Exploring Universe Polymorphism in Ωmega

Daniel Brown

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

`danb@cs.utexas.edu`

May 5, 2006

Abstract

Ωmega extends Haskell with novel features for practical functional programming: GADT's, extensible kinds, and type functions. With both extensible types and extensible kinds in place, there is a tendency for redundant datatype definitions; likewise for functions that operate over these structures. *Universe polymorphism* is a way to abstract over levels in the typing hierarchy, unifying these redundant constructs. In this paper, we use Ωmega's novel features to encode simplified models of Ωmega as an object language, and then use these models to begin exploring the design space for universe polymorphism in Ωmega.

1 Introduction

Types are used in programming to machine-check semantic properties of programs; they are partial correctness proofs. Type checking is important because it eliminates certain classes of run-time errors, increasing the overall reliability of software.

Static type systems come in many different forms. A more powerful type system captures stronger properties of its programs and eliminates larger classes of errors from them. Some systems, like C and C++, are *weakly typed*, allowing the user to subvert sound typing and combine data in undefined ways — often leading to program crashes or, worse, unpredictable behavior. Most other languages are *safe*: they detect errors, possibly at run-time, and prevent such undefined behavior. The goal in designing modern type systems for practical languages is to describe richer sets of program properties, thereby eliminating larger classes of run-time errors, bugs, and crashes.

One way to increase the descriptive power of an existing type system is to add *type-level programming*, a feature that few practical languages support. Type-level programming is simply the ability to describe types as computations, like values. Since types can be viewed as propositions [7], computing types allows the programmer to express more complicated propositions about a program and capture stronger properties about its semantics.

Type programming is not a new idea to theorists. Systems like Coquand and Huet's Calculus of Constructions [4] (CC), some of the systems classified by the lambda cube [1], and extensions of CC [8, 5] feature a typed lambda calculus where abstractions are allowed over arbitrary terms. In these systems, computing types is as natural and useful as computing values.

However, all of these systems include a more dramatic feature: *dependent types*. Dependent types, or *strong products*, generalize arrow types and universal quantification such that the output type of the product is a function of the input *value* — dependent types compute types from values [10]. This violates basic assumptions that many programmers make about the separation of compile-time and run-time information and, indeed, introduces difficult problems for type checking.

These basic assumptions can be summarized as, "run-time entities cannot affect compile-time entities". Cardelli calls this idea the *phase distinction* [2]. Restricted forms of dependent types exist that respect the phase distinction — they can't communicate information from run-time to compile time. For example, simply prohibiting dependent products where the input is a value and the output is a non-value respects the separation: every run-time entity is a value, so no run-time information can travel backward in time.

We are interested in practical programming systems with powerful and expressive typing. As such, we are interested in systems that respect the phase distinction and avoid the impractical consequences of full dependent typing. However, we recognize the power of dependent types and believe that successfully incorporating them with pragmatic programming techniques would dramatically increase the power of modern languages and the quality of software.

Ωmega, derived from Haskell, is a pure functional language with similar goals [12]. Omega supports type-level data structures and a limited form of type programming, but provides no way to unify commonalities between value-level and type-level programs. This violates a common dictum of software engineering: avoid code duplication.

In this paper, we explore the design of an extension to Ωmega called *universe polymorphism* (proposed for Ωmega in [14] and originally formulated in [6]) that would enable code reuse between value- and type-level programs. To do so, we use Ωmega as a meta-language to model a simplified sub-language of itself, and then evaluate different ways of incorporating universe polymorphism into the existing language design and implementation.

2 Ωmega

T. Sheard's Ωmega [12, 14, 11] is an experimental derivative of Haskell that adds novel functional programming features like GADT's, extensible kinds and type functions, and omits some complicating features like type classes. These new features support type-level programming and even some forms of dependent typing.

2.1 Generalized Algebraic Datatypes

Algebraic datatypes (ADT's) are ubiquitous in functional programming. Recently, the community has seen the advent of *Generalized Algebraic Datatypes* (GADT's) which generalize various extensions to algebraic datatypes: refinement types, guarded recursive datatypes, type families, phantom types, and equality qualified types [13].

For example, consider the list datatype:

$$\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

The range type of each constructor is fully polymorphic in the type variable a:

$$\begin{aligned} \mathit{Nil}\ &:: \forall a.\ \mathit{List}\ a \\ \mathit{Cons}\ &:: \forall a.\ a \to \mathit{List}\ a \to \mathit{List}\ a \end{aligned}$$

As a second example, consider encoding a simple term language using an ADT:

$$\textbf{data}\ \mathit{Term} = \mathit{Const}\ \mathit{Int} \mid \mathit{Fun}\ (\mathit{Int} \to \mathit{Int}) \mid \mathit{Apply}\ \mathit{Term}\ \mathit{Term}$$

$$\begin{aligned} a = \mathit{Const}\ 3\ &:: \mathit{Term} \\ f = \mathit{Fun}\ \mathit{fact}\ &:: \mathit{Term} \\ \mathit{Apply}\ f\ a\ &:: \mathit{Term} \end{aligned}$$

The object types of the terms aren't represented in the meta-types constructors, so this information is unavailable during compilation of the meta-program. This can be improved by representing the object types as a parameter to the type *Term*.

$$\begin{aligned} a = \mathit{Const}\ 3\ &:: \mathit{Term}\ \mathit{Int} \\ f = \mathit{Fun}\ \mathit{fact}\ &:: \mathit{Term}\ (\mathit{Int} \to \mathit{Int}) \\ \mathit{Apply}\ f\ a\ &:: \mathit{Term}\ \mathit{Int} \end{aligned}$$

GADT's enable this by giving explicit types to each constructor: the range of *Const* can be specified as $\mathit{Term}\ \mathit{Int}$, the range of *Fun* can be specified as $\mathit{Term}\ (\mathit{Int} \to \mathit{Int})$, etc. In fact, if the type of *Const* is generalized from $\mathit{Int} \to \mathit{Term}\ \mathit{Int}$ to $a \to \mathit{Term}\ a$, then *Fun* becomes redundant and can be omitted.

$$\begin{aligned} &\textbf{data}\ \mathit{Term} :: \star \leadsto \star\ \textbf{where} \\ &\quad \mathit{Const} :: a \to \mathit{Term}\ a \\ &\quad \mathit{Apply} :: \mathit{Term}\ (a \to b) \to \mathit{Term}\ a \to \mathit{Term}\ b \end{aligned}$$

Defining a GADT also requires assigning an explicit *kind* to the type; kinds classify types just as types classify values. The type constructor *Term* is given kind $\star \leadsto \star$, typing — or *kinding* — *Term* as a one-argument type constructor. Kinds are discussed more below.

Directly encoding the object types into the meta types is advantageous because the meta-level type system checks that the object terms are well-typed! *Apply* illustrates this: $Apply\ x\ y$ is a well-typed meta-term only if the object type of x is $a \to b$ and the object type of y is a.

GADT's allow for type specialization in the range of constructors and thus better type refinement. Another form of type refinement is the use of type indexes, described below. GADT's will play a large role throughout this paper; we will use Ωmega as a meta-language to study various object languages, all of which will be encoded as GADT's.

2.2 Extensible Kinds

Datatype definitions introduce new values that can be used in run-time computations and new types to classify those values. As a logical step towards programmable types, Ωmega introduces *extensible kinds* — datatype definitions that introduce new types classified by new kinds. The key is that the introduced types play the role of data for type-level computation.

For example, consider an encoding of natural numbers at the value level:

$$\textbf{data}\ Nat = Z \mid S\ Nat$$

Ωmega's extensible kinds allow a slight syntactic change to define natural numbers at the type level instead:

$$\textbf{kind}\ Nat = Z \mid S\ Nat$$

These two definitions are identical except for the keywords **data** and **kind**. The constructors differ only in that they are classified at different levels in the hierarchy — one set with values, the other with types.

In this way, the set of type-level terms can be extended to enriching the both the typing hierarchy and the data available for type programming.

2.3 Type Functions

With GADT's and extensible kinds, Ωmega offers elaborate type-level *constructions*; to support type-level *computation*, it extends this feature set with *type functions*.

Continuing the previous example, addition over type-level natural numbers can be defined as follows:

$$\begin{aligned}
&plus :: Nat \rightsquigarrow Nat \rightsquigarrow Nat\\
&\{plus\ Z\ \ \ \ \ \ \ m\} = m\\
&\{plus\ (S\ n)\ m\} = S\ \{plus\ n\ m\}
\end{aligned}$$

This definition is analogous to a similar function on values. Notice that Ωmega requires curly braces around type function application and definition — simply a syntactic design choice.

$$\begin{aligned}
&plus :: Nat \to Nat \to Nat\\
&plus\ Z \qquad m = m\\
&plus\ (S\ n)\ m = S\ (plus\ n\ m)
\end{aligned}$$

Type functions can be used to compute types for values. For example, Consider a statically-sized list type, where the type of any list value encodes its length:

$$\begin{aligned}
&\textbf{data}\ List :: \star \rightsquigarrow Nat \rightsquigarrow \star\ \textbf{where}\\
&\quad Nil \quad :: List\ a\ Z\\
&\quad Cons :: a \to List\ a\ n \to List\ a\ (S\ n)
\end{aligned}$$

The *Cons* constructor is easily defined without type functions — it simply uses the type constructor S. But typing the *append* operation requires type-level arithmetic, and thus the type function *plus*:

$$\begin{aligned}
&append :: List\ a\ n \to List\ a\ m \to List\ a\,\{plus\ n\ m\}\\
&append\ Nil \qquad\qquad ys = ys\\
&append\ (Cons\ x\ xs)\ ys = Cons\ x\ (append\ xs\ ys)
\end{aligned}$$

The type of *append* says that the length of its output list is the sum of the lengths of its input lists.

A final note: to keep type checking tractable — and thus usable — Ωmega checks that a type function expresses a confluent and terminating set of rewrite rules [11].

2.4 Singleton Types

Extensible kinds allow the programmer to define datatypes at the level of types and kinds, effectively introducing type-level data. But this data is a world apart from the main flow of value-driven computation in a program: it can't be stored in (value-level) structures or passed to (value-level) functions. Singleton types serve to bridge the gap between value- and type-level data.

Types introduced by a new kind definition don't classify anything, but *witness* values can be created such that there is a one-to-one correspondence between the witnesses and the type data. To create this correspondence, a set of *singleton types* are created from a GADT indexed by the new kind.

For example, recall the type-level natural numbers:

$$\textbf{kind}\ Nat = Z \mid S\ Nat$$

Now consider a similar construction at the value level:

$$\begin{aligned}
&\textbf{data}\ Nat' :: Nat \rightsquigarrow \star\ \textbf{where}\\
&\quad Z' :: Nat'\ Z\\
&\quad S' :: Nat'\ n \to Nat'\ (S\ n)
\end{aligned}$$

Each value-level constructor is indexed by a corresponding type from kind Nat. Since Z' is the only value in type $Nat'\ Z$, $S'\ Z'$ is the only value in type

Nat' $(S\ Z)$, and so on, then any type $Nat'\ n$ is indeed a singleton type — it has a unique inhabitant. This establishes the one-to-one correspondence between the values Z', $S'\ Z'$, etc. and the types Z, $S\ Z$, etc. The type Nat' is called the *reflection* of the kind Nat since it effectively reflects the type-level data introduced by Nat down into the value level.

Value data in Nat' can be used to represent type data in Nat. For example, consider a function that creates a statically-sized list of a given length, simply repeating a given element throughout the list:

$$
\begin{aligned}
&nOf :: Nat'\ n \to a \to List\ a\ n\\
&nOf\ Z \qquad a = Nil\\
&nOf\ (S\ n)\ a = Cons\ a\ (nOf\ n\ a)
\end{aligned}
$$

To use this function, the programmer supplies a witness value, say $S'\ Z'$, and then its type, $Nat'\ (S\ Z)$, tells the type checker exactly which value is given. This is possible only because the witness is the sole inhabitant of its singleton type.

In essence, singleton types simulate certain kinds of dependent types. In other systems, nOf might be given a dependent type: $nOf :: \forall(a :: \star).\ \Pi(n :: Nat).\ a \to List\ a\ n$. The dependent product exposes the value of the Nat input, n, making it visible to the type being computed. In general, this visibility violates the phase distinction: values are run-time entities, whereas types are compile-time entities. But in cases like nOf, that n is a run-time entity is incidental; it could just as well be compile-time data — a type, kind or higher. Thus, in the Ωmega version of nOf above, the kind Nat lifts the value n to the type level — making it compile-time data — and singleton types enable the programmer to specify a particular type, say Z, indirectly via its witness value, Z'.

In fact, the dual value/type representation enabled by type indexes and singleton types is so useful that Ωmega includes an infinite set of built-in singleton types and witnesses — *tags* and *labels* — constructed from arbitrary Ωmega identifiers.

The kind Tag classifies the infinite set of types named by a backtick (‘) followed by a valid identifier. Thus, $`foo :: Tag$, $`Nat' :: Tag$ and $`Tag :: Tag$. The reflection of $Tag :: \star 1$ is $Label :: Tag \rightsquigarrow \star 0$. Like tags, labels are constructed with backticks and identifiers; context determines whether such syntax denotes a label (value) or a tag (type). So $`foo :: (Label\ `foo) :: Tag$. Tag and $Label$ are related in the same way as Nat and Nat', except the name $Label$ is used instead of Tag'.

2.5 Infinite Universe Hierarchy

Extensible kinds create a richer classification hierarchy than is found in most functional languages. As a result, Ωmega includes an infinite hierarchy of kinds (only the bottom few levels are actually necessary).

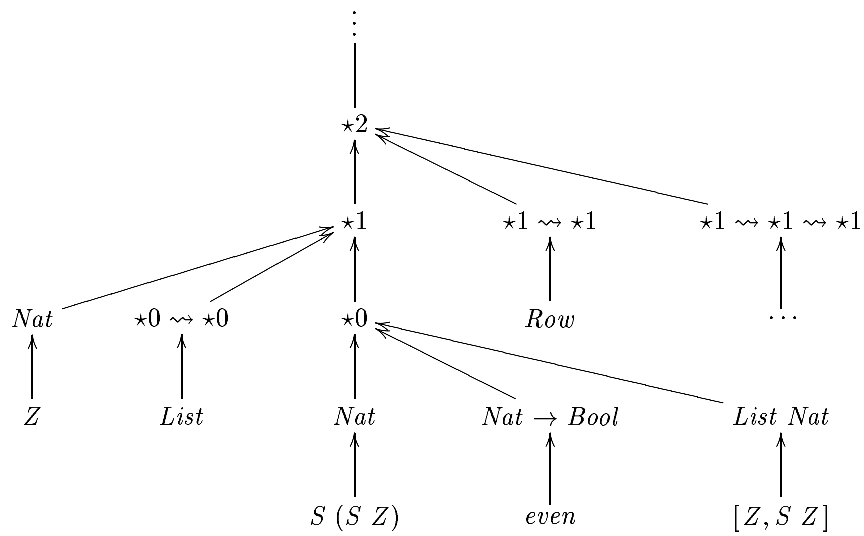

Figure 1: Universe Hierarchy

In Haskell, kinds are used to classify data constructors and can't be introduced by the programmer. Classifying data constructors is achieved with a simple set of kinds:

$\kappa ::= \star \mid \kappa \rightsquigarrow \kappa$

The kind $\star$, pronounced "star", or "type", classifies types that in turn classify values. All other kinds — those containing $\rightsquigarrow$ — classify type constructors, which in turn classify nothing. For example, *List* is a type constructor: *List Nat* classifies lists of numbers, but *List* itself classifies nothing.

Ωmega inherits these basic kinds and also allows datakind definitions to introduce new kinds and kind constructors. This requires more classification. One extra level above kinds suffices, but Ωmega goes further and establishes an infinite number of levels, each classifying the next. $\star$ is aliased as $\star 0$, which is then classified by $\star 1$, classified by $\star 2$, and so on. Figure 1 illustrates a few terms in the hierarchy and the classification relations between them.

Just as *List* has type $\star 0 \rightarrow \star 0$, a unary kind constructor for row types, *Row*, would have type $\star 1 \rightarrow \star 1$. (For simplicity, we say "type" to describe the classification relationship between any terms in the hierarchy, regardless of level.) Notice that all terms in Ωmega are classified somewhere under $\star 2$; allowing higher-level type definitions (above kinds) would change this.

2.6 Future: Universe Polymorphism

Ωmega extends the reach of computation a step up in the classification hierarchy — from values to types — but stops short of two interesting extensions:

extending computation further upward to kinds and beyond, and supporting code reuse at multiple levels in the hierarchy. We call the latter *universe polymorphism* (after [6]). It entails the former: supporting the reuse of datatypes and functions throughout the hierarchy requires supporting computation anywhere in the hierarchy.

As a first step towards universe polymorphism, this paper pursues a design for universe-polymorphic, or *universal*, datatype definitions — that is, type definitions that can be instantiated anywhere in the universe hierarchy. The key to such definitions is a way to replace the numerical indexes in the star constants $\star 0$, $\star 1$, etc. with some kind of *level variable*. Then a *level-varying* star term $\star n$ could be used to type universal datatypes. For example, consider defining a universal natural number datatype:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Nat} :: \star n\ \textbf{where} \\
\quad Z :: \mathit{Nat} \\
\quad S :: \mathit{Nat} \rightarrow \mathit{Nat}
\end{array}
$$

Since this introduces a "type" *Nat* classified by $\star n$, and $\star n$ can take on any of the values $\star 0$, $\star 1$, etc., then *Nat* can be instantiated in any universe below another containing a star term — anywhere at or above the type level. Consequently, the numbers Z, $S\ Z$, etc. can be instantiated at *any* level.

"Instantiating" a universal term simply involves using it in a non-universal context. If $\mathit{List} :: \star 0 \leadsto \star 0$, then $\mathit{List}\ \mathit{Nat} :: \star 0$, instantiating *Nat* at the type level. A value of this type, $[Z, S\ Z]$, instantiates Z and S at the value level. If instead we use statically-sized lists $\mathit{List} :: \star 0 \leadsto \mathit{Nat} \leadsto \star 0$, then *Nat* is instantiated at the kind level, and a type like $\mathit{List}\ a\ (S\ Z)$ instantiates Z and S at the type level. Clearly, the universal natural number datatype subsumes the pair of value- and type-level natural number datatypes from before.

To flex the idea a little more, consider a statically-sized list of numbers: $[Z, S\ Z] :: \mathit{List}\ \mathit{Nat}\ (S\ (S\ Z))$. Here, *Nat* is instantiated at both the type and kind levels (the kind-level *Nat* is implicit as a type for $S\ (S\ Z)$), and Z and S are instantiated at both the value and type levels. Further, if this list type is also universal, $\mathit{List} :: \star n \leadsto \mathit{Nat} \leadsto \star n$, then the applied type $\mathit{List}\ \mathit{Nat}\ (S\ (S\ Z)) :: \star n$ is still universal, only to be grounded by eventual use in a non-universal term.

Similar ideas could be explored for functions, but we leave that as future work. Adding the level-varying $\star n$ term is a first step towards universal datatype definitions and is the focus of this paper.

3 Modeling Universe Polymorphism

To think about and experiment with possible formations of universe polymorphism for Ωmega, we model a simple language using GADT's and extend it with universe polymorphism. In this way, Ωmega is used as a meta-language, with a subset of itself as the object language, to formulate key properties of its own extension. We define a GADT to represent terms in the object language and use type indexes to represent object types for those terms, thereby enforcing object-level type correctness with the meta-level type checker. This technique is originally presented by Sheard in [13].

We exploit Ωmega's type polymorphism to encode both object-language type polymorphism and level polymorphism. This way, we avoid building our own mechanisms for things like unification, substitution and fresh name generation which are necessary to implement polymorphic type checking.

However, reusing mechanisms for abstractions and name binding to encode the same in the object language isn't as immediate, so we omit them from the model. This restricts the object language to one of just constructors and constructor application. Since values inhabiting polymorphic types are another form of abstraction, they are also excluded from the constructor language. Although we can't encode polymorphic values, we can still encode their type constructors and will put them to good use. Interesting future research would be to use higher-order abstract syntax [9] (HOAS) to try to reuse abstractions and name binding mechanisms from Ωmega to complete the object language.

Even with these restrictions, the object language will help us think about and enable us to experiment with designs for universe polymorphism in Ωmega.

4 A Constructor Language

We start with a basic term language of constructors and constructor application. It is a standard model with an infinite universe hierarchy, starting with values, types and kinds. The value constructors are classified by type constructors, which are classified by the kinds $\star 0$, $\star 0 \rightsquigarrow \star 0$, etc., which are all classified by $\star 1$, classified by $\star 2$, and so on.

4.1 Language Encoding

The GADT *Value* represents value-level terms. The simplified object language has only constructors and application at the value level, each indexed by types in kind *Type′* that represent their object types. A constructor, *Con n t* is built from its name and type. The object type is a witness of the singleton type t, giving a *Con* term a representation of its type at both the value and type levels in the meta-language. The k index to *Type* is explained below.

$$
\begin{array}{l}
\textbf{data } \mathit{Value} :: \mathit{Type}' \rightsquigarrow \star 0 \textbf{ where} \\
\quad \mathit{Apply} :: \mathit{Value}\ (\mathit{Arrow}'\ a\ b) \rightarrow \mathit{Value}\ a \rightarrow \mathit{Value}\ b \\
\quad \mathit{Con} \quad :: \mathit{Label}\ \mathit{name} \rightarrow \mathit{Type}\ t\ k \rightarrow \mathit{Value}\ t
\end{array}
$$

Type indexes in the meta-language are the key to enforcing well-formedness in object-level terms. Here, the *Apply* meta-constructor ensures that term application only occurs between values classified by arrows (i.e. functions) and values in the domain of the arrow. For example, if $f :: \mathit{Value}\ (\mathit{Arrow}'\ A\ B)$ and $a :: \mathit{Value}\ A$, then *Apply f a* is well-typed in the meta-language but *Apply a a* is not. This meta-level typing embodies the intended object-level typing: it makes sense to apply the function *succ* to the value *one*, but it doesn't make sense to apply the *one* to itself.

The check enforced by *Apply* corresponds to a well-known elimination typing rule for the lambda calculus:

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash x : A}{\Gamma \vdash f\, x : B}$$

Correspondences between constructor types and typing rules will be common; indeed our goal is to experiment with GADT object-language encodings to guide the design of typing rules for universe polymorphism. Notice that they make sense for constructors involving multiple terms, like *Apply*, since these constructors govern the composition of terms in the object language. However, there are no corresponding object-language typing rules for constructors like *Con* that compose an object term only from meta-terms. These compositions are already governed by their meta-level types.

Type′ is similar to *Value* but adds arrows. Though Ωmega offers indexed types, it doesn't offer indexed kinds, so *Type′* has a simpler structure than the *Value* GADT.

$$
\begin{array}{l}
\textbf{kind}\ \mathit{Type}' \\
\quad = \mathit{Arrow}'\ \mathit{Type}'\ \mathit{Type}' \\
\quad \mid\ \mathit{TApply}'\ \mathit{Type}'\ \mathit{Type}' \\
\quad \mid\ \mathit{TCon}'\ \mathit{Tag}\ \mathit{Kind}'
\end{array}
$$

Richness lost in the kind *Type′* is regained in its downward reflection, *Type*.

Type is a standard reflection of the kind *Type′*: each term in *Type* is indexed by term in *Type′*. But since *Type* is a GADT, it is also able to enforce well-formedness constraints, so each term is indexed by a *Kind′*, as well — just as *Value* used *Type′*. Arrows ensure their domain and range have the same kind. Applications and constructors are like those in *Value*.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathit{Type} :: \mathit{Type}' \leadsto \mathit{Kind}' \leadsto \star 0\ \textbf{where}} \\
\quad \mathit{Arrow} & :: & \mathit{Type}\ a\ (\mathit{Star}'\ n) \to \mathit{Type}\ b\ (\mathit{Star}'\ n) \\
 & & \to \mathit{Type}\ (\mathit{Arrow}'\ a\ b)\ (\mathit{Star}'\ n) \\
\quad \mathit{TApply} & :: & \mathit{Type}\ a\ (\mathit{KArrow}'\ k\ l) \to \mathit{Type}\ b\ k \to \mathit{Type}\ (\mathit{TApply}'\ a\ b)\ l \\
\quad \mathit{TCon} & :: & \mathit{Label}\ \mathit{name} \to \mathit{Kind}\ k\ l \to \mathit{Type}\ (\mathit{TCon}'\ \mathit{name}\ k)\ k
\end{array}
$$

Again, constraints in the constructors correspond to typing rules for the object terms. The restriction on arrows mentioned above implies this formation rule *schema* — every choice of i generates a separate rule:

$$\frac{\Gamma \vdash A : \star_i \qquad \Gamma \vdash B : \star_i \qquad i \in \mathbb{N}}{\Gamma \vdash A \to B : \star_i}$$

The elimination rule for type application is like the one for values, except that it pertains to types and kinds instead of values and types:

$$\frac{\Gamma \vdash F : \kappa \rightarrow \kappa' \qquad \Gamma \vdash A : \kappa}{\Gamma \vdash F\, A : \kappa'}$$

$Kind'$ encodes the rest of the universe hierarchy. It consists only of stars and arrows.

$$\begin{array}{l}
\textbf{kind}\ \mathit{Kind}' \\
\quad = \mathit{Star}'\ \mathit{Nat} \\
\quad \mid\ \mathit{KArrow}'\ \mathit{Kind}'\ \mathit{Kind}'
\end{array}$$

Its reflection, $Kind$, is indexed by $Kind'$ twice — the first to witness singleton types, and the second to encode typing. Thus $Kind$ encodes kind-level terms and all terms above the kind level. As in Ωmega, a star is typed by another star from the level above. Arrows between kinds are only valid when the two kinds are at the same level.

$$\begin{array}{l}
\textbf{data}\ \mathit{Kind} :: \mathit{Kind}' \rightsquigarrow \mathit{Kind}' \rightsquigarrow \star 0\ \textbf{where} \\
\quad \mathit{Star} \quad\ :: \mathit{Nat}'\ n \rightarrow \mathit{Kind}\ (\mathit{Star}'\ n)\ (\mathit{Star}'\ (S\ n)) \\
\quad \mathit{KArrow} :: \mathit{Kind}\ a\ (\mathit{Star}'\ n) \rightarrow \mathit{Kind}\ b\ (\mathit{Star}'\ n) \\
\qquad\qquad\quad \rightarrow \mathit{Kind}\ (\mathit{KArrow}'\ a\ b)\ (\mathit{Star}'\ n)
\end{array}$$

The type assigned to $Star\ n$ implies a simple rule schema:

$$\frac{i \in \mathbb{N}}{\vdash \star_i : \star_{i+1}}$$

Constraints on kind arrows are similar to those on type arrows and correspond to a similar rule:

$$\frac{\Gamma \vdash \kappa : \star_i \qquad \Gamma \vdash \kappa' : \star_i}{\Gamma \vdash \kappa \rightsquigarrow \kappa' : \star_i}$$

Together, $Value$, $Type\ t\ k$ and $Kind\ k\ l$ form an object language in which constructors can be applied to other constructors. This doesn't sound very exciting: the benefit is that the meta-level type system does all the hard work of object-level typing by checking the constraints encoded in constructors! This allows the user to test the well-formedness of object terms simply by entering them into the Ωmega interpreter. When the type of an object term surprises the user, she can simply refine the GADT encodings.

4.2 Example Terms

As examples, here are algebraic data types for natural numbers and booleans in the new object language:

$$\begin{array}{l}
\mathit{nat} \ \ = \mathit{TCon}\ \text{`}\mathit{Nat} \quad (\mathit{Star}\ \#0) \\
\mathit{zero} = \mathit{Con} \quad \text{`}\mathit{Zero}\ \ \mathit{nat}
\end{array}$$

$$
\begin{aligned}
&succ = Con\ \ `Succ\ \ (nat\,`Arrow`\,nat)\\
&bool\ = TCon\,`Bool\ \ (Star\ \#0)\\
&true = Con\ \ `True\ \ bool\\
&false = Con\ \ `False\ bool
\end{aligned}
$$

The meta-language allows these object terms:

$$
\begin{aligned}
&true\\
&zero\\
&Apply\ succ\ zero\\
&Apply\ succ\ (Apply\ succ\ zero)
\end{aligned}
$$

But it rejects these:

$$
\begin{aligned}
&Apply\ succ\ true\\
&Apply\ succ\ succ\\
&Apply\ false\ zero
\end{aligned}
$$

Representing parameterized types is also possible, but encoding their polymorphic value constructors isn't since it requires representations for binding constructs — an object-language feature intentionally omitted. For example, here is the list type (without *nil* and *cons*):

$$list = TCon\,`List\ (Star\ \#0\,`KArrow`\,Star\ \#0)$$

The interpreter accepts these terms:

$$
\begin{aligned}
&TApply\ list\ nat\\
&TApply\ list\ (TApply\ list\ nat)
\end{aligned}
$$

But it rejects these:

$$
\begin{aligned}
&TApply\ bool\ nat\\
&TApply\ list\ list
\end{aligned}
$$

4.3 Type Assignment

So far, the values, types and kinds in the object language are only loosely related by the object-level typing rules encoded in the types of the meta-constructors. We can directly relate values to types and types to kinds by assigning a type (or kind) to each value (or type).

Computing the type assignments is mostly straightforward but requires a little machinery. It requires three functions — one each for values, types and kinds:

$$
\begin{aligned}
&typeOfV :: Value\ t\ \ \ \ \to Type\ \ t\ \ k\\
&typeOfT :: Type\ \ \ t\ \ k \to Kind\ k\ l\\
&typeOfK :: Kind\ \ \ k\ \ l \to Kind\ l\ \ m
\end{aligned}
$$

As written, these three functions are uninhabited: the output type indexes k, l and m are universally quantified, but no constructor in *Type* or *Kind* is polymorphic in these type variables — by design, they each impose some constraint to encode a typing rule. Conceptually, each of these type indexes is determined by the input type: each is the object type of the corresponding index from the input type. To express this relation, we use two type functions, $typeOfT'$ and $typeOfK'$, which are defined below.

$$\begin{aligned}
&typeOfV :: Value\ t \quad \to Type\ t\ \{typeOfT'\ t\}\\
&typeOfT :: Type\ \ t\ k \to Kind\ k\ \{typeOfK'\ k\}\\
&typeOfK :: Kind\ \ k\ l\ \to Kind\ l\ \{typeOfK'\ l\}
\end{aligned}$$

Given these typings, defining the value functions is straightforward:

$$\begin{aligned}
&typeOfV :: Value\ t \to Type\ t\ \{typeOfT'\ t\}\\
&typeOfV\ (Apply\ f\ a) \qquad = \mathbf{case}\ typeOfV\ f\ \mathbf{of}\ Arrow\ t\ u \to u\\
&typeOfV\ (Con\ name\ t) \quad = t\\
&typeOfT :: Type\ t\ k \to Kind\ k\ \{typeOfK'\ k\}\\
&typeOfT\ (Arrow\ a\ b) \qquad = typeOfT\ b\\
&typeOfT\ (TApply\ f\ a) \qquad = \mathbf{case}\ typeOfT\ f\ \mathbf{of}\ KArrow\ t\ u \to u\\
&typeOfT\ (TCon\ name\ k) = k\\
&typeOfK :: Kind\ k\ l \to Kind\ l\ \{typeOfK'\ l\}\\
&typeOfK\ (Star\ n) \qquad\qquad = Star\ (S\ n)\\
&typeOfK\ (KArrow\ a\ b) \quad = typeOfK\ b
\end{aligned}$$

Defining the type functions is almost as simple, except that Ωmega doesn't allow case expressions or conditionals in type-level computation [11]. This prevents defining the $TApply'$ case of $typeOfT'$. Although clearly a drawback of Ωmega's support for type programming, this restriction is unimportant here since the $TApply'$ case never occurs: $typeOfT'$ is only invoked on the type indexes of *Value* terms and, because the term encoding can't represent polymorphic terms, no *Value* terms are typed by applications.

$$\begin{aligned}
&typeOfT' :: Type' \rightsquigarrow Kind'\\
&\{typeOfT'\ (Arrow'\ a\ b)\} \qquad = \{typeOfT'\ b\}\\
&\{typeOfT'\ (TCon'\ name\ k)\} = k\\
&\quad\text{--}\ \{typeOfT'\ (TApply'\ f\ a)\} = \mathbf{case}\ typeOfT'\ f\ \mathbf{of}\ Arrow'\ t\ u \to u\\
&typeOfK' :: Kind' \rightsquigarrow Kind'\\
&\{typeOfK'\ (Star'\ n)\} \qquad = Star'\ (S\ n)\\
&\{typeOfK'\ (KArrow'\ k\ l)\} = \{typeOfK'\ l\}
\end{aligned}$$

Here are some examples of computing typings using *typeOfV*, *typeOfT* and *typeOfK*:

```
prompt> zero
(Con ‘Zero (TCon ‘Nat (Star #0))) : Value (TCon’ ‘Nat (Star’ #0))
prompt> typeOfV zero
```

```
(TCon 'Nat (Star #0)) : Type (TCon' 'Nat (Star' #0)) (Star' #0)
prompt> nat
(TCon 'Nat (Star #0)) : Type (TCon' 'Nat (Star' #0)) (Star' #0)
prompt> typeOfT (typeOfV zero)
(Star #0) : Kind (Star' #0) (Star' #1)
prompt> typeOfK (typeOfT (typeOfV zero))
(Star #1) : Kind (Star' #1) (Star' #2)
```

This section modeled a well-typed, semi-polymorphic constructor language, built without writing a type checker — or even a unification algorithm! But it represented the bottom three levels of the universe hierarchy separately. Since the goal of universe polymorphism is to instantiate a single term at different levels, a more suitable model will represent all levels uniformly. This is the subject of the next section.

5 Self-Typing Terms

The previous model lacks a uniform way to handle terms at different levels in the universe hierarchy, but without this uniformity, encoding universe polymorphism will be cumbersome. In this section, we flatten the encoding so that terms at all levels in the hierarchy are represented by the same GADT.

5.1 Language Encoding

The model from before represents values, types and kinds separately and uniformly encodes kinds and everything above.

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Value} :: \mathit{Type}' \rightsquigarrow \star 0\ \mathbf{where}\\
\quad \mathit{Apply} :: \mathit{Value}\ (\mathit{Arrow}\ a\ b) \to \mathit{Value}\ a \to \mathit{Value}\ b\\
\quad \mathit{Con} \quad :: \mathit{Label}\ \mathit{name} \to \mathit{Type}\ t\ k \to \mathit{Value}\ t\\
\mathbf{data}\ \mathit{Type} :: \mathit{Type}' \rightsquigarrow \mathit{Kind}' \rightsquigarrow \star 0\ \mathbf{where}\\
\quad \mathit{Arrow} \quad :: \mathit{Type}\ a\ (\mathit{Star}\ n) \to \mathit{Type}\ b\ (\mathit{Star}\ n)\\
\qquad\qquad\quad \to \mathit{Type}\ (\mathit{Arrow}\ a\ b)\ (\mathit{Star}\ n)\\
\quad \mathit{TApply} :: \mathit{Type}\ a\ (\mathit{KArrow}\ k\ l) \to \mathit{Type}\ b\ k \to \mathit{Type}\ (\mathit{TApply}\ a\ b)\ l\\
\quad \mathit{TCon} \quad :: \mathit{Label}\ \mathit{name} \to \mathit{Kind}\ k\ l \to \mathit{Type}\ (\mathit{TCon}\ \mathit{name}\ k)\ k\\
\mathbf{data}\ \mathit{Kind} :: \mathit{Kind}' \rightsquigarrow \mathit{Kind}' \rightsquigarrow \star 0\ \mathbf{where}\\
\quad \mathit{Star} \quad\ :: \mathit{Nat}'\ n \to \mathit{Kind}\ (\mathit{Star}\ n)\ (\mathit{Star}\ (S\ n))\\
\quad \mathit{KArrow} :: \mathit{Kind}\ a\ (\mathit{Star}\ n) \to \mathit{Kind}\ b\ (\mathit{Star}\ n)\\
\qquad\qquad\quad \to \mathit{Kind}\ (\mathit{KArrow}\ a\ b)\ (\mathit{Star}\ n)
\end{array}
$$

The essential ideas are applications, constructors, arrows and stars. Arrows for types and kinds embody the same idea and can be unified into a single arrow term. Similarly, applications for values and types can be unified into a single term. Since type constructors classify value constructors, they actually play different roles and are kept separate.

Value, *Type* and *Kind* are merged into *Term*, which represents the entire object language. To encode the type of each object term in its meta-constructor, we want a kind similar to *Term* that encodes the same object terms. These meta-types could them be used as indexes to the *Term* constructors. This motivates a curiously ill-formed Ωmega GADT:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Term} :: \mathit{Nat} \rightsquigarrow \underline{\mathit{Term}\ n\ t} \rightsquigarrow \star 0\ \textbf{where} \\
\quad \mathit{Star} \quad :: \mathit{Nat}'\ n \to \mathit{Term}\ (S\ (S\ n))\ (\mathit{Star}\ (S\ n)) \\
\quad \mathit{Arrow} :: \mathit{Term}\ (S\ n)\ (\mathit{Star}\ n) \to \mathit{Term}\ (S\ n)\ (\mathit{Star}\ n) \\
\qquad\qquad \to \mathit{Term}\ (S\ n)\ (\mathit{Star}\ n) \\
\quad \mathit{Apply} :: \mathit{Term}\ n\ (\mathit{Arrow}\ a\ b) \to \mathit{Term}\ n\ a \to \mathit{Term}\ n\ b \\
\quad \mathit{VCon} :: \mathit{Label}\ \mathit{name} \to \ldots \\
\quad \mathit{TCon} :: \mathit{Label}\ \mathit{name} \to \ldots
\end{array}
$$

Notice the underlined kind, *Term n t*: it doesn't exist! Conceptually, we want to recursively define the type *Term n t* using a kind with the same structure — a possible application of universe polymorphism, but not something Ωmega currently supports!

To solve this problem, we introduce a new kind $\mathit{Preterm}'$, similar to *Term*, to be used as the t index. Preterms consist of stars, arrows, and type constructors. Since these meta-types only represent object terms that will type other object terms, we omit value constructors and applications: value constructors don't classify anything and, in the restricted semi-polymorphic term language, terms classified by applications aren't representable.

$$
\begin{array}{l}
\textbf{kind}\ \mathit{Preterm}' \\
\quad = \mathit{PStar}\ \mathit{Nat} \\
\quad \mid \mathit{PArrow}\ \mathit{Preterm}'\ \mathit{Preterm}' \\
\quad \mid \mathit{PTCon}\ \mathit{Tag}\ \mathit{Preterm}'
\end{array}
$$

The reflection of $\mathit{Preterm}'$ includes a *Nat* type index to encode the level of each preterm. This information will be useful later to compute types for terms and preterms.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Preterm} :: \mathit{Nat} \rightsquigarrow \mathit{Preterm}' \rightsquigarrow \star 0\ \textbf{where} \\
\quad \mathit{PStar} \quad :: \mathit{Nat}'\ n \to \mathit{Preterm}\ (S\ (S\ n))\ (\mathit{PStar}\ n) \\
\quad \mathit{PArrow} :: \mathit{Preterm}\ (S\ n)\ a \to \mathit{Preterm}\ (S\ n)\ b \\
\qquad\qquad\quad \to \mathit{Preterm}\ (S\ n)\ (\mathit{PArrow}\ a\ b) \\
\quad \mathit{PTCon} :: \mathit{Label}\ \mathit{name} \to \mathit{Preterm}\ (S\ (S\ n))\ t \\
\qquad\qquad\quad \to \mathit{Preterm}\ (S\ n)\ (\mathit{PTCon}\ \mathit{name}\ t)
\end{array}
$$

Now the object language can be encoded using $\mathit{Preterm}'$ indexes:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Term} :: \mathit{Nat} \rightsquigarrow \mathit{Preterm}' \rightsquigarrow \star 0\ \textbf{where} \\
\quad \mathit{Star} \quad :: \mathit{Nat}'\ n \to \mathit{Term}\ (S\ (S\ n))\ (\mathit{PStar}\ (S\ n)) \\
\quad \mathit{Arrow} :: \mathit{Term}\ (S\ n)\ (\mathit{PStar}\ n) \to \mathit{Term}\ (S\ n)\ (\mathit{PStar}\ n) \\
\qquad\qquad \to \mathit{Term}\ (S\ n)\ (\mathit{PStar}\ n)
\end{array}
$$

$$\begin{aligned}
&Apply :: Term\ n\ (PArrow\ a\ b) \to Term\ n\ a \to Term\ n\ b\\
&TCon :: Label\ name \to Preterm\ (S\ (S\ n))\ t \to Term\ (S\ n)\ t\\
&VCon :: Label\ name \to Preterm\ (S\ n)\ t \to Term\ n\ t
\end{aligned}$$

This encoding subsumes the one achieved by *Value*, *Type* and *Kind*. The only surprise is that *TCon* and *VCon* accepts preterms instead of terms to specify their object types. This is because the output meta-type of each constructor depends on the value of the input object type — a dependent typing problem. Ωmega's standard solution to such problems is singleton types and witness objects, so the singleton type *Preterm n t* is used to specify the object type.

Notice that *Term* enjoys some desirable properties: stars only live at levels two and above, above values and types; arrows are constructed from pairs of things at the level of types or above — an arrow between two values doesn't make sense; type constructors live with types and above, not with values; and if a term T classifies term t, then T lives exactly one level higher than t. All of these properties are checked by the constraints in the GADT constructors.

As before, the constraints in *Term* correspond to typing rules for the object language. *Star*, *Arrow* and *Apply* each implicate a rule similar (or equivalent) to rules seen in the last section. Respectively:

$$\frac{i \in \mathbb{N}}{\vdash \star_i : \star_{i+1}}$$

$$\frac{\Gamma \vdash a : \star_i \qquad \Gamma \vdash b : \star_i \qquad i \in \mathbb{N}}{\Gamma \vdash a \to b : \star_i}$$

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash x : A}{\Gamma \vdash f\, x : B}$$

5.2 Example Terms

The object-level algebraic data types exemplified in the last section are defined similarly in this new object language. The major difference is that constructors take preterms to specify their object types. To eliminate redundancy, we introduce a simple function, *tConPair*, that makes both a term and preterm version of a type constructor.

$$\begin{aligned}
&tConPair\ l\ p = (TCon\ l\ p, PTCon\ l\ p)\\
&(nat, nat') &&= tConPair\ \text{`}Nat\ (PStar\ \#0)\\
&zero &&= VCon\ \text{`}Zero\ nat'\\
&succ &&= VCon\ \text{`}Succ\ (nat'\ \text{`}PArrow\text{`}\ nat')\\
&(bool, bool') &&= tConPair\ \text{`}Bool\ (PStar\ \#0)\\
&true &&= VCon\ \text{`}True\ bool'\\
&false &&= VCon\ \text{`}False\ bool'\\
&list &&= TCon\ \text{`}List\ (PStar\ \#0\ \text{`}PArrow\text{`}\ PStar\ \#0)
\end{aligned}$$

5.3 Type Assignment

Computing object types for terms poses two problems: the output type of *typeOf* requires a type-level typing function on preterms, and computing types for constructors requires a conversion from preterms to terms.

$$
\begin{array}{l}
\mathit{typeOf} :: \mathit{Term}\ n\ t \to \mathit{Term}\ (S\ n)\,\{\mathit{typeOfPre}\ t\} \\
\ldots \\
\mathit{typeOf}\ (\mathit{TCon}\ \_\ t) = \mathit{fromPre}\ t \\
\mathit{typeOf}\ (\mathit{VCon}\ \_\ t) = \mathit{fromPre}\ t \\
\ldots
\end{array}
$$

The first problem is easily solved. As before, stars are typed with higher stars, arrows' types are determined by their range, and preterm type constructors contain their type, which is always a star.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{typeOfPre} :: \mathit{Preterm}' \rightsquigarrow \mathit{Preterm}'} \\
\{\mathit{typeOfPre}\ (\mathit{PStar}\ n)\} & = \mathit{PStar}\ (S\ n) \\
\{\mathit{typeOfPre}\ (\mathit{PArrow}\ a\ b)\} & = \{\mathit{typeOfPre}\ b\} \\
\{\mathit{typeOfPre}\ (\mathit{PTCon}\ \mathit{name}\ (\mathit{PStar}\ n))\} & = \mathit{PStar}\ n
\end{array}
$$

Given the design of terms and preterms, the second problem is also easily solved with a simple embedding of preterms into terms.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{fromPre} :: \mathit{Preterm}\ n\ t \to \mathit{Term}\ n\,\{\mathit{typeOfPre}\ t\}} \\
\mathit{fromPre}\ (\mathit{PStar}\ n) & = \mathit{Star}\ n \\
\mathit{fromPre}\ (\mathit{PArrow}\ a\ b) & = \mathit{Arrow}\ (\mathit{fromPre}\ a)\ (\mathit{fromPre}\ b) \\
\mathit{fromPre}\ (\mathit{PTCon}\ \mathit{name}\ (\mathit{PStar}\ n)) & = \mathit{TCon}\ \mathit{name}\ (\mathit{PStar}\ n)
\end{array}
$$

With definitions for *typeOfPre* and *fromPre* in hand, defining *typeOf* is straightforward:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{typeOf} :: \mathit{Term}\ n\ t \to \mathit{Term}\ (S\ n)\,\{\mathit{typeOfPre}\ t\}} \\
\mathit{typeOf}\ (\mathit{Star}\ n) & = \mathit{Star}\ (S\ n) \\
\mathit{typeOf}\ (\mathit{Arrow}\ a\ b) & = \mathit{typeOf}\ b \\
\mathit{typeOf}\ (\mathit{Apply}\ f\ a) & = \mathbf{case}\ \mathit{typeOf}\ f\ \mathbf{of}\ \mathit{Arrow}\ a\ b \to b \\
\mathit{typeOf}\ (\mathit{TCon}\ \_\ t) & = \mathit{fromPre}\ t \\
\mathit{typeOf}\ (\mathit{VCon}\ \_\ t) & = \mathit{fromPre}\ t
\end{array}
$$

We compute object types like before with similar results:

```
prompt> zero
(VCon 'Zero (PTCon 'Nat (PStar #0)))
  : Term #0 (PTCon 'Nat (PStar #0))
prompt> typeOf zero
(TCon 'Nat (PStar #0)) : Term #1 (PStar #0)
prompt> nat
(TCon 'Nat (PStar #0)) : Term #1 (PStar #0)
```

```
prompt> typeOf (typeOf zero)
(Star #0) : Term #2 (PStar #1)
prompt> typeOf (typeOf (typeOf zero))
(Star #1) : Term #3 (PStar #2)
```

This section used a single Ωmega GADT to uniformly encode an object language with stratified typing and an infinite universe hierarchy. With this accomplished, we pursue an encoding for the level-varying star term, $\star n$.

6 Level Terms and the Star Constructor

To model universal datatypes, we want to represent the level-varying star term $\star n$ in the object language. The language already has star constants like $\star 0$ and $\star 1$, so we would like to factor-out the commonalities: the *star constructor* $\star$, and *level indexes*.

Level indexes and the star constructor form a small, cohesive language for building all of the star terms, constant and varying. Level indexes can be expressed with this simple grammar:

```
<level> ::= l <num> | n | m | ...
<num>   ::= 0 | 1 | ...
```

Each *li* term represents a constant level index, and n, m, etc. represent level variables.

Now, the star terms are easily recovered by applying the star constructor to a level. Notice that the level indexes are off by two from the levels of universes: applying $\star$ to *l0* gives $\star 0$ — a term at the second level built from the zeroth level index. This is done simply to agree with the counting of the existing star constants.

Though the star terms must live in the object language, nothing requires that the star constructor and level indexes do. Thus there is an immediate choice between placing this small language at the object level or the meta-level.

The meta-level is the natural first choice. The star constructor is only used with level indexes to create star terms, and level indexes are only used with the star constructor; they have no business commingling with the rest of the terms.

However, the current Ωmega interpreter (~v1.2) is ~11,000 lines of Haskell, at least ~6,000-7,000 of which implement the type checker. Adding a small meta language for levels and a star constructor would effect change in much of this code. Implementing universe polymorphism would be much easier if levels and the star constructor could fit into the existing term language. Thus we prefer the latter choice and consider it first.

6.1 Option 1: Levels as Object Terms

To ease implementation costs, our goal in this section is to explore ways to integrate level indexes and the star constructor into the existing object language.

In particular, we want to reuse the existing arrow to type the star constructor and avoid having to implement a second form of abstraction in the Ωmega interpreter.

The purpose of the star constructor is to build star terms from level indexes. For instance, the constant $\star 0$ should be built as $\star(l0)$ — the star constructor applied to the zeroth level index. Likewise, $\star 1$ should be built as $\star(l1)$, and the level-varying constant $\star n$ should be $\star(n)$, where n is a level variable.

Used this way, the star constructor must have an arrow type: $\star :: a \to b$, for some a and b. Since star constants are typed as $\star 0 :: \star 1 :: \star 2 :: \ldots$, then it must be the case that these new constructions are typed similarly: $\star(l) :: \star(l+1)$, where l denotes a level index. This requires that $\star$ has the dependent type $\star : \Pi l{:}L.\ \star_{l+1}$. We can translate this into Ωmega using singleton types: $\star :: L\ i \to \star(i+1)$, introducing singleton types $L\ i$ and witnesses $l\ i$ to represent the level indexes.

The star constructor has a strange type. We don't know what it means or, more importantly, whether it's meaningful at all! It doesn't seem well-founded, since $\star$ occurs in its own type. We have no solution for this problem, but are interested in fully exploring the possibility of exploiting existing mechanisms for implementing star terms and levels, so we explore the typings for levels as well.

How does the type of the star constructor constrain the possible types for level terms? Given the current formation rule, the domain and range types in an arrow must have the same level.

$$\frac{\Gamma \vdash a : \star_i \qquad \Gamma \vdash b : \star_i \qquad i \in \mathbb{N}}{\Gamma \vdash a \to b : \star_i}$$

One option is to work within the constraints of this formation rule and encode level terms throughout the universe hierarchy. Alternatively, we can relax the formation rule to allow the domain type to live at or above the level of the range type; this lets us place all levels together at some "sufficiently high" universe. A third option is to introduce a second formation rule that simply permits level terms in the domain of an arrow, allowing the most flexible placement of level terms. We explore all three ideas.

Leaving the formation rule unchanged requires that each level term lives in a different universe in the hierarchy. To see why, consider $\star 0 = \star(l\ 0) :: \star 1$. Since domain and range types must be in the same universe, and $l\ 0 :: L\ 0$, then $L\ 0$ and $\star 1$ must be in the same level. Thus, $L\ 0 :: \star 2$; in general, $L\ i :: \star(i+2)$. This creates an ascending sequence of level terms — pairs of singleton types and witnesses — parallel to the existing sequence of star constants $\star 0 :: \star 1 :: \star 2 :: \ldots$. This arrangement is illustrated in Figure 2.

Alternatively, we can relax the assumptions in the formation rule to allow the input type to be in any universe at or higher than that of the output type:

$$\frac{\Gamma \vdash a : *_j \qquad \Gamma \vdash b : *_i \qquad j \geqslant i \qquad i, j \in \mathbb{N}}{\Gamma \vdash a \to b : *_i}$$

Now, each level term can be placed at infinitely many different levels. $L\ 0$ can be typed by $\star 2$, $\star 3$ or higher; in general, we can choose any of the typings

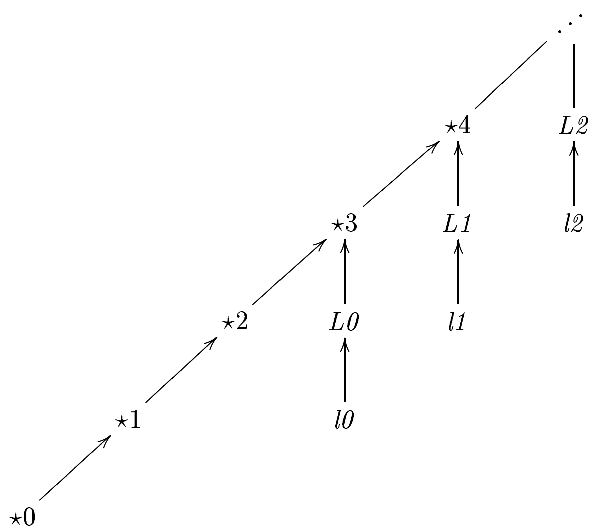

Figure 2: Level placement with original formation rule

$L\ i :: \star(i + 2 + j)$, where j is non-negative. With this new flexibility, a more organized version of the previous arrangement might be to collect all the level terms together at some sufficiently high universe so that anywhere a level term is required, it can be "plucked out" from above. However, we still assume that each level term is typed by a star constant; if every level term is grouped together at the same level in the hierarchy, then no star constant can be constructed to type the levels! This arrangement is illustrated in Figure 3.

The third approach is to introduce a second formation rule so that level terms can be typed by something other than a star constant. If each $L\ i$ is typed by some new term, say $\mathbb{L}$, then the second formation rule could be:

$$\frac{\Gamma \vdash l : \mathbb{L} \qquad \Gamma \vdash b : *_i \qquad i \in \mathbb{N}}{\Gamma \vdash l \to b : *_i}$$

With this rule and the fact that level terms are no longer typed by star constants, every star constant could be constructed no problem. The new problem is how to type $\mathbb{L}$. Axiomatizing $\mathbb{L} : \mathbb{L}$ introduces logical paradoxes [3]. We might add a second infinite hierarchy to avoid loops, but it would be mostly superfluous. This arrangement, the most promising of the three, is illustrated in Figure 4.

All of this reasoning is very informal, guided primarily by intuition. But we think it's thought-provoking and worth presenting: where we see dead-ends, others might see exciting possibilities. Regardless of how level terms are typed, the critical problem is the type of the star constructor: its current form doesn't seem logically sound.

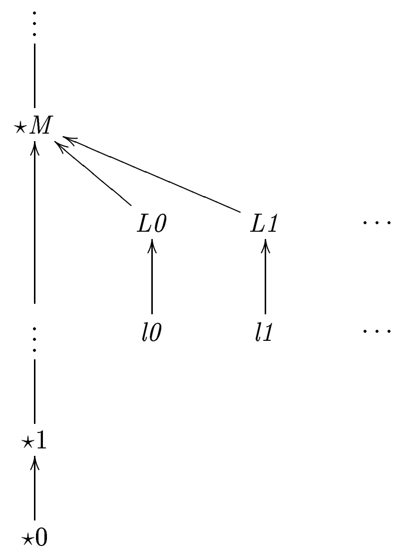

Figure 3: Level placement with relaxed formation rule

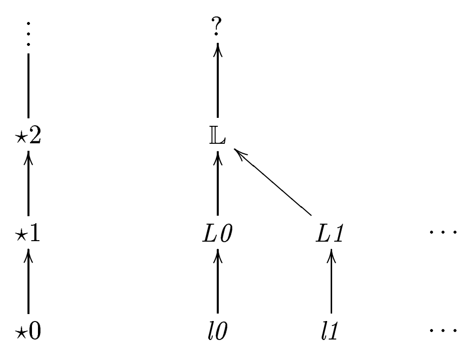

Figure 4: Level placement with overloaded formation rule

6.2 Option 2: Levels as Meta Terms

To avoid typing the star constructor, we accept the implementational burden of adding a small meta-language to house the star constructor and level indexes. From it, star terms can be constructed and introduced into the actual object language.

The encoding of the new meta-language is split over two GADT's: level terms live in one, and the star constructor lives in the *Term* datatype, since its output is a term. The encoding again exploits meta-level type polymorphism to represent the level variable and any constraints on it.

$$
\begin{array}{l}
\textbf{data } \mathit{Level} :: \mathit{Nat} \leadsto \star 0 \textbf{ where} \\
\quad \mathit{Lv} :: \mathit{Nat}'\ n \to \mathit{Level}\ n \\
\quad N\ \ :: \mathit{Level}\ n
\end{array}
$$

Lv is a simple injection from Nat', and N represents a level variable using a type variable.

The encoding of the object language changes slightly to incorporate the new level terms. As intended, star terms are now constructed from level terms instead of meta-numbers:

$$\mathit{Star} :: \mathit{Level}\ n \to \mathit{Term}\ (S\ (S\ n))\ (\mathit{PStar}\ (S\ n))$$

This entails adding a new typing rule for level-varying star terms:

$$\frac{}{\vdash \star_n : \star_m, \{m = n + 1\}}$$

Here, $\{m = n + 1\}$ is meant to express a constraint between the level variables n and m. Like in checking type polymorphism, constraints can be used to check (and infer) level variables. The rest of the changes to the encoding of the object language are straightforward and uninteresting, so they are omitted here.

As examples of universal terms, consider universal natural numbers and a universal list constructor:

$$
\begin{array}{ll}
(\mathit{unat}, \mathit{unat}') & = \mathit{tConPair}\ \text{`}\mathit{UNat}\ (\mathit{PStar}\ N) \\
\mathit{uzero} & = \mathit{VCon}\ \text{`}\mathit{UZero}\ \mathit{unat}' \\
\mathit{usucc} & = \mathit{VCon}\ \text{`}\mathit{USucc}\ (\mathit{unat}'\ \text{`}\mathit{PArrow}\text{`}\ \mathit{unat}') \\
\mathit{ulist} & = \mathit{TCon}\ \text{`}\mathit{List}\ (\mathit{PStar}\ N\ \text{`}\mathit{PArrow}\text{`}\ \mathit{PStar}\ N)
\end{array}
$$

$unat$ and $ulist$ are like nat and $list$, except they are constructed from level-varying star terms $PStar\ N$ and can be instantiated in any universe. For instance, applying the normal list type constructor to the universal number type instantiates $unat$ as a type:

$$\mathit{Apply}\ \mathit{list}\ \mathit{unat} :: \mathit{Term}\ \#1\ (\mathit{PStar}\ \#0)$$

In general, combining universal terms with grounded terms instantiates the universal terms and produces grounded terms. Combining universal terms with other universal terms creates composite universal terms (which will ground only when eventually combined with grounded terms). For example, $Apply\ ulist\ unat$ is universal, but $Apply\ list\ (Apply\ ulist\ unat)$ is grounded.

$$
\begin{array}{l}
\mathit{nat}\ \ :: \mathit{Term}\ \#1\ (\mathit{PStar}\ \#0) \\
\mathit{unat} :: \mathit{Term}\ \#1\ (\mathit{PArrow}\ (\mathit{PStar}\ \#0)\ (\mathit{PStar}\ \#0)) \\
\mathit{list}\ \ :: \forall a : \mathit{Nat}.\ \mathit{Term}\ \#(1 + a)\ (\mathit{PStar}\ a) \\
\mathit{ulist} :: \forall a : \mathit{Nat}.\ \mathit{Term}\ \#(1 + a)\ (\mathit{PArrow}\ (\mathit{PStar}\ a)\ (\mathit{PStar}\ a)) \\
\mathit{Apply}\ \mathit{list}\ \ \ \mathit{nat} \qquad\qquad\qquad :: \mathit{Term}\ \#1\ (\mathit{PStar}\ \#0)
\end{array}
$$

$$
\begin{array}{ll}
\mathit{Apply\ ulist\ nat} & :: \mathit{Term}\ \#1\ (\mathit{PStar}\ \#0) \\
\mathit{Apply\ ulist\ unat} & :: \forall a : \mathit{Nat}.\ \mathit{Term}\ \#(1+a)\ (\mathit{PStar}\ a) \\
\mathit{Apply\ list}\ (\mathit{Apply\ ulist\ unat}) & :: \mathit{Term}\ \#1\ (\mathit{PStar}\ \#0)
\end{array}
$$

This is a beginning for universe polymorphism!

7 Related Work

In [8], Luo presented an "Extended Calculus of Constructions" (ECC), extending Coquand and Huet's Calculus of Constructions (CC) with strong sums, an infinite type hierarchy — similar to the one in Ωmega — and *cumulativity.* Cumulativity freely promotes terms up the hierarchy: a term can be typed by anything typing its type, or its type's type, etc. This feature can be used to achieve similar effects to universe polymorphism, but violates the phase distinction: values permeate upward throughout the entire hierarchy, promoting every run-time entity to a compile-time entity.

Harper and Pollack [6] study Coquand's "Generalized Calculus of Constructions" [3] (CC^{ω}), also extending CC with an infinite type hierarchy (but omitting strong sums). They use a similar notion of cumulativity as Luo, but add the idea of using binding-site polymorphism, like let-polymorphism in ML and Haskell, to achieve universe polymorphism. The cumulativity rule in CC^{ω} is similar to the one in ECC, promoting run-time information in the same way.

Restricting these systems to respect the phase distinction and applying them to Ωmega would provide a nice theoretical underpinning for universe polymorphism. An initial challenge in doing so would be formalizing Ωmega's existing type system.

8 Conclusion

The motivation for this paper was to build models to explore universe polymorphism for Ωmega. In particular, it followed the methodology illustrated in [13], utilizing Ωmega GADT's with extensible kinds, singleton types and type functions to partially type-check an object language with features necessary for universe polymorphism: self-typing terms, a level-varying star term $\star n$, and an infinite universe hierarchy.

This paper explored some consequences of incorporating the star constructor and level terms into an Ωmega-like language with self-typing terms. It outlined difficulties encountered when trying to add the terms directly into the language, opening many questions, and it illustrated the potential ease with which they can be encapsulated within a small meta-language for constructing star terms. It also illustrated a way to encode an Ωmega-like language with self-typing terms and universal datatypes as an object language within Ωmega using GADT's, contributing to Sheard's technique presented in [13].

Acknowledgements

I would like to thank my advisor, William Cook, for two years of stimulating teaching and discussion, and the rest of my thesis committee, Greg Lavender and Mohamed Gouda, for valuable feedback and review. I also recognize Tim Sheard with whom I did the bulk of this research, and thank him for dedicating so much time towards working with me. Lastly, I thank the Undergraduate Research Opportunities Program (UROP) at the University of Texas at Austin for funding my research, which enabled me to travel to Portland and work with Tim in the first place.

References

[1] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[2] Luca Cardelli. Phase distinctions in type theory. Unpublished manuscript.

[3] Thierry Coquand. An analysis of girard's paradox, 1986.

[4] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, 1988.

[5] Judicaël Courant. Explicit universes for the calculus of constructions. Theorem Proving in Higher Order Logics, 2002.

[6] Robert Harper and Robert Pollack. Type checking with universes. In *TAPSOFT '89: 2nd international joint conference on Theory and practice of software development*, pages 107–136, Amsterdam, The Netherlands, The Netherlands, 1991. Elsevier Science Publishers B. V.

[7] William Howard. The formulae-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[8] Zhaolui Luo. Ecc, an extended calculus of constructions. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, June 1989.

[9] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PDLI '89)*, pages 199–208. ACM Press, 1989.

[10] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[11] Tim Sheard. Omega download page. http://www.cs.pdx.edu/ sheard/Omega/.

[12] Tim Sheard. Putting curry-howard to work. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*. ACM Press, September 2005.

[13] Tim Sheard. Playing with type systems: Automated assistance in the design of programming langauges, 2006. http://web.cecs.pdx.edu/ sheard/papers/PlayingWithTypes2.ps.

[14] Tim Sheard, James Hook, and Nathan Linger. Gadts + extensible kinds = dependent programming. Technical report, Portland State University, 2005. http://www.cs.pdx.edu/ sheard/papers/GADT+ExtKinds.ps.