Meta-Programming with Typed Object-language Representations

Emir Pašalić and Nathan Linger

OGI School of Science & Engineering
Oregon Health & Science University
{pasalic,rlinger}@cse.ogi.edu *

Abstract. We present two case studies demonstrating the use of type-equality constraints in a meta-language to enforce semantic invariants of object-language programs such as scoping and typing rules. We apply this technique to several interesting problems, including (1) the construction of tagless interpreters; (2) statically checking de Bruijn indices involving pattern-based binding constructs; and (3) evolving embedded DSL implementations to include domain-specific types and optimizations that respect those types.

1 Introduction

Meta-programs manipulate object-programs as data. Traditionally, these object-programs are represented with algebraic datatypes that enforce *syntactic invariants* of the object-language: only syntactically valid object programs are representable. In this paper, we explore a method for object-program representation that also enforces the *semantic invariants* of scoping and typing rules. The type system of the meta-language then guarantees that all meta-programs respect these additional object-language properties, thereby increasing our assurance in the correctness of meta-programs.

Although this can be done using an *encoding* of equality types in (standard extensions to) the Haskell 98 type system, the meta-language Ωmega [1] that we use in this paper directly supports the notion of *type equality*. Its type system automatically propagates and solves type-equality constraints, thus implementing a form of Cheney and Hinze's *First Class Phantom Types* [2]. Our case studies show that such support from the type system makes programming with well-typed object programs considerably less tedious than explicitly encoding equality types. Ωmega also supports user-defined kinds and staging. This integration of features makes Ωmega a powerful meta-programming tool.

In this paper, we apply this new meta-programming approach to several interesting problems: (1) the construction of tagless interpreters; (2) statically checking de Bruijn indices involving pattern-based binding constructs; and (3)

* The work described in this paper is supported by the National Science Foundation under the grant CCR-0098126.

evolving embedded DSL implementations to include domain-specific types and optimizations that respect those types.

Our case studies demonstrate that these techniques support the embedding of the logical frameworks style of judgments into a programming language such as Haskell. This is important because it allows programmers to reason about their programs as they write them rather than separately at the meta-logical level.

Tagless Staged Interpreters. Staging a definitional interpreter written in a staged language (e.g. MetaML) is one way of deriving an implementation that is both reliable and efficient [3]: reliable because the staged interpreter retains a close link with the original (reference) interpreter, and efficient because staging can remove an entire layer of interpretive overhead, thereby yielding a simple compiler.

However, many existing staged interpreter implementations retain at least one significant source of overhead: *tagging* [4]. Unnecessary tagging arises when both the meta-language and the object-language are strongly typed. The meta-language type system forces the programmer to encode object-language values into an universal (tagged) domain, leading to numerous and unnecessary runtime tagging and untagging operations. In previous work [5], we have addressed the problem of tagging by using a dependently typed meta-language which allows the user to type the interpreter without the need for a universal value domain. In this paper, we show how to get the same effect using well-typed object-language representations.

Statically Checked de Bruijn Indices. De Bruijn formulated [6] a technique for handling binding of variables by using a nameless, position dependent naming scheme. While elegant, this framework is notorious for subtle off-by-one errors. Using well-typed object-language representations, these off-by-one errors become meta-program typing errors and are identified earlier. We show that this technique scales gracefully to handle richer binding mechanisms involving patterns.

A Process for Evolving DSL Implementations. The benefits of domain-specific languages (DSLs) are well known, but the cost of design and implementation can outweigh the benefits. The technique of embedding a DSL's implementation in a host language with flexible abstraction mechanisms has proven a good way to reduce these costs.

We illustrate how the features of Ωmega's type system make it well suited as a host language for embedding DSLs. User-defined kinds provide a mechanism for defining a domain-specific type system. Well-typed object-language representations harness the type system to prove type-correctness of domain-specific optimizations to object-programs. Again, staging constructs allow us to define a tagless staged interpreter for our DSL. This combination of language features makes Ωmega a powerful host language for embedding DSL implementation. The process presented here for evolving DSL implementations should be straightforward to reproduce. Parts of it can be automated.

The remaining sections are organized as follows. Section 2 introduces Ωmega. We then develop two comprehensive case studies: First, we implement a tagless staged interpreter for a functional language with pattern matching (Sections 3 and 4). Second, we present a development of a small domain-specific language for describing geometric regions (Sections 5 and 6) where we describe not only an efficient staged interpreter, but also type-preserving optimizations. Section 8 gives an example of using the region language. Sections 9 and 10 discuss related and future work.

2 Ωmega: A Meta-language with Support for Type Equality

In this section we shall familiarize the reader with the most important features of Ωmega: type equality, user-defined kinds and staging. The syntax and type system of Ωmega are descended from Haskell, while its staging support descends from that of MetaML.

Type Equality in Haskell. A key technique that inspired the work described in this paper is the encoding of *equality between types* as a Haskell type constructor (`Equal a b`). Thus a non-bottom value (`p::Equal a b`), can be regarded as a proof of the proposition that `a` equals `b`.

The technique of encoding the equality between types `a` and `b` as a polymorphic function of type $\forall\varphi.\ \varphi$ `a` $\rightarrow \varphi$ `b` was proposed by both Baars & Swierstra [7], and Cheney & Hinze [2] at about the same time, and has been described somewhat earlier in a different setting by Weirich [8]. We illustrate this by the datatype `Equal` : $* \rightarrow * \rightarrow *$

```
data Equal a b = Equal (∀φ. φ a → φ b)
cast :: Equal a b → φ a → φ b
cast (Equal f) = f
```

The logical intuition behind this definition (also known as Leibniz equality [9]) is that two types are equal if, and only if, they are interchangeable in any context. This context is represented by the arbitrary Haskell type constructor φ. Proofs are useful, since from a proof `p :: Equal a b`, we can extract functions that *cast* values of type C[`a`] to type C[`b`] for type contexts C[ ]. For example, we can construct functions `a2b::Equal a b` $\rightarrow$ `a` $\rightarrow$ `b` and `b2a::Equal a b` $\rightarrow$ `b` $\rightarrow$ `a` which allow us to cast between the two types `a` and `b` in the identity context. Furthermore, it is possible to construct combinators that manipulate equality proofs based on the standard properties of equality (transitivity, reflexivity, congruence, and so on).

Equality types are described elsewhere [7], and we shall not further belabor their explanation. The essential characteristic of programming with type equality in Haskell is that programmers must explicitly manipulate proofs of equalities between types using a specific set of combinators. This has two practical drawbacks. First, such explicit manipulation is tedious. Second, while present throughout a program, the equality proof manipulations have no real computational content – they are used solely to leverage the power of the Haskell type system to accept certain programs that are not typable when written without the proofs. With all the clutter induced by proof manipulation, it is sometimes difficult to discern the difference between the truly important algorithmic part of the program and mere equality proof manipulation. This, in turn, makes programs brittle and rather difficult to change.

2.1 Type Equality in Ωmega

What if we could extend the type system of Haskell, in a relatively minor way, to allow the type checker itself to manipulate and propagate equality proofs? Such a type system was proposed by Cheney and Hinze [2], and is one of the ideas behind Ωmega [1]. In the remainder of this paper, we shall use Ωmega, rather than pure Haskell to write our examples. We conjecture that, in principle, whatever it is possible to do in Ωmega, it is also possible to do in Haskell (plus the usual set of extensions, possibly using some unsafe operations), only in Ωmega it is expressed more cleanly and succinctly.

The syntax and type system of Ωmega has been designed to closely resemble Haskell (with GHC extensions). For practical purposes, we could consider (and use) it as a conservative extension to Haskell. In this section, we will briefly outline the useful differences between Ωmega and Haskell.

In Ωmega, the equality between types is not encoded explicitly (as the type constructor `Equal`). Rather, it is built into the type system, and is used implicitly by the type checker. Consider the following (fragmentary) datatype definitions.[1]

```
data Exp e t = Lit Int           where t = Int
             | V (Var e t)
data Var e t = ∀γ.  Z            where e = (γ,t)
             | ∀γα. S (Var γ t) where e = (γ,α)
```

Each data constructor in Ωmega may contain a `where` clause which contains a list of equations between types in the scope of the constructor definition. These equations play the same role as the Haskell type `Equal` in Section 2, with one important difference: the user is not required to provide any actual evidence of type equality – the Ωmega type checker keeps track of equalities between types and proves and propagates them automatically.

The mechanism Ωmega uses for this is very similar to the constraints that the Haskell type checker uses to resolve class based overloading. A special qualified type [10] is used to assert equality between types, and a constraint solving system

[1] *Syntactic and Typographical Conventions.* We adopt the GHC syntax for writing the existential types with a universal quantifier that appears to the left of a data constructor. We also replace the keyword `forall` with the symbol ∀. We shall write explicitly universally or existentially quantified variables with Greek letters. Arrow types (`->`) will be written as $\rightarrow$, and so on.

is used to simplify and discharge these assertions. When assigning a type to a type constructor, the equations specified in the where clause become predicates in a qualified type. Thus, the constructor `Lit` is given the type `∀e t. (t=Int) => Int → Exp e t`. The equation `t=Int` is just another form of predicate, similar to the class membership predicate in Haskell (e.g., `Show a => a → String`).

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

Obligations. The first set of constraints is a set of *obligations*. Obligations are generated by the type checker either when (a) the program constructs values with constructors that contain equality constraints; or (b) an explicit type signature in a definition is encountered.

For example, consider type checking the expression `(Lit 5)`. The constructor `Lit` is assigned the type `∀e t.(t=Int) => Int → Exp e t`. Since `Lit` is polymorphic in `e` and `t`, the type variable `t` can be instantiated to `Int`. Instantiating `t` to `Int` also makes the equality constraint obligation `Int=Int`, which can be trivially discharged by the type checker.

```
Lit 5 :: Exp e Int      with obligation     Int = Int
```

Data constructors of `Exp` and `Var` have the following types:

```
Lit :: ∀e t.       (t=Int)      => Int → Exp e t
Z   :: ∀e e' t.    (e=(e',t))   => Var e t
S   :: ∀e t e' t'. (e=(e',t'))  => Var e' t → Var e t
```

which can be *instantiated* as follows:

```
Lit :: Int → Exp e Int
Z   :: Var (e',t) t
S   :: Var e' t → Var (e',t') t
```

We have already seen this for `Lit`. Consider the case for `Z`. First, the type variable `e` can be instantiated to `(e',t)`. After this instantiation, the obligation introduced by the constructor becomes `(e',t)=(e',t)`, which can be immediately discharged by the built-in equality solver. This leaves the instantiated type `(Var (e',t) t)`.

Assumptions. The second set of constraints is a set of *assumptions* or *facts*. Whenever, a constructor with a `where` clause is pattern-matched, the type equalities in the where clause are added to the current set of assumptions in the scope of the pattern. These assumptions can be used to discharge obligations. For example, consider the following partial definition:

```
evalList ::  Exp e t → e → [t]
evalList exp env = case exp of Lit n → [n]
```

When the expression `exp` of type `(Exp e t)` is matched against the pattern `(Lit n)`, the equality `t=Int` (see definition of `Lit`) is introduced as an assumption. The type signature of `evalList` induces the obligation that the body of the definition has the type `[t]`. The right-hand side of the `case` expression, `[n]`, has the type [Int]. The type checker now must discharge (prove) the obligation [t]=[Int], while using the fact, introduced by the pattern (Lit n) that t=Int. The Ωmega type checker uses an algorithm based on congruence closure [11], to discharge equality obligations. It automatically applies the laws of equality to solve such equations. In this case, the equation is discharged using congruence.

2.2 User-defined kinds

Another feature of Ωmega that we will use is the facility for *user-defined kinds.* Kinds classify types in the same way that types classify values. Just as value expressions can have varying types, type expressions can have varying kinds. There is a base kind $*$ (pronounced "Star") that classifies all types that classify values such as Int, [Char], Bool $\rightarrow$ Float. The kind $* \rightarrow *$ classifies type constructors such as [], the list type constructor, and Maybe, a type constructor for optional values. Note that there are no values of type [] or Maybe.

In addition to kinds built up from $*$ and $\rightarrow$, Ωmega allows programmers to define their own kinds. The syntax is analogous to datatype definitions:

```
kind Unit = Pixel | Centimeter
```

This declaration defines a new kind Unit and two new types, Pixel and Centimeter, of kind Unit. Though there are no values of type Pixel or Centimeter, these types can be used as type parameters to other type constructors. This use of type parameters has been called indexed types [12].

2.3 An Introduction to Staging

Staging is a form of meta-programming that allows users to partition a program's execution into a number of *computational stages.* The languages MetaML [13], MetaOCaml [14], and Template Haskell support this partitioning by the use of special syntax called staging annotations. This is also the case with Ωmega, and we will use two such staging annotations in the examples we present in this paper.

Brackets, [| _ |], surrounding an expression, lift the expression into the next computational stage. This is analogous to building a piece of *code* that when evaluated will be equivalent to the bracketed expression. In staged languages, the type system reflects the delay, so that if an expression e has the type Int, the expression [|e|] has the type (Code Int) (pronounced "code of int"). Escape, $(_), can only occur inside of code brackets and drops the expression it surrounds into the previous computational stage. Once the escaped expression is evaluated, its result, which must itself be delayed, is then incorporated at the point at which the escape has occurred. This is analogous to evaluating a computation that builds *code* and then "splices" the resulting code into the larger piece of code being built.

$$\begin{array}{lll}
\tau \in \mathbb{T} & ::= & \mathsf{Nat} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \\
\Gamma \in \mathbb{G} & ::= & \langle\rangle \mid \Gamma, \tau \\
p \in \mathbb{P} & ::= & \bullet_\tau \mid \mathsf{Inl}_{\tau_1\ \tau_2}\ p \mid \mathsf{Inr}_{\tau_1\ \tau_2}\ p \mid (p_1, p_2) \\
e \in \mathbb{E} & ::= & \mathsf{Lit}\ n \mid \mathsf{Var}\ n \mid \lambda p.e \mid e_1\ e_2 \mid (e_1, e_2) \mid \mathsf{Inl}_{\tau_1\ \tau_2}\ e \mid \mathsf{Inr}_{\tau_1\ \tau_2}\ e \mid \mathsf{case}\ e_1 \mathsf{of}\ \overline{p_n \to e_n}
\end{array}$$

Fig. 1. Syntax of L_1. The notation $\overline{x}$ indicates a sequence of 1 or more x's.

Annotations introduce a notion of *level* into the syntax of programs. The level of an expression is the number of its surrounding brackets minus the number of its surrounding escapes. Levels correspond to stages, roughly, in the sense that an expression at level n is evaluated in the n-th stage. The type system of MetaML (and similar staged languages) statically guarantees that the staged program is free from *phase errors* – situations in which a variable bound at a later stage is used in an earlier stage. By adding staging annotations to an interpreter, we can change its behavior so that *static* computation is performed in an earlier stage. This specializes the interpreter with respect to a particular object-program, and produces a more efficient "residual" program in the next stage, which is free from all interpretive overhead [16]. In effect, this produces a compiler from an interpreter [3].

There are two more staging annotations in Ωmega. The `lift` annotation evaluates an expression of primitive type to a literal value and builds some trivial code that returns that value. The final annotation is `run`. It evaluates its argument (of code type) and executes the resulting code.

3 A Language With Patterns

In this section, we present a simple λ-calculus based language with sums and products, which are eliminated by the use of pattern matching. Variable binding is done with *de Bruijn* indices [6]. In this notation, variables are named by natural number indices. The index of a variable is the number of intervening binding sites between a variable's use and its definition. The notion of binding site is made more complex by the presence of pattern matching, and handling this complication in an elegant and type aware manner is a contribution of this work. The language is based on the simply typed λ-calculus. We shall refer to this language as L_1.

3.1 Syntax

The syntax of the language L_1 is given in Figure 1. Four sets of terms are defined:
(1) A set of *types*, $\mathbb{T}$, consisting of a base (in this case natural numbers), function, product, and sum types.
(2) A set of *type assignments*, $\mathbb{G}$, which are defined as finite sequences of types. The binding convention used in this presentation is that the n-th type from the right in the sequence is the type of the free variable with the index n.

(3) A set of *patterns*, $\mathbb{P}$. The most basic kind of pattern is the (nameless) *variable binding pattern*, $\bullet_\tau$: Patterns can also be sum patterns, $\mathsf{Inl}_{\tau_1\ \tau_2}\ p$ and $\mathsf{Inr}_{\tau_1\ \tau_2}\ p$, or product (pair) patterns (p_1, p_2). Patterns can be nested to arbitrary depth.

(4) A set of *expressions*, $\mathbb{E}$. The mixing of de Bruijn notation and patterns make expressions slightly non-standard. As in standard de Bruijn notation, variables are represented by natural number indices. Variables are bound in patterns, which occur in λ-abstractions and case-expressions. In standard de Bruijn notation the index of a variable indicates the number of intervening binding sites between the use and binding site of the variable, the index 0 being the "most recently bound variable." In this language a pattern might bind several variables, so the notion of "binding site" must choose a particular strategy as to which variables in a pattern are bound "earlier" than others. Sum types are introduced by injection constructs $\mathsf{Inl}\ e$ and $\mathsf{Inr}\ e$. Products are introduced by the pairing construct, (e_1, e_2). Both sum and product types are eliminated by pattern matching. Due to the limited space in this paper, we only show a subset of the language features we have been able to implement. Additional features include recursive definitions, let-bindings, patterns with guards, staging constructs, etc.

3.2 Meta-language Formalization

Figure 1 is a typical formalization of the syntax of a language. A good meta-language should not only capture the syntactic properties of the language, but semantic properties as well. Semantic properties of a language are often captured as judgments over the syntax of the language. Using the equality type extensions of Ωmega, we can implement the object language L_1 in a way which enforces the static semantics as defined in Figures 2 through 4. In this, we shall use the following techniques:

Type equality constraints. A key technique is the type equality constraint. Written as equations in `where`-clauses in datatype definitions, these constraints allow the programmer to specify an exact shape of type arguments to type constructors, and to use that shape to encode properties.

Type indexes. Object language types are represented by meta-language Ωmega types. For example, the L_1 type ($\mathsf{Nat} \to \mathsf{Nat}$) corresponds to the Ωmega type `(Int -> Int)`.

Well-typed terms. Each judgment of the static semantics is represented by an Ωmega datatype. A value of each of these datatypes represents a derivation of the corresponding judgment. The actual `data` type definitions for the typing judgments of each syntactic category are found in the corresponding right-hand columns of Figures 2 through 4.

For example, a derivation of the pattern judgment $\Gamma \vdash p : \tau \Rightarrow \Gamma'$ is represented by a value of the type `(Pat t gammaIn gammaOut)`, where `t` corresponds to τ, `gammaIn` to Γ and `gammaOut` to Γ'. Note that only the `t`, `gammaIn` and `gammaOut` arguments of the judgment are also in the `data` type, but *not* the pattern argument (p). This is possible because the judgments are syntax directed

$$\frac{}{\Gamma, \tau \vdash 0 : \tau}(\text{Base}) \quad \frac{\Gamma \vdash n : \tau}{\Gamma, \tau' \vdash (n+1) : \tau}(\text{Weak})$$

```
data Var e t
  = ∀γ.  Z  where e = (γ,t)
  | ∀γα. S (Var γ t) where  e=(γ,α)
```

Fig. 2. Static semantics of L_1 variables: ($\Gamma \vdash n : \tau \subseteq \mathbb{G} \times \mathbb{N} \times \mathbb{T}$) and `Var e t`.

and the constructors of the derivations encode exactly the same information. This trick is used in the other judgments as well.

Instead of manipulating syntax, the meta-program manipulates data structures representing derivations of the typing judgments. The main advantage of this representation scheme, is that only well-typed L_1 terms can be created and manipulated in an Ωmega program. One might think that constructing and manipulating judgments is more complicated than constructing and manipulating syntax. We will argue that this is not necessarily the case.

3.3 Static Semantics

The static semantics of the language L_1 is defined as a set of three inductive judgment relations for variables, patterns and expressions.

Variables (Figure 2). The variable typing judgment (Figure 2) is defined inductively on the natural number index of a variable. Its task is to project the appropriate type for a variable from the type assignment: the constructor `Z` projects the 0-th type; iterating the constructor `S` n times projects the n-th type. Not surprisingly, the structure of variable judgments is reminiscent of the structure of natural numbers. In the right-hand column of Figure 2 variable judgments are represented by the type constructor `(Var e t)`, whose first argument, `e`, is the typing assignment, and whose second argument, `t`, is the type of the variable expression itself.

The two constructors, `Z` and `S`, correspond to using the Base and Weak rules to construct the variable judgments. The constructor `Z` translates the inductive definition directly: its definition states that there exists some environment γ such that the environment `e` is equal to γ extended by `t`. The constructor `S` takes a "smaller" judgment `(Var` γ `t)`, and asserts the requirement that the environment `e` is equal to the pair (γ, α), where both γ and α are existentially quantified.

Patterns (Figure 3). The pattern typing judgment *relates* an "input" type assignment Γ, a pattern p which should match a value of type τ, and an extended type assignment Γ' which assign types to variables in p. It is defined in Figure 3. As more than one variable in a pattern can be bound, the judgment specifies how names of variables are related to numerical indices. The choice is expressed

$$\frac{}{\Gamma \vdash \bullet_\tau : \tau \Rightarrow \Gamma, \tau}(\text{Var}) \quad \frac{\Gamma \vdash p : \tau_1 \Rightarrow \Gamma'}{\Gamma \vdash \mathsf{Inl}_{\tau_1\ \tau_2}\ p : \tau_1 + \tau_2 \Rightarrow \Gamma'}(\text{Inl})$$

$$\frac{\Gamma \vdash p : \tau_2 \Rightarrow \Gamma'}{\Gamma \vdash \mathsf{Inr}_{\tau_1\ \tau_2}\ p : \tau_1 + \tau_2 \Rightarrow \Gamma'}(\text{Inr})$$

$$\frac{\Gamma \vdash p_1 : \tau_1 \Rightarrow \Gamma' \qquad \Gamma' \vdash p_2 : \tau_2 \Rightarrow \Gamma''}{\Gamma \vdash (p_1, p_2) : \tau_1 \times \tau_2 \Rightarrow \Gamma''}(\text{Pair})$$

```
data Pat t gin gout =
    PVar  where gout = (gin,t)
  | ∀αβ. PInl (Pat α gin gout)
    where t = (Either α β)
  | ∀αβ. PInr (Pat β gin gout)
    where t = (Either α β)
  | ∀αβγ.PPair (Pat α gin γ)
         (Pat β γ  gout)
    where t = (α,β)
```

Fig. 3. Static semantics of L_1 patterns ($\Gamma \vdash p : \tau \Rightarrow \Gamma' \subseteq \mathbb{G} \times \mathbb{P} \times \mathbb{T} \times \mathbb{G}$) and `Pat t gin gout`.

in the Pair rule: the "furthest" variable binding site is the leftmost bottommost variable. For example: $\lambda(\bullet,\bullet).\,(\mathsf{Var}\ 0, \mathsf{Var}\ 1)$ corresponds to the function $\lambda(x,y).\,(y,x)$.

The pattern typing judgment $\Gamma_{\mathsf{in}} \vdash p : \tau \Rightarrow \Gamma_{\mathsf{out}}$ is encoded by the Ωmega datatype (`Pat t gin gout`) in the right-hand column of Figure 3.

The constructor function for variable-binding patterns `PVar` requires an equality constraint that the type of the target type assignment `gout` is equal to the source type assignment `gin` paired with the type of the pattern itself (`gout=(gin,t)`). The constructor functions for building patterns for sum types `PInl` (respectively `PInr`) take sub-pattern judgments (`Pat` α `gin gout`) (respectively (`Pat` β `gin gout`)), and require that `t` equals (`Either` α β). The most interesting case is the pattern constructor function for pair types `PPair`.

```
··· | ∀αβγ. PPair (Pat α gin γ) (Pat β γ gout) where t = (α,β)
```

It takes a pair of sub-pattern judgments. It is worth noting how the target type assignment of the first argument, γ, is then given as a source type assignment to the second argument, thus imposing left-to-right sequencing on type assignment extension for pairs.

Expressions (Figure 4). The typing judgment for expressions is defined in Figure 4. The expression judgments are represented by the type constructor (`Exp e t`). Again, `e` is the type assignment, and `t` the type of the expression itself. A general pattern emerges: by using existentially quantified variables and type equality constraints, the constructor functions of the datatype mimic the structure of the formal judgments. In the `Exp` data type there are two interesting constructor functions `Abs` and `Case`, which include the `Pat` sub-judgment. These cases implement the static scoping discipline of the language, and ensure that both the scoping discipline and the typing discipline are maintained by the metaprogram.

```
··· | ∀αβγ. Abs (Pat α e γ) (Exp γ β) where t = α → β
```

In the λ-abstraction case, the sub-pattern judgment transforms the environment from `e` to γ, and the body must be typable in the environment γ. Only then

$$\frac{}{\Gamma \vdash \mathsf{Lit}\ n : \mathsf{Nat}}\text{(Lit)} \qquad \frac{\Gamma \vdash n : \tau}{\Gamma \vdash \mathsf{Var}\ n : \tau}\text{(Var)}$$

$$\frac{\Gamma \vdash p : \tau_1 \Rightarrow \Gamma' \quad \Gamma' \vdash e : \tau_2}{\Gamma \vdash \lambda p.e : \tau_1 \to \tau_2}\text{(Abs)}$$

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}\text{(App)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}\text{(Pair)}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{Inl}_{\tau_1\ \tau_2}\ e : \tau_1 + \tau_2}\text{(Inl)}$$

$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{Inr}_{\tau_1\ \tau_2}\ e : \tau_1 + \tau_2}\text{(Inr)}$$

$$\frac{\Gamma \vdash e : \tau \quad \overline{\Gamma \vdash p_n : \tau \Rightarrow \Gamma_n \quad \Gamma_n \vdash e_n : \tau'}}{\mathsf{case}\ e\ \mathsf{of}\ \overline{p_n \to e_n} : \tau'}\text{(Case)}$$

```
data Exp e t
  =       Lit Int (Equal t Int)
  |       V (Var e t)
  | ∀αβγ. Abs (Pat α e γ) (Exp γ β)
             where t = (α → β)
  | ∀α.   App (Exp e (α → t))
             (Exp e α)
  | ∀αβ.  Inl (Exp e α)
     where t = Either α β
  | ∀αβ.  Inr (Exp e β)
    where t = Either α β
  | ∀αβ.  Pair (Exp e α) (Exp e β)
    where t =(α,β)
  | ∀α.   Case (Exp e α) [Match e α t]

data Match e t' t =
    ∀γ. Match (Pat t' e  γ) (Exp γ t)
```

Fig. 4. Static semantics of L_1 expressions: $\Gamma \vdash e : \tau \subseteq \mathbb{G} \times \mathbb{E} \times \mathbb{T}$ and `Exp e t`.

is the whole λ-term well formed. Of course, the type `t` of the overall λ-abstraction must be equal to a function type between the domain and the codomain $(\alpha \to \beta)$.

```
data Exp e t       = ∀α. Case (Exp e α) [Match e α t] | ···
data Match e t' t = ∀γ. Match (Pat t' e  γ) (Exp γ t)
```

A case expression, with a type assignment `e`, of type `t` consists of a discriminated expression of type `Exp e` α, and a number of pattern matches. Each match consists of a pattern which augments the type assignment `e` to some type assignment γ, and a body which produces a value of type `t` in the type assignment γ. Since in each match the pattern can extend the environment differently, the extended environment, γ, is existentially quantified. This use of existential types allows us to give the same type to an entire collection of matches.

4 Dynamic Semantics

We illustrate meta-programming over typed object-language syntax by defining a series of interpreters for the language L_1. We are interested in applying *staging* to interpreters to obtain efficient implementations. The efficiency of such an implementation comes from eliminating *interpretive overhead* and *tagging overhead* from the interpreter. To demonstrate our technique: We sketch out a preliminary *tagged* dynamic semantics for L_1 to illustrate the concept of *tagging overhead.* (Section 4.1). We define an unstaged definitional interpreter. This interpreter avoids tagging altogether by the use of equality constraints (Section 4.2). We then stage the definitional interpreter (Section 4.3). Finally, we apply a *binding time improvement* to the staged interpreter (Section 4.4).

```
eval0 :: Exp e t → [V] → V
eval0 (Lit i _) env = VI i
eval0 (V var) env = evalVar0 var env
eval0 (App f x) env =
  unVF (eval0 f env) (eval0 x env)
eval0 (Abs pat e _) env =
  VF (\v →
     eval0 e (unJust(evalPat0 pat v env)))
```

```
data V = VF (V → V) | VI Int
       | VP V V | VL V | VR V
unVF (VF f) = f

evalVar0 :: Var e t → [V] → V
evalVar0 (Z _) (v:vs) = v
evalVar0 (S s _) (v:vs) = evalVar0 s vs

evalPat0::Pat t i o → V → [V] → Maybe [V]
evalPat0 (PVar _) v env = return (v:env)
```

Fig. 5. The tagging interpreter. These functions are purposely incomplete, and are given only to illustrate the use of tags.

4.1 The Tagging Interpreter

To illustrate the problem with tagging we write a dynamic semantics for L_1 as the function `eval0`. This interpreter uses a tagged value domain `V` which encodes in a single sum type all the possible values an L_1 program can return. Runtime environments are then represented by lists of these values. This scheme has been widely used for writing (interpreted) language implementations. In Figure 5 we give a sketch of how such an interpreter is implemented. "Unfolding" this interpreter on the input expression, `(app (abs (var z)) (lit 0))`, yields the following value: `unVF (VF (\v → v)) (VI 0)`

The unfolding (we ignore for a moment how such an unfolding can be achieved) has removed the recursive calls of `eval0`, but the program still contains the *tags* `VF`, `unVF` and `VI`. Such tags may indeed be necessary if the object language is untyped/dynamically typed. However, in our implementation, only well-typed L_1 expressions are ever interpreted. This means that the tagging and untagging operations in the residual programs never do any useful work, since the strong typing of the object language guarantees that no mismatch of tags ever occurs. Practical testing [4] has revealed that the performance penalty exacted on staged interpreters by unnecessary tags may be as high as a factor of 2-3 (in some cases even as high as 3-10 [17]).

4.2 The Tagless Interpreter

A dynamic semantics that takes advantage of well-typed object terms can be given in a "categorical style": by writing a set of semantic functions, one for each of the judgments.

```
eval    :: Exp e t → (e → t)
evalVar :: Var e t → (e → t)
evalPat :: Pat t ein eout → (t → ein → (Maybe eout → a) → a)
```

For example, the semantic function `eval` is defined inductively on the structure of typing judgments for expressions. Its meaning is an "arrow" (i.e., here a Haskell function) from the meaning of type assignments (the runtime environment) to the meaning of types. For example, the meaning of the type assignment $(\langle\rangle, \mathsf{Int}, \mathsf{Int}, \mathsf{Int} \to \mathsf{Int})$ is a Haskell value of the nested product type `((((),Int),Int),Int-> Int)`.

Before we proceed to define the semantics of various judgments of L_1, we digress briefly to discuss *effects* introduced by presence of patterns in the object language L_1. Pattern matching failure may manifest itself in two different (and related) ways:

(1) Global failure. Pattern matching may fail when matched against an incompatible value. This may occur, for example, in λ-expressions, such as $(\lambda(\mathsf{Inl}\,\bullet).\,\mathsf{Var}\ 0)\ (\mathsf{Inr}\ 10)$. In case of such a failure, the meaning of the program is undefined. In our implementation we will model the global failure by the undefined, bottom value in Haskell (the function `error`). *(2) Local failure.* Pattern matching may also fail in one or more alternatives in a `case` expression. Local failure may, or may not, be promoted into a global failure: if a pattern match in one arm of a case expression fails, the control should be passed to the next arm of the case expression, until one of them succeeds. If there are no more arms, a global failure should take place.

One way to model pattern matching failure is to use a *continuation.* The denotations of patterns that produce an environment of type `eout` are functions of type `(t -> ein -> (Maybe eout -> a) -> a)`. In other words, they take a value, and input environment, and a *continuation* κ which consumes the output environment and produces "the rest of the program" of type `a`. The argument to κ is a maybe type so that the continuation can decide how to continue in case of the success or failure of pattern matching.

We now define the function `eval` and its siblings `evalVar` and `evalPat` (we shall refer to the implementation in Figure 6). It is in these functions that the assumptions and obligations of judgment constructors play an essential role.

Expressions: `eval`. Let us look at several clauses of the function `eval` (Figure 6) to demonstrate the main points of our technique.

Literals (line 2). Consider the case of evaluating integer literals. Here we see the most elementary use of the equality constraints. The function `eval` must return a result of type `t`, but what we have is the integer `i`. However, pattern matching over the constructor `Lit` introduces the *fact* that `t = Int`. The Ωmega type checker uses this fact to prove that `i` indeed has the type `t`.

Abstraction (line 4). The abstraction case returns a function of type $\alpha \to \beta$, where α and β are the types of its domain and codomain. In the body of this function, its parameter `x` is matched against the pattern `pat`. The continuation `h` given to `evalPat` results in global failure if the pattern matching fails. If the pattern matching succeeds, it simply evaluates the function body with the extended environment produced by `evalPat`. Finally, the fact that `t` = $\alpha \to \beta$ is used to give the resulting function the required type `t`.

```
1   eval :: (Exp e t) → e → t
2   eval (Lit i) env = i
3   eval (V v) env = evalVar v env
4   eval (Abs pat exp) env =
5     (\x → evalPat pat x env h)
6      where h Nothing = error "Glob. Failure"
7            h (Just env) = eval exp env
8   eval (App f x) env = (eval f env)
9                        (eval x env)
10  eval (Pair x y) env =  (eval x env,
11                          eval y env)
12  eval (Case e branches) env =
13    (evalCase (eval e env) branches env)
14
15  evalVar :: (Var e t) → e → t
16  evalVar Z env = snd env
17  evalVar (S v) env = evalVar v (fst env)
18
19  evalPat :: (Pat t ein eout) → t → ein →
20            (Maybe eout → a) →  a
21  evalPat (PVar) v e k = k (Just (e,v))
22  evalPat (PInl pt) v e k =
23    case v of
24      Left x → evalPat pt x e k
25      Right _ → k Nothing
26  evalPat (PInr pt) v e k =
27    case  v of
28      Left _ → k Nothing
29      Right x → evalPat pt x e k
30  evalPat (PPair pat1 pat2) v e k =
31    case v of
32     (v1,v2) → evalPat pat1 v1 e h
33    where h Nothing = k Nothing
34          h (Just eout1) = evalPat pat2 v2
35                                eout1 k
36
37  evalCase :: t1 → [Match ein t1 t2] →
38              ein → t2
39  evalCase val [] env = error "Empty Case!"
40  evalCase val ((Match(pat,body)):rest) env =
41   (evalPat pat val env k)
42   where k Nothing = evalCase val rest env
43         k (Just env2) = eval body env2
```

Fig. 6. Tagless interpreter for L_1. The semantic functions operate over the structure of judgments.

Application (line 9). The function part of the application is evaluated, obtaining a function value of type $\alpha \rightarrow$ `t`; next, the argument is evaluated obtaining a value of type α. Finally the resulting function is applied, obtaining a result of type `t`. The function `eval` is a polymorphic recursive function with type `eval :: ((Exp e t)` $\rightarrow$ `e` $\rightarrow$ `t)`. It is called recursively at two different instances. This is frequently the case in this kind of meta-programming.

Case (line 12). The implementation of `case` involves an interesting interaction with the semantics of patterns, `evalPat`. The function `eval` first evaluates the discriminated expression `e`, and then calls the auxiliary function `evalCase` which matches each arm of the case against this value. The function `evalCase` examines a list of matches. If the list is empty, matching has failed with global failure. Otherwise, in each arm, `evalPat` matches the pattern against the discriminated value `val`. The function `evalPat` is given the continuation `k` as its argument. The continuation `k` proceeds to evaluate the body *if the pattern succeeds*, or simply calls `evalCase` recursively with the next pattern if the pattern matching fails (lines 42-43).

Variables: `evalVar` (*line 15*). The variable case of `eval` passes control directly to the function `evalVar`, which projects the appropriate value from the runtime environment. The base case for variables casts the runtime environment of type `e`, using the fact that `e=(`γ`,t)`, to obtain the pair `(`γ`,t)`. Then, the function `snd` is applied, to obtain the correct result value of type `t`. In the weakening case, the fact that `e=(`γ`,t')` is again used to treat the runtime environment as a pair. Then, the function `evalVar` is called recursively on the predecessor of the variable (`v`) index together with the first component of the pair (the sub-environment `(fst env)`).

Patterns: `evalPat`. The function `evalPat` has four arguments: a pattern judgment of type `(Pat t ein eout)`; a value of type `t` against which the pattern matching will be done; an *input* environment of type `ein`; and a continuation. The continuation either (a) consumes an extended output environment of type `eout` to produce some final result of type `a` or, (b) knows how to produce some alternative result of type `a` in case the pattern matching fails.

Variable patterns (line 21). For variable patterns, the value `v` is simply added to the environment `e`. This transforms the initial environment into a larger environment, which is then supplied to the continuation `k`.

Sum patterns (line 22). The case for `Inl` (`Inr`) patterns is more interesting. First, the value `v` is examined to determine whether it is the left (or right) injection of a sum. If the injection tag of the pattern does not match the injection tag of the value, the continuation `k` is immediately applied to `Nothing` indicating pattern match failure. If the injection tag matches, the sub-value `x` is selected from `v` and `evalPat` is recursively invoked with the sub-pattern `p`, the sub-value `x`, and the same continuation `k`. It is this recursive call to `evalPat` that binds variables.

Pair patterns (line 30). The case for pair patterns involves taking a pair value apart, and matching the left sub-pattern with the first element of the value. However, this invocation of `evalPat` is given an enlarged continuation `h`. The continuation `h` looks at its argument. If it is `Nothing` it immediately invokes the initial continuation `k` with `Nothing`, thus propagating failure that must have occurred during the pattern matching of the left sub-pattern. If, however, its argument is some output environment `eout1` obtained from matching the left sub-pattern, it recursively invokes `evalPat` on the right sub-pattern with the runtime environment `eout1` and the initial continuation `k`.

Note that the structure of the pattern judgment (specified in Figure 3) forces us to thread the environment correctly: were we to make the mistake of first evaluating the right-hand side pattern and threading its result to the evaluation of the left-hand side pattern, the types simply would not work out. This is an important advantage of using typed object-language representations: *meta-level types catch object-level semantic errors.*

Ubiquitous in these definitions is the implicit use of equality constraints, manipulated behind the scenes by the type checker to ensure that the functions are well-typed. In `eval`, for example, they are always used to show that although each equation in the definition of `eval` returns a value of a different type, all of those types can be made equal to the type `t` from `eval`'s type signature. Thus, the type checker automatically performs precisely the same role as *tags* in an interpreter which uses a universal domain. However, the crucial difference is that while tags are checked *dynamically*, at the runtime of the interpreter, the

```
1   evalS :: Exp e t → Code e → Code t
2   evalS (V v) env = evalVarS v env
3   evalS (Lit i) env =  [| i |]
4   evalS (Abs pat exp) env =
5    [| \x → $(evalPatS pat [|x|] env h) |]
6      where h Nothing = [| error "Failure" |]
7            h (Just env2) = evalS exp env2
8   ...  ...  ...
9
10  evalVarS :: Var e t → Code e → Code t
11  evalVarS Z env =  [| snd ($env) |]
12  evalVarS (S v) env =
13    [| ($(evalVarS v [|fst ($env)|])) |]
14
15  evalPatS :: (Pat t e1 e2) →
16              Code t → Code e1 →
17              ((Maybe (Code e2)) → Code a) →
18              Code a
19  evalPatS PVar v e k  =
20     k (Just [| ($([| ($e,$v) |])) |])
```

Fig. 7. The staged interpreter `eval`, take 1.

equality constraint manipulation is performed statically, at type checking time. Thus, unlike tags, it incurs no runtime performance penalty for the interpreter.

4.3 The Staged Interpreter

The interpreter presented in Figure 6 does not rely on a universal domain for its values. Rather, it starts from a well-typed judgment `(Exp e t)` and ultimately returns a value of type `t`: such use of polymorphism gives us, in effect, a whole family of `evals`, one for each resulting type. Instead of tags, the equality constraints partition this family of `evals`. Thus the tagging overhead has been removed, but the interpreted overhead of traversing the data representing the program remains. Staging can remove this overhead. It is straightforward to add staging to the interpreter of Figure 6: we modify the types of the interpreter, adding `Code` to the types of the runtime environment and the result. Thus, the new types of the modified semantic (interpreter) functions are changed as follows (we also change their names, appending "`S`" for "staged"):

```
evalS     :: Exp e t → Code e → Code t
evalVarS  :: Var e t → Code e → Code t
evalPatS  :: Pat t ein eout → Code t → Code ein →
             (Maybe (Code eout) → Code a) → Code a
```

Figure 7 gives the relevant definitions of the staged semantics. (Note that, due to limitations of space, we show only those parts of the implementation that differ from the final version of the interpreter in Figure 8). Consider the simplest case, that of literals (Figure 7, line 3). Two things are different from the unstaged version: First, the result of the function is a code value, enclosed in code brackets. Recursive calls to `evalS` are always escaped into a larger piece of code that is being built. Second, the type equalities of the form `t=Int`, introduced as facts by pattern-matching on the judgments (line 3), are used in a richer type context. For example, in line 3 the type checker "converts" a result of `Code Int` to type `Code t`.

Another thing to note is the slightly changed type of the continuation in `evalPat`. The continuation takes an argument of type `(Maybe (Code env))`, i.e.,

```
data PSE e = EMPTY where e = ()
           | ∀αβ. EXT (PSE α) (Code β)
                   where e=(α,β)

eval2S :: Exp e t → PSE e → Code t
eval2S (Lit i) env = [| i |]
eval2S (V v) env = evalV2S v env
eval2S (App e1 e2) env =
  [| $(eval2S e1 env) $(eval2S e2 env) |]
eval2S (EInl e) env =
  ([|Left  ($(eval2S e env))|])
eval2S (EInr e) env =
  ([|Right ($(eval2S e env))|])
eval2S (Abs pat body) env =
 ([| x → $(evalPat2S pat [|x|] env h)|])
 where h (Nothing) = [| error "fail" |]
       h (Just e) = eval2S body e
eval2S (ECase e matches) env = [|
let value = $(eval2S e env)
in  $(evalCase2S [|value|] matches env)|]

evalCase2S ::
Code t1 → [Match e t1 t2] → PSE e → Code t2
evalCase2S val [] env = [| error "fail" |]
evalCase2S val ((Match (pat,body)):rest) env =
 evalPat2S pat val env h
 where h (Nothing) = evalCase2S val rest env
       h (Just env2) = eval2S body env2

evalVar2S :: Var e t → PSE e → Code t
evalVar2S Z  (EXT _ b) = b
evalVar2S (S s) (EXT e _ ) = evalVar2S s e

evalPat2S :: Pat t ein eout → (Code t) →
   (PSE ein) →  (Maybe (PSE eout) → Code ans) →
   Code ans
evalPat2S PVar v ein k = k (Just  (EXT ein v))
evalPat2S (PInl pt) v ein k = [|
 case  $v of
 Left x  → $(evalPat2S pt [|x|] ein k)
 Right x → $(k Nothing) |]
evalPat2S (PInr pt) v ein k = [|
 case $v of
 Left x → $(k Nothing)
 Right x → $(evalPat2S pt [|x|] ein k) |]
evalPat2S (PPair pt1 pt2) v ein k =  [|
 case $v of
 (v1,v2) →
   $(evalPat2S pt1 [|v1|] ein (h [| v2 |]))|]
 where h n Nothing = k Nothing
       h n (Just eout1) =
         evalPat2S pt2 n eout1 k
```

Fig. 8. Binding time improved staged interpreter.

the success or failure portion of the argument is *static*, while the environment itself is dynamic. This means that we can *statically* generate both the success and failure branches of the pattern.

We will not further belabor the explication of this particular staged implementation, since we will rewrite and improve it in the following section. It is instructive, however, to examine the residual code produced by the interpreter from Figure 7 as this will motivate the improvements. Consider the source program $\lambda\bullet.\,\lambda\bullet.\,(\mathsf{Var}\ 0)\ (\mathsf{Var}\ 1)$. The code produced by the staged interpreter `evalS` is basically: `\x → \f →` $\boxed{\texttt{(snd (((),x),f))}}$ $\boxed{\texttt{(snd (fst (((),x),f)))}}$.

The two boxed expressions in the residual code above correspond to the variable case of `evalS`: as the interpreter descends under the abstraction terms, it builds ever larger runtime environments. At the variable use sites, the interpreter `evalVarS` then generates projection functions (e.g., `snd (fst ···)`) to project runtime values from these environments. This leads to variable lookup at runtime which is proportional to the length of the runtime environment, an instance of interpretive overhead *par excellence.*

4.4 Improved Staged Interpreter

The process of (slightly) changing the interpreter to make it more amenable to staging is known as *binding time improvement* [16]. In the remainder of this section, we will apply a binding time improvement to the staged interpreter for L_1 with the goal of removing the dynamic lookup mentioned above. The full implementation of the interpreter with the binding time improvement is given in Figure 8.

The previously presented staged interpreter fails to take advantage of the fact that the runtime environment is *partially static*. Namely, while the values in the environment are not known until stage one, the actual *shape* of the environment is known statically and depends only on the syntactic structure of the interpreted term. Therefore, we should be able to do away with the dynamic lookup of values in the runtime environment. The resulting interpreter should produce residual code for the above example that looks like this: `[| \x → \f → f x |]`. Recall that environments in the previous definitions of the interpreter are dynamic nested pairs of the form `[|((...,v2),v1)|]`. The corresponding partially static environment is a static nested pair tuple, in which each second element is a dynamic value: `((...,[|v2|]),[|v1|])`. This relationship between environment types and the corresponding partially static environments is encoded by the following datatype:

```
data PSE e  = EMPTY where e = ()
            | ∀αβ. EXT (PSE α) (Code β) where e=(α,β)
```

A partially static environment (hence, a PSE) can either empty (`EMPTY`), or it can be a PSE extended by a dynamic value. In a PSE, constructed using `EXT`, the *shape* of the environment is known statically, but the actual values that the environment contains are known only at the next stage. This allows us to perform the projections from the PSE statically, while the actual values projected are not known until runtime. The type equality constraint ensures that the type index `e` is identical in form (i.e. nested pairs) to the form of the environment argument of judgments (the `e` in `(Exp e t)` and `(Var e t)`). Now, we can give a new type to the interpreter, as follows:

```
    eval2S :: Exp e t → PSE e → Code t
    evalVar2S :: Var e t → PSE e → Code t
```

The interpreter now takes a judgment `(Exp e t)`, and a partially static environment `(PSE e)`, and produces a delayed result of type `(Code t)`. The largest change is in the evaluation function for variables, `evalVar2S`. The base case takes a zero variable judgment and a PSE `(EXT _ b)`, with `b::code` β, and introduces the equality `e=(`α`,`β`)`. From this fact, the type checker can easily conclude that β is equal to `t`. A simple congruence then further allows the type checker to conclude that `(Code` β`)` is equal to `(Code t)`. The inductive case is similar: the equality constraints introduced by the pattern matching are used to treat the environment as a pair, and then a sub-environment is projected from the environment and used in the recursive call to `evalVar2S`. Partially static environments are created in the `PVar` case of `evalPat2S`.

Now, if we consider the L_1 program $(\lambda\bullet.\lambda\bullet.(\mathsf{Var}\ 0)(\mathsf{Var}\ 1))$, the code generated for it by the interpreter in Figure 8 looks like this: `[|\x → \f → f x|]`. All the recursive calls to `eval` have been unfolded, all the dynamic lookups in the environment have been replaced by just variables in the residual program, and *there are no tags.*

5 The Region DSL: Embedded Implementation

The Region language is a simple DSL whose programs are descriptions of two-dimensional geometric regions. The example is just large enough to show the techniques involved.

5.1 Magnitudes

Recall the user-defined kind `Unit` we defined in Section 2.2. Using the kind `Unit`, we can first define a datatype of lengths indexed by a unit of measure:

```
data Mag u  =  Pix  Int    where u = Pixel
            |  Cm   Float  where u = Centimeter
```

Ωmega infers the kind of `Mag` to be `Unit -> *`. Note how the parameter to `Mag` is not of kind `*`, this allows the user to define domain-specific type systems for the DSL within the host languages type system. Magnitudes are lengths with an associated type (unit) index.

As we have seen earlier, equality constraints force the constructors `Pix` and `Cm` to produce values of type `Mag` annotated by the appropriate unit of measurement. Here are some programs manipulating `Mag u` values:

```
scale = 28

pxTOcm          :: Mag Pixel → Mag Centimeter
cmTOpx          :: Mag Centimeter → Mag Pixel

pxTOcm (Pix i)  = Cm (intToFloat (div i scale))
cmTOpx (Cm f)   = Pix (round (f #* intToFloat scale))
```

A note on Ωmega arithmetical operator names: built-in arithmetic functions on floats are prefixed with a hash mark (`#`). Note that these equations define total functions even though the case analysis is not exhaustive. This is because clauses such as, for example, `pxTOcm (Cm f) = ...` would be ill-typed because the obligation `u = Centimeter` cannot be discharged given the assumption `u = Pixel`. For this reason, the type parameter `u` in `Mag u` acts more like a *type index* than a usual parameter.

We lift the following arithmetic operations to be used on Magnitudes.

```
neg                  :: Mag u → Mag u
plus, minus, times   :: Mag u → Mag u → Mag u
leq                  :: Mag u → Mag u → Bool
```

The lifting is straightforward, for example:

```
times (Pix a)  (Pix b)  = Pix (a*b)
times (Cm a)   (Cm b)   = Cm (a#*b)
```

We will also need some other derived operators.

```
square a             = times a a
between a b c        = leq a b && leq b c
```

5.2 Regions

Now we can define an embedded implementation of the Region language. The meaning of a region is a set of points in the two-dimensional plane. In this style of implementation, we shall represent a region by its characteristic function.

```
type Region u = Mag u → Mag u → Bool
```

Primitive regions The Region language supports four primitive regions, circles of a given radius (centered at the origin), rectangles of a given width and height (centered at the origin), the all-inclusive universal region, and the empty region.

```
circle     :: Mag u → Region u
rect       :: Mag u → Mag u → Region u
univ       :: Region(u)
empty      :: Region(u)

circle r   = \x y → leq  (plus (square x) (square y)) (square r)
rect w h   = \x y →  between (neg w) (plus x x) w &&
                      between (neg h) (plus y y) h
univ       = \x y → True
empty      = \x y → False
```

Region combinators The Region language supports the following combinators for manipulating regions: The combinator `trans` (δ_x, δ_y) translates a region by the given amounts along both the `X` and `Y` axes; The combinator `convert t` changes the units used to describe a region according to `t`. The combinators `intersect` and `union` perform set intersection and set union, respectively, on regions considered as sets of points.

```
trans            :: (Mag u,Mag u) -> Region u -> Region u
convert          :: (Mag v -> Mag u) -> Region u -> Region v
intersect        :: Region u -> Region u -> Region u
union            :: Region u -> Region u -> Region u

trans (a,b) r     = \x y -> r (minus x a) (minus y b)
convert t r       = \x y -> r (t x) (t y)
intersect r1 r2   = \x y -> r1 x y && r2 x y
union r1 r2       = \x y -> r1 x y || r2 x y
```

6 Intensional implementation

One problem with the embedded implementation of the previous section is that we can not write programs that manipulate the *intensional form* of a region. Such programs include optimizing transformations, pretty printers, and program analyses (such as computing a bounding box for a region). Because regions are represented by their denotations as functions, their structure is opaque to the rest of the program. The solution to removing this opacity is to use a data structure to represent regions. Consider the following naïve attempt at defining such a data structure.

```
data Len = PixLen Int | CmLen Float

data RegExp  = Univ
             | Empty
             | Circle Len
             | Rect Len Len
             | Union RegExp RegExp
             | Inter RegExp RegExp
             | Trans (Len,Len) RegExp
             | Convert CoordTrans RegExp

data CoordTrans = CM2PX | PX2CM
```

While solving one problem, this introduces another: meta-programs can now build ill-typed object programs. Here is one such meaningless program:

```
Convert CM2PX (Convert CM2PX Univ)
```

This is a step backwards from the embedded approach, in which object-language type errors show up as meta-language type errors. Can we achieve both intensional manipulation and object-level type safety? Yes – by using type-equality constraints. This allows us to define an `Unit` indexed intensional representation that captures all the object-level type information present in the embedded function approach.

```
data RegExp u
      =       Univ
      |       Empty
      |       Circle (Mag u)
      |       Rect (Mag u) (Mag u)
      |       Union (RegExp u) (RegExp u)
      |       Inter (RegExp u) (RegExp u)
      |       Trans (Mag u, Mag u) (RegExp u)
      | ∀ v.  Convert (CoordTrans v u) (RegExp v)
```

```
data CoordTrans u v
       =  CM2PX where u = Centimeter,  v = Pixel
       |  PX2CM where u = Pixel,       v = Centimeter
```

This representation enforces typing constraints as in the embedded approach but also supports intensional analysis of Region expressions.

6.1 Type relations

The definition of the datatype `CoordTrans` is interesting in its own right, demonstrating the technique of embedding *type relations* in types of the host language. The relation encoded by `CoordTrans` would be written in more traditional mathematical notation as

$$\{(Centimeter, Pixel), (Pixel, Centimeter)\}$$

The elements `CM2PX` and `PX2CM` are *witnesses* to the relation. Pattern matching over such witnesses allows the type inference to locally make use of the fact that the relation holds. Consider the following program:

```
opp :: CoordTrans a b -> CoordTrans b a
opp CM2PX = PX2CM
opp PX2CM = CM2PX
```

While type checking the body of the first equation, we have the assumptions `a = Centimeter` and `b = Pixel` at our disposal. We need to check that `PX2CM` has type `CoordTrans b a`. This check results in the obligations `b = Pixel` and `a = Centimeter`, precisely the assumptions we have at our disposal.

The simplest type relation just encodes a single type-equality constraint. For this purpose, the Ωmega prelude defines the following datatype:

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

A value of type `Eq a b` is a token representing the fact that `a` equals `b`. We can make use of this fact by pattern matching the value against the pattern `Eq`. We use this technique later. For now, just note that building values of type `Eq a b` is a way to inform the type checker of a fact it wasn't already aware of. The following function shows how one might construct an `Eq a b` value.

```
twoPoint :: CoordTrans a b -> CoordTrans b c -> Eq a c
twoPoint CM2PX PX2CM = Eq
twoPoint PX2CM CM2PX = Eq
```

Once again, these equations are exhaustive because no other combinations type check. In the first equation, we have the assumptions `a = Centimeter`, `b = Pixel`, `c = Centimeter` available when discharging the proof obligation `a = c`.

6.2 Interpreters for free

We can reuse the combinators from the embedded implementation to define an interpreter for this representation of region expressions. Notice how easy it is to define `interp`: just replace each data constructor with its corresponding combinator.

$$\text{univ} \cup r = \text{univ} \qquad \text{empty} \cap r = \text{empty}$$

$$\text{empty} \cup r = r = \text{univ} \cap r \qquad (r_1 \cup r_2) \cap (r_1 \cup r_3) = r_1 \cup (r_2 \cap r_3)$$

$$\text{conv } \tau_{12} \text{ (conv } \tau_{21} \ r) = r \qquad \text{conv } \tau_{12} \ (r_1 \cup r_2) = \text{conv } \tau_{12} \ r_1 \cup \text{trans } \tau_{12} \ r_2$$

$$\text{trans } \delta \text{ (trans } \delta' \ r) = \text{trans } (\delta + \delta') \ r \qquad \text{trans } \delta \ (r_1 \cup r_2) = \text{trans } \delta \ r_1 \cup \text{trans } \delta \ r_2$$

Fig. 9. Some algebraic identities on region expressions.

```
interp                          :: RegExp u -> Region u
interp Univ                     = univ
interp Empty                    = empty
interp (Circle r)               = circle r
interp (Rect w h)               = rect w h
interp (Union a b)              = union (interp a) (interp b)
interp (Inter a b)              = intersect (interp a) (interp b)
interp (Trans xy r)             = trans xy (interp r)
interp (Convert trans r)        = convert (interpTrans trans) (interp r)

interpTrans                     :: CoordTrans u v -> Mag v -> Mag u
interpTrans CM2PX               = pxTOcm
interpTrans PX2CM               = cmTOpx
```

6.3 A simple optimizer

Region expressions have a rich algebraic structure. Figure 9 show a handful of identities over region expressions. These identities constitute *domain-specific knowledge* and can be used to optimize object programs for performance. We illustrate the utility of the typed intensional region representation for program transformation by writing a function `simplify` that applies domain-specific region equalities like those in Figure 9 to simplify a region expression.

```
simplify :: RegExp u -> RegExp u
simplify (Union a b)    = mkUnion (simplify a) (simplify b)
simplify (Inter a b)    = mkInter (simplify a) (simplify b)
simplify (Trans xy r)   = mkTrans xy (simplify r)
simplify (Convert t r)  = mkConvert t (simplify r)
simplify t              = t
```

The function `simplify` works by replacing (in a bottom up fashion) each data constructor in a Region expression with its corresponding “smart” constructor. The smart constructor “implements” the algebraic identites for that combinator. Here is the smart constructor for `Convert` expressions.

```
mkConvert :: CoordTrans u v -> RegExp u -> RegExp v
mkConvert t  (Convert t’ r)     = case twoPoint t’ t of Eq -> r
mkConvert t  (Univ)             = Univ
mkConvert t  (Empty)            = Empty
mkConvert t  (Union a b)        = Union  (mkConvert t a)
                                         (mkConvert t b)
mkConvert t  (Inter a b)        = Inter  (mkConvert t a)
                                         (mkConvert t b)
```

```
mkConvert t  (Circle rad)       = Circle (convMag t rad)
mkConvert t  (Rect w h)         = Rect (convMag t w) (convMag t h)
mkConvert t  (Trans (dx,dy) r)  = Trans (convMag t dx,convMag t dy)
                                        (mkConvert t r)
```

```
convMag :: CoordTrans u v -> Mag u -> Mag v
convMag t a = interpTrans (opp t) a
```

In the first equation, we have the typing assignments `t :: CoordTrans u v`, `t' :: CoordTrans v' u`, and `r :: RegExp v'` for some arbitrary (Skolem) type `v'`. We want to return `r`, but the type checker doesn't know that it has the right type, `RegExp v`. In order to inform the type checker of this fact we build a `Eq v' v` witness using the previously defined function `twoPoint` and pattern match it against `Eq`. In the scope of the pattern, the type checker knows `v' = v` and uses this fact to deduce `r :: RegExp v`. An important advantage of using a typed intensional representation is that manipulations of Region expressions must obey the typing invariants of the Region language as captured in the definition of `RegExp`. It is impossible to write a domain-specific optimization function that violates the typing discipline of the object language.

7 Staged implementation

The intensional approach of the previous section introduces a layer of interpretive overhead into the implementation (a problem we encountered in Section 4.1). In the embedded approach this extra overhead was not present. To remove this overhead [4] we employ the staging techniques already demonstrated in Section 4.3.

The goal of this section is to stage the interpreter `interp` to obtain a simple compiler `comp`. The transition from `interp` to `comp` is accomplished in a few simple steps:

1. Stage the `Mag` datatype so that it contains `(Code Int)` instead of `Int` and `(Code Float)` instead of `Float`. This allows us to know the units of a magnitude at "compile time", even though we will not know its value until "run time". The definition of `CodeMag` accomplishes this staging.

```
data CodeMag u
      =  CodePix  (Code Int)    where u = Pixel
      |  CodeCm   (Code Float)  where u = Centimeter
```

2. Stage the primitive functions on `Mag` values to work with this new type. We follow the convention that the staged versions of helper functions are given the same name but suffixed with an "S".

```
pxTOcmS             :: CodeMag Pixel -> CodeMag Centimeter
cmTOpxS             :: CodeMag Centimeter -> CodeMag Pixel
```

```
pxTOcmS (CodePix i)  =
          CodeCm [| intToFloat (div $i $(lift scale)) |]
cmTOpxS (CodeCm f)   =
          CodePix [| round ($f #* $(lift (intToFloat scale)))|]
```

Note that these definitions differ from their unstaged counterparts only in the addition of the staging annotations `[| _ |]` , `$(_)`, and `(lift _)`.

```
timesS :: CodeMag u -> CodeMag u -> CodeMag u
timesS (CodePix a)  (CodePix b)  = CodePix  [| $a * $b |]
timesS (CodeCm a)   (CodeCm b)   = CodeCm   [| $a #* $b |]

squareS a      = timesS a a
betweenS a b c = [| $(leqS a b) && $(leqS b c) |]
```

Each of these staged functions is derived from the original in a systematic way. In fact our work on using type annotations as specifications to binding time analysis [18,19] leads us to believe this process can be completely automated. We also define a lifting function for magnitudes:

```
liftMag          :: Mag u -> CodeMag u
liftMag (Pix i)  = CodePix (lift i)
liftMag (Cm i)   = CodeCm (lift i)
```

3. Stage the `Region` type so that it is defined in terms of `CodeMag` instead of `Mag`. The definition of `CReg` (Compiled `Region`) accomplishes this staging.

```
type CRegion u = CodeMag u -> CodeMag u -> Code Bool
```

4. Redefine the region combinators and interpreter `interp` to manipulate `CReg` values instead of `Region` values. The updated definition is given in terms of the staged primitives from step 2. This is accomplished by `comp` and its helper functions. First we stage the primitive region constructors:

```
circleS  :: Mag u -> CRegion u
rectS    :: Mag u -> Mag u -> CRegion u
univS    :: CRegion u
emptyS   :: CRegion u

circleS r  x y  = leqS  (plusS (squareS x) (squareS y))
                        (liftMag (square r))
rectS w h  x y  =
    [|  $(betweenS (liftMag (neg w)) (plusS x x) (liftMag w)) &&
        $(betweenS (liftMag (neg h)) (plusS y y) (liftMag h)) |]
univS      x y  = [| True |]
emptyS     x y  = [| False |]
```

Then we stage the region combinators:

```
transS      :: (Mag u,Mag u) -> CRegion u -> CRegion u
convertS    :: (CodeMag v  -> CodeMag u) ->
                             CRegion u -> CRegion v
intersectS  :: CRegion u -> CRegion u -> CRegion u
unionS      :: CRegion u -> CRegion u -> CRegion u

transS (dx,dy) r  x y = r (minusS x dx) (minusS y dy)
convertS t r      x y = r (t x) (t y)
intersectS r1 r2  x y = [| $(r1 x y) && $(r2 x y) |]
unionS r1 r2      x y = [| $(r1 x y) || $(r2 x y) |]
```

Then we stage the interpreter:

```
comp                    :: RegExp u -> CRegion u
comp Univ               = univS
comp Empty              = emptyS
comp (Circle r)         = circleS r
comp (Rect w h)         = rectS w h
comp (Union a b)        = unionS (comp a) (comp b)
comp (Inter a b)        = intersectS (comp a) (comp b)
```

```
comp (Trans xy r)       = transS xy (comp r)
comp (Convert trans r)  = convertS (compTrans trans) (comp r)

compTrans :: CoordTrans u v -> CodeMag v -> CodeMag u
compTrans CM2PX  = pxTOcmS
compTrans PX2CM  = cmTOpxS
```

The structure of each staged definition is the same as the unstaged definition modulo staging annotations. Once we stage the types to reflect what information should be static (known at compile time) versus dynamic (known at run time), the changes made to the definitions are completely systematic. In fact, we have had some success in building automatic tools to accomplish this [18,19]. These tools borrow from research on staging and binding time analysis from the partial evaluation community. We believe it is possible to completely automate this task once the types are staged.

8 Example Region language program

The DSL is still embedded, so we can apply the full power of the host language in constructing region programs. For example, the code below constructs three adjacent circles. The resulting data structure `r` is relatively large and complicated.

```
f         :: [RegExp Centimeter] -> RegExp Centimeter
f []      = Empty
f (x:xs)  = Union x (Trans (Cm 2.0,Cm 0.0) (f xs))

c  = Circle (Cm 1.0)
r  = Convert CM2PX (Trans (Cm (#- 2.0),Cm 0.0) (f [c,c,c]))
```

```
prompt> r
(Convert CM2PX (Trans ((Cm -2),(Cm 0))
    (Union (Circle (Cm 1))
           (Trans ((Cm 2),(Cm 0))
                  (Union (Circle (Cm 1))
                         (Trans ((Cm 2),(Cm 0))
                                (Union (Circle (Cm 1))
                                       (Trans ((Cm 2),(Cm 0)) Empty)
                                )))))) : RegExp Pixel
```

The intensional representation allows for domain-specific transformations like our `simplify` function. Such transformations can dramatically reduce the size and complexity of the region's representation.

```
prompt> simplify r
(Union (Trans ((Pix -56),(Pix 0))
              (Circle (Pix 28)))
       (Union (Trans ((Pix 0),(Pix 0))
                     (Circle (Pix 28)))
              (Trans ((Pix 56),(Pix 0))
                     (Circle (Pix 28))))) : RegExp Pixel
```

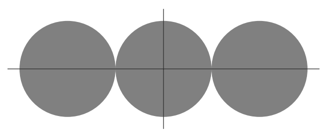
Staging allows the removal of all interpretive overhead incurred by creating an intensional representation, so that our technique is no less efficient than the embedded combinator approach (in fact, once the domain-specific optimizations are used, DSL programs can be implemented *more* efficiently this way).

```
prompt> [| \(Pix a) (Pix b) ->
           $(comp (simplify r) (CodePix [|a|]) (CodePix [|b|])) |]
[| \ (Pix x) (Pix y) ->
     (x + 56) * (x + 56) + (y * y) <= 784 ||
     (x * x) + (y * y) <= 784 ||
     (x - 56) * (x - 56) + (y * y) <= 784
|] : Code ((Mag Pixel) -> (Mag Pixel) -> Bool)
```

A further optimization phase could eliminate common sub-expressions.

```
[| \ (Pix x) (Pix y) ->
     let a = x + 56 in
       let b = 784 - (y * y) in
         let c = x - 56 in
           a * a <= b || x * x <= b || c * c <= b
|] : Code ((Mag Pixel) -> (Mag Pixel) -> Bool)
```

The Region DSL can be interpreted in other ways (besides its denotation as a characteristic function). For example, we can translate Region expressions into PostScript. The PostScript for the following picture was generated by running an alternative interpreter in Ωmega on the region program `r` defined above.



9 Related Work

Interpreters with Equality Proofs. Implementations of simple interpreters that use equality proof objects implemented as Haskell datatypes, have been given by Weirich [8] and Baars and Swierstra [7]. Baars and Swierstra use an untyped syntax, but use equality proofs to encode dynamically typed values.

Phantom Types. Hinze and Cheney [20,2] have recently resurrected the notion of "phantom type," first introduced by Leijen and Meijer [21]. Hinze and Cheney's phantom types are designed to address some of the problems that arise when using equality proofs to represent type-indexed data. Their main motivation is to provide a language in which polytypic programs, such as generic traversal operations, can be more easily written. Cheney and Hinze's system bears a strong similarity to Xi et al.'s *guarded recursive datatypes* [22], although it seems to be a little more general.

We adapt Cheney and Hinze's ideas to meta-programming and language implementation. We incorporate their ideas into a Haskell-like programming language. The value added in our work is additional type system features (user-defined kinds and arbitrary rank polymorphism, not used in this paper) applying these techniques to a wide variety of applications, including the use of typed syntax, the specification of semantics for patterns, and its combination with staging to obtain tagless interpreters, and the encoding of logical framework style judgments as first class values within a programming language.

Tagless Interpreters and Typeful Object-language Representations. The problem of tags in interpreters has been addressed in a number of settings. Tag elimination [4] is, in terms of its results, the closest to this work. In this approach, a separate tag elimination phase is introduced into the meta-language. It transforms a tagged residual program into a tagless one, guaranteeing that the meaning of the residual program is preserved. There are two major drawbacks compared to the approach outlined in this paper. First, a separate meta-theoretic proof is required to show that tags will be eliminated from a particular object language interpreter – there is no such guarantee statically. As our sample implementation shows, the staged interpreter we presented is *tagless by construction*. Second, more particularly, we know of no tag elimination for an object language with patterns.

The technique of manipulating well-typedness judgments has been used extensively in various logical frameworks [23,24]. We see the advantage of our work here in translating this methodology into a more mainstream functional programming idiom. Although our examples are given in Ωmega, most of our techniques can be adapted to Haskell with some fairly common extensions. Tagless interpreters can easily be constructed in dependently typed languages such as Coq [25] and Cayenne [26]. These languages, however, do not support staging, nor have they gained a wide audience in the functional programming community. Construction of tagless staged interpreters has been shown possible in a meta-language (provisionally called MetaD) with staging and dependent types [5]. The drawback of this approach is that there is no "industrial strength" implementation for such a language. In fact, the technique presented in this paper is basically the same, except that instead of using a hypothetical dependently typed language, we encode the necessary machinery in a language which extend Haskell only minimally. By using explicit equality types, everything can be encoded using the standard GHC extensions to Haskell 98.

A technique using *indexed type systems* [12], a restricted and disciplined form of dependent typing, has been used to write interpreters and source-to-source transformations on typed terms [22]. Recently, Xi and Chen have used an encoding similar to ours as a basis for meta-programming [27]: they provide a meta-theoretical translation which embeds MetaML-style code into a datatype similar to `Exp`: this allows them to use staging syntax, but manipulate well-typed terms judgments "under the hood." Where our work is different and complementary is (a) that we consider an object language with patterns; (b) we use a similar encoding to accomplish a slightly different goal (i.e., rather then give it as a meta-theory for staged programming, we use staged programming to obtain efficient tagless interpreters). It would be interesting to see how our interpreter could be implemented in their meta-language λ_{code}.

Domain Specific Langauges. Hudak introduced the notion of a "domain-specific embedded language" [28]. He argues that DSLs are "the ultimate abstraction", capturing precisely a certain domain of calculation, and suggests embedding a

DSL in a host language has the benefits of inheriting the infrastructure of the host language.

The Pan compiler of Elliot et al. [29] uses similar techniques to ours here. They use phantom types to partially ensure the well-typedness of object programs. The safety guarantees are only as strong as the discipline of the programmer who chooses to use the type safe interface constructors. The type system of Ωmega, however, enforces the same constraints at the level of the actual type constructors. Rhiger [30] proves that this style of programming provides safety guarantees about programs that *build* object programs, but notes that all type information is lost when deconstructing object programs in this way. This means that the type checker may not accept certain well-typed transformations of object programs. Cheney and Hinze motivate their work with a similar line of reasoning.

10 Discussion and Future Work

Finally, we discuss some outstanding issues and identify areas of future work.

Constructing Judgments and Other Applications. One final important consideration is whether arbitrary typing judgments could be constructed *at runtime*. In other words, could we write a *parsing* function that takes a string (or other) representation of object-language programs and compute a typing judgment? The problem is that the type of the judgment must be known *statically*, since, strictly speaking, judgments for different object-language terms have different types. Fortunately, there is a technique which allows us to construct just the kind of parsing functions discussed above. The key to this technique is to use an existential type, where the parsing function takes a textual representation of a program and constructs a judgment of the type $\exists\tau.(\texttt{Exp e }\tau)$. We do not have the space here to further discuss the particulars here.

Another interesting set of meta-programs is *source-to-source* transformations. In our setting, source-to-source transformations manipulate proofs of typing-judgments – the Ωmega type system guarantees that all such transformations respect the object-language types. Section 6.3 shows one transformation: an optimizer for a simple DSL. Others we were able to implement, but do not discuss here, include (1) substitution of well-typed terms of type `t` for a free variable of type `t` and (2) a big-step evaluator. The key to implementing substitution is to define an encoding for well-typed substitution judgments, as in the typed calculi with explicit substitutions [31, for example].

Meta-language Implementation. The meta-language used in this paper can be seen as a (conservative) extension of Haskell, with built-in support for equality types. It was largely inspired by the work of Cheney and Hinze. The meta-language we have used in our examples in this papers is the functional language Ωmega, a language designed to be as similar to Haskell. We have implemented our own Ωmega interpreter, similar in spirit and capabilities to the Hugs interpreter for Haskell [32]. Recent work on adding staging constructs to Haskell (albeit in a slightly different way [15]) or Objective Caml [14,33] indicate that adding staging to industrial strength functional language implementation is feasible. Theoretical work demonstrating the consistency of type equality support in a functional language has been carried out by Cheney and Hinze. We have implemented these type system features into a type inference engine, combining it with an equality decision procedure to manipulate type equalities. The resulting implementation has seen a good deal of use in practice, but more rigorous formal work on this type inference engine is indicated.

Ωmega as a DSL Implementation Platform. We have shown how several features of Ωmega allow the programmer to lower the cost of DSL implementations by bridging the gap between embedded and intensional DSL representations. We have advocated the use of well-typed object-language syntax representations, allowing the implementors to write both type safe optimizations and translations into other target languages.

Though the Region language is quite simple, the ideas presented here scale up to larger expression languages with variables and environments. At OGI, we have used these techniques to capture security type systems, temporal properties of APIs, closedness of code, and others.

Polymorphism and Binding Constructs in Types. The language L_1, presented in this paper, is simply typed: there are no binding constructs or structures in any index arguments to `Exp`. If, however, we want to represent object languages with universal or existential types, we will have to find a way of dealing with *type constructors* or *type functions* as index arguments to judgments, which is difficult to do in Haskell or Ωmega. We are currently working on extending the Ωmega type system to do just that. This would allow us to apply our techniques to object languages with more complex type systems (e.g. polymorphism, dependent types, and so on).

Logical Framework in Ωmega. The examples presented in this paper succeed because we manage to encode the usual logical framework style of inductive predicates into the type system of Ωmega. We have acquired considerable experience in doing this for typing judgments, lists with length, logical propositions, and so on. What is needed now is to come up with a formal and general scheme of translating such predicates into Ωmega type constructors, as well as to explore the range of expressiveness and the limitations of such an approach. We intend to work on this in the future.

11 Acknowledgment

The work described in this paper is supported by the National Science Foundation under the grant CCR-0098126. We also wish to thank our thesis advisor Tim Sheard for countless hours of discussion on these and similar topics.

References

1. Sheard, T., Pasalic, E., Linger, N.: The Ωmega implementation. Available on request from the authors. (2003)

2. Cheney, J., Hinze, R.: First class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cis/TR2003-1901 (2003)
3. Sheard, T., Benaissa, Z., Pašalić, E.: DSL implementation using staging and monads. In: Second Conference on Domain-Specific Languages (DSL'99), Austin, Texas, USEUNIX (1999)
4. Taha, W., Makholm, H., Hughes, J.: Tag elimination and Jones-optimality. Lecture Notes in Computer Science **2053** (2001) 257–275
5. Pašalić, E., Taha, W., Sheard, T.: Tagless staged interpreters for typed languages. In: The International Conference on Functional Programming (ICFP '02), Pittsburgh, USA, ACM (2002)
6. de Bruijn, N.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagaciones Mathematische **34** (1972) 381–392 This also appeared in the Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen, Amsterdam, series A, 75, No. 5.
7. Baars, A.I., Swierstra, S.D.: Typing dynamic typing. In: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002. SIGPLAN Notices 37(9), ACM Press (2002)
8. Weirich, S.: Type-safe cast: functional pearl. In: Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00). Volume 35.9 of ACM Sigplan Notices., N.Y., ACM Press (2000) 58–67
9. Nordström, B., Peterson, K., Smith, J.M.: Programming in Martin-Lof's Type Theory. Volume 7 of International Series of Monographs on Computer Science. Oxford University Press, New York, NY (1990) Currently available online from first author's homepage.
10. Jones, M.P.: Qualified Types: Theory and Practice. PhD thesis, Oxford University (1992) Also available as Programming Research Group technical report 106.
11. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. Journal of the ACM **27** (1980) 356–364
12. Xi, H., Pfenning, F.: Dependent types in practical programming. In ACM, ed.: Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas. ACM SIGPLAN Notices, New York, NY, ACM, ACM Press (1999) 214–227
13. Taha, W., Sheard, T.: MetaML: Multi-stage programming with explicit annotations. Theoretical Computer Science **248** (2000)
14. Taha, W., Calcagno, C., Huang, L., Leroy, X.: MetaOCaml: A compiled, type-safe multi-stage programming language. Available from http://cs-www.cs.yale.edu/homes/taha/MetaOCaml/ (2001)
15. Sheard, T., Jones, S.P.: Template meta-programming for Haskell. In: ACM SIGPLAN Workshop on Haskell, ACM Press (2002) 1–16
16. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall (1993) Avaiable online from http://www.dina.dk/ sestoft.
17. Huang, L., Taha, W.: A practical implementation of tag elimination. In preparation (2002)
18. Sheard, T., Linger, N.: Search-based binding time analysis using type-directed pruning. In: Proceedings of the ACM SIGPLAN Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation (ASIA-PEPM), ACM Press (2002) 20–31

19. Linger, N., Sheard, T.: Binding-time analysis for MetaML via type inference and constraint solving. Proceedings of TACAS '04 (2004)
20. Cheney, J., Hinze, R.: A lightweight implementation of generics and dynamics. In: Proc. of the workshop on Haskell, ACM Press (2002) 90–104
21. Leijen, D., Meijer, E.: Domain-specific embedded compilers. In: Proceedings of the 2nd Conference on Domain-Specific Languages, Berkeley, CA, USENIX Association (1999) 109–122
22. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In Norris, C., Fenwick, J.J.B., eds.: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL-03). Volume 38, 1 of ACM SIGPLAN Notices., New York, ACM Press (2003) 224–235
23. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. In: Proceedings Symposium on Logic in Computer Science, Washington, IEEE Computer Society Press (1987) 194–204 The conference was held at Cornell University, Ithaca, New York.
24. Pfenning, F., Schürmann, C.: System description: Twelf — A meta-logical framework for deductive systems. In Ganzinger, H., ed.: Proceedings of the 16th International Conference on Automated Deduction (CADE-16). Volume 1632 of LNAI., Berlin, Springer-Verlag (1999) 202–206
25. Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J., Giménez, E., Herbelin, H., Huet, G., Muñoz, C., Murthy, C., Parent, C., Paulin, C., Saïbi, A., Werner, B.: The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA (1997)
26. Augustsson, L., Carlsson, M.: An exercise in dependent types: A well-typed interpreter. In: Workshop on Dependent Types in Programming, Gothenburg (1999) Available online from `www.cs.chalmers.se/~augustss/cayenne/interp.ps`.
27. Chen, C., Xi, H.: Meta-programming through typeful code representation. In: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, Uppsala, Sweden (2003) 275–286
28. Hudak, P.: Building domain-specific embedded languages. ACM Computing Surveys **28** (1996) 196
29. Elliott, C., Finne, S., de Moor, O.: Compiling embedded languages. Journal of Functional Programming **8** (1998) 543–572
30. Rhiger, M.: A foundation for embedded languages. ACM Transactions on Programming Languages and Systems TOPLAS **25** (2003) 291–315
31. Benaissa, Z.E.A., Briaud, D., Lescanne, P., Rouyer-Degli, J.: $\lambda\nu$, a calculus of explicit substitutions which preserves strong normalisation. Journal of Functional Programming **6** (1996) 699–722
32. Jones, M.P.: The Hugs 98 user manual (2000)
33. Calcagno, C., Taha, W., Huang, L., Leroy, X.: Implementing multi-stage languages using ASTs, gensym, and reflection. In Czarnecki, K., Pfenning, F., Smaragdakis, Y., eds.: Generative Programming and Component Engineering (GPCE). Lecture Notes in Computer Science, Springer-Verlag (2003)