actually caught before type checking in the syntactic sugar pre-processing phase.

- **Running code with free variables.** The following program should be rejected because the `run` will force the evaluation of the not-yet-bound variable `x`.

```
val bad = <fn x => ~(run <x>) + 4>;
```

How does this program hold up in our Ωmega encoding?

```
|- Br (Abs 'x (Esc (Run (Br (V 'x)))))
Error: Br (V 'x) isn't polymorphic enough
Expected type: forall v.
               Exp a u (Rec v,b)
                       (Cd v b c)
   Found type: Exp a u (Rec {'x:c; w},b)
                       (Cd {'x:c; w} b c))
```

The program is rejected because the term `(Br (V 'x))` isn't polymorphic enough. The term has the type

```
Exp a u (Rec {'x:b; v},c) (Cd {'x:b; v} c b)
```

See how the variable `x` shows up in the current context of the code type. Run can be applied only to code whose current context is completely polymorphic.

- **Tracking Alpha-renaming.** Recall the test programs highlighting the subtle issues at work here.

```
-| val puzzle1 =
       <fn a => ~((fn x => <x>)(fn y => <a>)) 0>;
-| val puzzle2 =
       <fn a => ~(<fn y => <a>>) 0>;
```

We argued that `(run puzzle1)` should cause a type error, but `(run puzzle2)` shouldn't.

The Ωmega encoding behaves in exactly this way. Consider the type of the Ωmega translation of `puzzle1`.

```
Br (Abs 'a
        (App (Esc (App (Abs 'x (Br (Csp (V 'x))))
                       (Abs 'y (Br (V 'a)))))
             (Const 0) ))
:: Exp a u (Rec v,b)
           (Cd v b (c -> Cd {'a:c; v} b c))
```

It appears polymorphic enough. Note the current context is typed by the type variable `v` inside the `Cd` type. But further inspection shows that this type variable `v` also appears in the type of the code returned `(c -> Cd {'a:c; v} b c)`. Thus it isn't really polymorphic at all.

```
-| run puzzle1
Error: puzzle1 isn't polymorphic enough
Expected Type:
   (forall v. Exp a u (Rec v,b)
                      (Cd v b c))
Found Type:   Exp a u (Rec v,b)
                      (Cd v b (c -> Cd {'a:c; v}
                                           b c)))
```

The type given to `puzzle1` illustrates of the problem. The environment `v` is needed even after the function is applied. But if the user performs the beta-reduction manually, then the problem does not occur.

```
-| puzzle2
   Br (Abs 'a (App (Esc (Br (Abs 'y (Br (V 'a)))))
                   (Const 0)))
 :: Exp a u (Rec v,(Rec {'a:b; w},c))
            (Cd v (Rec {'a:b; w},c)
                  (d -> Cd {'a:b; w} c b))
-| Run puzzle2
   Run (Br (Abs 'a (App (Esc (Br (Abs 'y (Br (V 'a)))))
                   (Const 0))))
 :: Exp a u (Rec {'a:b; v},c)
            (d -> Cd {'a:b; v} c b)
```

The Ωmega function `eval` shows that if we disallow `puzzle1` we can use a very simple implementation, yet still remain safe.

7. Related work.

There are two areas of related work that should be discussed. First, type systems for multi-stage languages, and second, the design of meta-languages for specifying and reasoning about object-languages.

Type Systems. Of the many papers on type systems for multi-stage languages only two achieve the scope of the system presented in this paper (in terms of the features supported), and one should be discussed because it uses a similar, but weaker approach. They are: *Environment Classifiers*[35] by Taha and Nielsen, *Meta-Programming through Typeful Code Representation*[4] by Chiyan Chen and Hongwei Xi, and *A Modal Analysis of Staged Computation* by Davies and Pfenning.

Neither of the first two has the simplicity of the system presented in this paper, and the third covers a much simpler language. The first two employ a single type context that records the type of all variables, rather than a sliding band of type contexts. The third uses a single stack, rather than a sliding band (two stacks).

The paper *Environment Classifiers* employs a level-stratified syntax and a complicated syntactic notion called *demotion*. Demotion is a term to term transformation (performed under a set of variables) that essentially places additional cross-stage persistent annotations around the variables mentioned in the set. The notion of demotion is necessary to show that reduction preserves typing. This considerably complicates the type system.

The paper *Meta-Programming through Typeful Code Representation* employs a de Bruijn notation similar to the notation used in this paper, but relies an a complicated notion of projection. Projection, computes a new type context with only those variables from specified levels from the "master" context with information about all levels. This considerably complicates proofs about program properties, like subject reduction. Neither system approaches the simplicity or the compactness of the system presented here.

The paper *A Modal Analysis of Staged Computation* [9] provides a type system for several staged languages based on intuitionist modal logic S4. While similar is some respects to the type system for MetaML presented here it has some basic differences. First, the languages presented can only build code for closed terms. For example: `<fn x => ~(f <x>)>` could not be typed because the argument to `f` is a code term with a free variable bound in the present level (from the perspective of inside the brackets). This severely limits the code generators that can be expressed. Second, the type system uses a single stack of environments one for each level, as opposed to a sliding band (a pair of stacks of environments), this simplification is sufficient because the language lacks the escape annotation. Instead it uses a weaker *unbox* construct that cannot be placed inside of brackets, also limiting the expressiveness of the language. Finally, none of there languages provides the explicit cross operator that lifts values from previous stages into code.

Meta-languages. There are many systems that could be used as meta-languages for manipulating object-languages. They include Inductive Families[7, 10], theorem provers (Coq[41], Isabelle[22]), logical frameworks (Twelf[23], LEGO[15]), proof assistants (ALF[19], Agda[6]) dependently typed languages (Epigram[16], RSP[34]).

These systems all choose a point in the design space where types and values are indistinguishable. In Ωmega we choose a point in the design space with a strict phase distinction between types and values. We also have a strong desire to minimize type annotations and other administrative work.

Several systems choose a point in the design space closer to ours, where the distinction between types and values is preserved. We owe much to these works for inspiration, examples, and implementation techniques. These include Guarded Recursive Datatype Constructors[44], First-class phantom types[5], Wobbly types[13], and Silly Type Families[1]. There are only minor syntactic differences between these systems and our own, and we imagine they could be used in a similar manner, except for one very important thing. In these systems, type indexes are restricted to types classified by `*0`, because the systems have no way of introducing new kinds. We consider the introduction of new kinds as an important contribution of our work. Without new kinds, rows, natural numbers, and other important meta-level entities at the type level are not possible.

8. Conclusion.

The goals of this paper were three fold. First, to introduce the meta-language Ωmega. Second, to illustrate that important explorations in language design can be assisted with the correct tools. Third, to demonstrate that a sound type system for a multi-stage language can also be simple, concise, and easy to understand. We believe the system using a sliding band of type contexts can generalize to any multi-stage language and is of general interest to all who study program generation. As illustrated the type system can disqualify programs with subtle properties, and can thus simplify the semantics of multi-stage languages because they no-longer need to address programs with these subtle characteristics.

References

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

[2] 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.

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

[4] 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.

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

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

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

[8] 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.

[9] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 38(3):555–604, 2001.

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

[11] 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.

[12] 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.

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

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

[15] Zhaohui Luo and Robert Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, LFCS, Computer Science Dept., University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, May 1992. Updated version.

[16] Connor McBride. Epigram: Practical programming with dependent types. In *Notes from the 5th International Summer School on Advanced Functional Programming*, August 2004. Available at: `http://www.dur.ac.uk/CARG/epigram/epigram-afpnotes.pdf` .

[17] Robin Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Science*, 17(3):348–375, 1978.

[18] 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.

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

[20] Emir Pasalic and Nathan Linger. Meta-programming with typed object-language representations. In *Generative Programming and Component Engineering (GPCE'04)*, pages 136 – 167, October 2004. LNCS volume 3286.

[21] 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.

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

[23] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *LNAI*, pages 202–206, Berlin, July 7–10, 1999. Springer-Verlag.

[24] François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL'04)*, pages 89–98, Venice, Italy, January 2004. Superseded by [25].

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

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

[27] Didier Remy. Syntactic theories and the algebra of record terms. Technical Report 1869, Institut National de Recherche en Informatique et en Automatique, March 1993.

[28] 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.

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

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

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

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

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

[34] Aaron Stump. Imperative LF meta-programming. In *Logical Frameworks and Meta-Languages workshop*, July 2004. Available at: `http://cs-www.cs.yale.edu/homes/carsten/lfm04/`.

[35] Taha and Nielsen. Environment classifiers. In *POPL: 30th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2003.

[36] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, November 1999.

[37] 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.

[38] 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.

[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] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, July 1991.

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

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

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

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