Meta-programming With Built-in Type Equality

[Extended Abstract]

Tim Sheard*
OGI
sheard@cse.ogi.edu

Emir Pasalic†
OGI
pasalic@cse.ogi.edu

ABSTRACT

We report our experience with exploring a new point in the design space for formal reasoning systems: the development of the programming language Ωmega. Ωmega is intended as both a practical programming language and a logic. The main goal of Ωmega is to allow programmers to describe and reason about semantic properties of programs from within the programming language itself, mainly by using a powerful type system.

We illustrate the main features of Ωmega by developing an interesting meta-programming example. First, we show how to encode a set of well-typed simply typed λ-calculus terms as an Ωmega data-type. Then, we show how to implement a substitution operation on these terms that is guaranteed by the Ωmega type system to preserve their well-typedness.

1. INTRODUCTION

There is a large semantic gap between what a programmer knows about his program and the way he has to express this knowledge to a formal system for reasoning about that program. While many reasoning tools are built on the Curry-Howard isomorphism, it is often hard for the programmers to conceptualize how they can put this abstraction to work. We propose the design of a language that makes this important isomorphism concrete – proofs are real object that programmers can build and manipulate without leaving their own programming language.

We have explored a new point in the design space of formal reasoning systems and developed the programming language Ωmega. Ωmega is *both* a practical programming language *and* a logic. These sometimes irreconcilable goals are made possible by embedding the Ωmega logic in a type system based on *equality qualified* types[6]. This design supports the construction, maintenance, and propagation of semantic properties of programs using powerful old ideas about types in novel ways.

For what kind of programming would a language like Ωmega be useful? The rest of this paper describes one possibility.

Meta-programming in Ωmega. Meta-programs manipulate object-programs represented as data. Traditionally, object-language programs are represented with algebraic data-types as *syntactic objects*. This representation preserves syntactic properties of object-language programs (i.e., it is impossible to represent syntactically incorrect object-language programs). In this paper, we explore the benefits of representing object-language programs as data in a manner that preserve important *semantic properties*, in particular *scoping* and *typing*. Representing typed object-languages in a way which preserves semantic properties can lead to real benefits. By preserving typing and scoping properties, we gain assurance in the correctness of a particular language processor (e.g. compiler, interpreter, or program analysis). Such semantics preserving representations statically catch errors introduced by incorrect meta-language programs.

Contributions. The first contribution is an approach to manipulating strongly typed object languages in a manner which is semantics preserving. This approach encodes *well-typed* and *statically scoped* object-language programs as data-types which embed the type of the object-language program in the type of its representation. While this can be done using only the standard extensions to the Haskell 98 type system (using equality types), we use Ωmega, an extension to Haskell inspired by Cheney and Hinze's work on phantom types [6].

The second contribution is an implementation of Cheney and Hinze's ideas that makes programming with well-typed object-language programs considerably less tedious than using equality types in Haskell alone. Our implementation of Ωmega also supports several other features, such as extensible kinds and staging, which we shall not discuss in this paper. This integration creates a powerful meta-programming tool.

The third contribution is a demonstration that semantic properties of meta-programs (i.e., preserving object-language types) can be encoded in the type of the meta-program itself – the programmer need not resort to using another meta-logic to (formally) assure himself that his substitution algorithm preserves typing. We demonstrate this by implementing a type-preserving substitution operation on the object-language of simply typed λ-calculus.

* Supported by NSF CCR-0098126.
† Supported by NSF CCR-0098126.

LFM 2004 Cork, Ireland, July, 2004
.

The last contribution is the demonstration that these techniques support the embedding of logical frameworks style judgments into a programming language such as Haskell. This is important because it moves logical style reasoning about programs from the meta-logical level into the programming language.

2. ΩMEGA: A META-LANGUAGE WITH SUPPORT FOR TYPE EQUALITY

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 $\forall\varphi$. (φ `a`) → (φ `b`) was proposed by both Baars & Swierstra [2], and Cheney & Hinze [6] at about the same time, and is described somewhat earlier in a different setting by Weirich [20]. We illustrate this by the data-type `Equal : * → * → *`

```
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 [12]) 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 → a → b` and `b2a::Equal a b → b → 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 [2], and we shall not belabor their explanation any further. The essential characteristic of programming with type equality in Haskell is the requirement that programmers manipulate proofs of equalities between types using equality combinators. This has two practical drawbacks. First, manipulation of proofs using combinators 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 [6], and is one of the ideas behind Ωmega [17]. 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), 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`). Instead, it is built into the type system, and is used implicitly by the type-checker. Consider the following (fragmentary) data-type definitions. (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 →, and so on.)

```
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 to keep track of equalities between types is very similar to the constraints that the Haskell type checker uses to resolve class-based overloading. A special qualified type [9] is used to assert equality between types, and a constraint solving system is used to simplify and discharge these assertions. When assigning a type to a type constructor, the equations specified in the where clause just 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 the Haskell type (for example, `Eq a => a → a → Bool`).

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 data-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`

One practical thing to note is that in this context, the data-constructors of `Exp` and `Var` are given the following types:

```
Lit :: ∀e t.t=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)
```

It is important to note that the above qualified types can be *instantiated* to the following types:

```
Lit :: 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` from the 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 easily using congruence.

3. Ωmega EXAMPLE: SUBSTITUTION

Now, we shall develop our main example, showcasing the meta-programming facilities of Ωmega. First, we shall define a sample object-language of simply typed λ-calculus judgments, and then implement a type-preserving substitution function on those terms. While this object-language is quite simple, useful perhaps only for didactic purposes, we have applied our techniques on a wider range of meta-programs and object-languages (e.g., tagless staged interpreters for typed imperative languages, object-languages with modal type systems, and so on [13, 14]).

This example demonstrates type-preserving *syntax-to-syntax* transformations between object-language programs. Substitution, which we shall develop in the remainder of this paper, is one such transformation. Furthermore, a correct implementation of substitution can be used to build more syntax-to-syntax transformations: we shall provide an implementation of big-step semantics that uses substitution.

The substitution operation we present preserves object-language typing. As a meta-program, it not only analyzes object-language typing judgments, but also builds new judgments based on the result of that analysis.

Expressions and types

$$
\begin{array}{lll}
\tau & \in \mathbb{T} ::= & \mathsf{b} \mid \tau_1 \rightarrow \tau_2 \\
\Gamma & \in \mathbb{G} ::= & \langle\rangle \mid \Gamma, \tau \\
e & \in \mathbb{E} ::= & \mathsf{Var}\, n \mid \lambda_\tau e \mid e_1\, e_2
\end{array}
$$

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

$$
\frac{\Gamma, \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda_{\tau_1}.e : \tau_1 \rightarrow \tau_2}(\text{Abs}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}(\text{App})
$$

Figure 1: The simply typed λ-calculus fragment.

Substitutions à la $\lambda\upsilon$ [4]

$$
\sigma \quad \in \mathbb{S} ::= \quad e/ \mid \Uparrow(\sigma) \mid \uparrow
$$

$$
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e/ : \Gamma, \tau}(\text{Slash}) \quad \frac{}{\Gamma, \tau \vdash \uparrow : \Gamma}(\text{Shift}) \quad \frac{\Gamma \vdash \sigma : \Gamma'}{\Gamma, \tau \vdash \Uparrow(\sigma) : \Gamma', \tau}(\text{Lift})
$$

Figure 2: Explicit substitutions fragment.

3.1 The Simply Typed λ-calculus with Typed Substitutions

Figures 1 and 2 define two sets of typed expressions. The first figure of expressions (Figure 1) is just the simply typed λ-calculus. The second figure (Figure 2) defines a set of typed substitutions. The substitution expressions are taken from the $\lambda\upsilon$-calculus [4]. There are several of other ways to represent substitutions explicitly as terms (see Kristoffer Rose's excellent paper [16] for a comprehensive survey), but we have chosen the notation of $\lambda\upsilon$ for its simplicity.

A substitution expression σ is intended to represent a mapping from de-Bruijn indices to expressions (i.e., a substitution), the same way that λ-expressions are intended to represent functions. As in $\lambda\upsilon$, we define three kinds of substitutions in Figure 2 (see Figure 3 for a graphical illustration):

1. *Slash* ($e/$). Intuitively, the slash substitution maps the variable with the index 0 to e, and any variable with the index $n+1$ to $\mathsf{Var}\, n$.

2. *Shift* ($\uparrow$). The shift substitution adjusts all the variable indices in a term by incrementing them by one. It maps each variable n to the term $\mathsf{Var}\,(n+1)$.

3. *Lift* ($\Uparrow(\sigma)$). The lift substitution ($\Uparrow(\sigma)$) is used to mark the fact that the substitution σ is being applied to a term in a context in which index 0 is bound and should not be changed. Thus, it maps the variable with the index 0 to $\mathsf{Var}\, 0$. For any other variable index $n+1$, it maps it to the term that σ maps to n, with the provision that the resulting term must be adjusted with a shift: $((n+1) \mapsto \uparrow(\sigma(n)))$.

Typing substitutions. The substitution expressions are typed. The typing judgments of substitutions, written $\Gamma_1 \vdash \sigma : \Gamma_2$, indicate that the type of a substitution, in a given type assignment, is another type assignment. The intuition behind

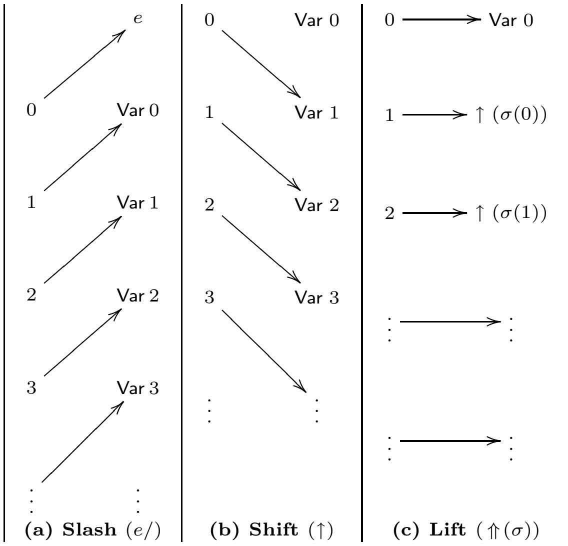

Figure 3: Substitutions

the substitution typing judgment is the following: given a term whose variables are assigned types by Γ_2, applying a the substitution σ yields an expression whose variables are assigned types by Γ_1.

Example. We describe a couple of example substitutions.

1. Consider the substitution ($\mathsf{True}/$). This substitution maps the variable with the index 0 to the Boolean constant `True`. The type of this substitution is $\Gamma \vdash \mathsf{True}/ : \Gamma, \mathsf{Bool}$. In other words, given any type assignment, the substitution ($\mathsf{True}/$) can be applied in any context where the variable 0 is assigned type Bool.

2. Consider the substitution $\sigma = (\Uparrow(\mathsf{True}/))$. σ is the substitution that replaces the variable with the index 1 with the constant True.

   Recall that the type of any substitution θ under a type assignment Γ, is a type assignment Δ (written $\Gamma \vdash \theta : \Delta$), such that for any expression e' to which the substitution θ is applied, the following must hold $\Delta \vdash e' : \tau$ and $\Gamma \vdash \theta(e') : \tau$.

   So, what type should we assign to σ? When applied to an expression, a lift substitution ($\sigma = \Uparrow(\mathsf{True}/)$) does not change the variable with the index 0. Thus, when typing σ as $\Gamma \vdash \sigma : \Delta$, we know something about the shape of Γ and Δ. Namely, for some Δ', we know that $\Delta = (\Delta', \tau)$, and for some Γ', we know that $\Gamma = (\Gamma', \tau)$. The type assignments Δ' and Γ' are determined by the sub-substitution $\mathsf{True}/$, yielding the following typing derivation:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathsf{True} : \mathsf{Bool}}\text{Const}}{\Gamma \vdash \mathsf{Bool}/ : \Gamma, \mathsf{Bool}}\text{Slash}}{\Gamma, \tau \vdash \Uparrow(\mathsf{Bool}/) : \Gamma, \mathsf{Bool}, \tau}\text{Lift}$$

There are three typing rules for the substitutions (Figure 2):

1. *Slash* ($e/$). A slash substitution $e/$ replaces the 0-index variable in an expression by e. Thus, in any context Γ, where e can be given type τ, the typing rule requires the substitution to work only on expressions in the type assignment Γ, τ, where the 0-index variable is assigned the type τ. Since the slash substitution also decrements the indexes of the remaning variables, they are all shifted to the right by one place, so that the remaning free variables can be assigned their old types in Γ after the substitution is applied.

$$\dfrac{\Gamma \vdash e : \tau}{\Gamma \vdash e/ : \Gamma, \tau}\text{(Slash)}$$

2. *Shift* ($\uparrow$). The shift substitution maps all variables n to $\mathsf{Var}\ (n+1)$. Thus, given a term whose variables are assigned type a by Γ, after performing the shift substitution, the types in the type assignment must for each variable must "move" to the left by one position. This is done by appending an arbitrary type τ for the variable with the index 0, which cannot occur free in the term after the substitution is performed.

$$\dfrac{}{\Gamma, \tau \vdash \uparrow : \Gamma}\text{(Shift)}$$

3. *Lift* ($\Uparrow(\sigma)$). For any variable index $(n+1)$ in a term, the substitution $\Uparrow(\sigma)$ applies σ to n and then shifts the resulting term. Thus, the 0-index term in the type assignment remains untouched, and the rest of the type assignment is as specified by σ:

$$\dfrac{\Gamma \vdash \sigma : \Gamma'}{\Gamma, \tau \vdash \Uparrow(\sigma) : \Gamma', \tau}\text{(Lift)}$$

Applying substitutions. In the remainder of this Section, we show how to implement a function (we call it subst) that takes a substitution expression σ, a λ-expression e, and returns an expression such that all the indices in e have been replaced according the substitution. In the simply typed λ-calculus, substitution preserves typing, so we expect the following property to be true of the substitution function subst: if $\Gamma \vdash \sigma : \Delta$ and $\Delta \vdash e : \tau$, then $\Gamma \vdash \mathsf{subst}\ \sigma\ e : \tau$.

How should subst work? Figure 4 presents two judgments, $(\sigma, e_1) \Rightarrow e_2$ and $(\sigma, n) \Rightarrow e$, which describe the action of substitutions on expressions and variables, respectively. These judgments are derived from the reduction relations of the $\lambda\upsilon$-calculus [4]. It is not difficult to show that this reduction strategy indeed does implement capture avoiding substitution sufficient to perform β reductions (see Benaissa, Lescanne & al. [4] for proofs).

4. IMPLEMENTING SUBSTITUTION IN ΩMEGA

Next, we show how to implement this substitution operation in Ωmega, using expression and substitution judgments instead of expressions and substitution expressions.

4.1 Judgments

The expression and substitution judgments can be easily encoded in Ωmega. The data-types `Var` and `Exp` encode expression and variable judgments presented in Figure 1.

Substitution on expressions

$$(\cdot,\cdot) \Rightarrow \cdot \subset \mathbb{S} \times \mathbb{E} \times \mathbb{E}$$

$$\frac{\begin{array}{c}(\sigma, e_1) \Rightarrow e_1' \\ (\sigma, e_2) \Rightarrow e_2'\end{array}}{(\sigma, (e_1\ e_2)) \Rightarrow e_1'\ e_2'} \qquad \frac{(\Uparrow(\sigma), e) \Rightarrow e'}{(\sigma, \lambda.e) \Rightarrow \lambda e'} \qquad \frac{(\sigma, n) \Rightarrow e}{(\sigma, \mathsf{Var}\ n) \Rightarrow e}$$

Substitution on variables

$$(\cdot,\cdot) \Rightarrow \cdot \subset \mathbb{S} \times \mathbb{N} \times \mathbb{E}$$

$$\frac{}{(e/, 0) \Rightarrow e} \qquad \frac{}{(e/, n+1) \Rightarrow \mathsf{Var}\ n} \qquad \frac{}{(\Uparrow(\sigma), 0) \Rightarrow \mathsf{Var}\ 0}$$

$$\frac{(\sigma, n) \Rightarrow e \quad (\uparrow, e) \Rightarrow e'}{(\Uparrow(\sigma), n+1) \Rightarrow e'} \qquad \frac{}{(\uparrow, n) \Rightarrow \mathsf{Var}\ (n+1)}$$

Figure 4: Applying substitutions to terms

```
data Var e t = ∀d.    Z where e = (d,t)
             | ∀d t2. S (Var d t) where e = (d,t2)

data Exp e t =          V   (Var e t)
             | ∀t1 t2. Abs (Exp (e,t1) t2)
                        where t = t1 → t2
             | ∀t1.    App (Exp e (t1 → t))
                            (Exp e t1)
```

The judgment `Var` implements the lookup and weakening rules for variables. Just as in the judgment of Figure 1, there are two cases:

1. First, there is the constructor `Z`. This constructor translates the definition of Figure 1 directly: the `where`-clause requires the type system of Ωmega to prove that there exists some environment γ such that the environment `t` is equal to γ extended by `t`.

2. The second constructor, `S` takes a judgment of type (`Var` γ `t`), and a requirement that the environment `e` is equal to the pair (γ, α), where both γ and α are existentially quantified.

The names `S` and `Z` are chosen to show how the judgments for variable are structurally the same as the natural number indices. Finally, the sub-judgments for the variable case are "plugged" into the definition of `Exp e t` using the constructor `V`.

The type of expression judgments (`Exp e t`) is constructed in a similar fashion. We shall only explain the abstraction case in some detail. The constructor `Abs` takes as its argument a judgment of type (`Exp (e,t1) t2`): an expression judgment of type `t2` in the type assignment `e`, extended so that it assigns the variable 0 the type `t1`. If this argument can be supplied, then the result type of the `Abs` judgment is the function type (`t1 → t2`), as indicated by the `where`-clause.

Next, we define a data-constructor `Subst gamma delta` that represents the typing judgments for substitutions . The type constructor `Subst gamma delta` represents the typing judgment $\Gamma \vdash \sigma : \Delta$ presented in Figure 2.

```
data Subst gamma delta =
  ∀t1.    Shift
```

```
1 subst :: Subst gamma delta →
2          Exp delta t → Exp gamma t
3 subst s (App e1 e2)  = App (subst s e1) (subst s e2)
4 subst s (Abs e) = Abs (subst (Lift s) e)
5 subst (Slash e) (V Z) =  e
6 subst (Slash e) (V (S n)) = V n
7 subst (Lift s) (V Z) = V Z
8 subst (Lift s) (V (S n)) = subst Shift (subst s (V n))
9 subst (Shift) (V n) = V (S n)
```

Figure 5: Substitution in simply typed λ-calculus.

```
          where gamma = (delta,t1)
| ∀t1.    Slash (Exp gamma t1)
          where delta = (gamma,t1)
| ∀del1 gam1 t1. Lift (Subst gam1 del1)
          where delta = (del1,t1),
                gamma = (gam1,t1)
```

4.2 Substitution

Finally, we define the substitution function `subst`. It has the following type:

```
subst :: Subst gamma delta →
         Exp delta  t → Exp gamma t
```

It takes a substitution whose type is `delta` in some type assignment `gamma`, an expression of type `t` that is typed in the type assignment `delta`, and produces an expression of type `t` typable in the type assignment `gamma`.

We will discuss the implementation of the function `subst` (Figure 5) in more detail. In several relevant cases, we shall describe the process by which the Ωmega type-checker makes sure that the definitions are given correct types. Recall that every pattern-match over one of the `Exp` or `Subst` judgments may introduce zero or more equations between types, which are then available to the type-checker in the body of a case (or function definition). The type checker may use these equations to prove that two types are equal. In the text below, we sometimes use the type variables `gamma` and `delta` for notational convenience, but also Skolem constants like `_1`. These are an artifact of the Ωmega type-checker (they appear when pattern-matching against values that may contain existentially quantified variables) and should be regarded as type constants.

1. The application case (line 3) simply applies the substitution to the two sub-expression judgments and then rebuilds the application judgment from the results.

2. The abstraction case (line 4) pushes the substitution under the λ-abstraction. It may be interesting to examine the types of the various subexpressions in this definition.

`Abs e`	:	`Exp delta t`, where `t=t1 → t2`
`e`	:	`Exp (delta,t1) t2`
`s`	:	`Subst gamma delta`
`Lift s`	:	`Subst (gamma,t1) (delta,t1)`
`subst (Lift s) e`	:	`Exp (gamma,t1) t2`

The body of the abstraction, `e` has the type `(delta,t1)`, where `t1` is the type of the domain of the λ-abstraction.

In order to apply the substitution `s` to the body of the abstraction (`e`), we need a substitution of type `(Subst (gamma,t1) (delta,t1))`. This substitution can be obtained by applying `Lift` to `s`. Then, recursively applying `subst` with the lifted substitution to the body `e`, we obtain an expression of type `(Exp (gamma,t1) t2)`, from which we can construct a λ-abstraction of the `(Exp gamma (t1 → t2))`.

3. The variable-slash case (line 5-6). There are two cases when applying the slash substitution to a variable expression:

   (a) Variable 0. The substitution `(Slash e)` has the type `(Subst (gamma) (gamma,t))`, and contains the expression `e :: Exp gamma t`. The expression `(V Z)` has the type `(Exp (delta,t) t)`. Pattern matching introduces the equation `gamma=delta`, and we can use `e` to replace `(V Z)`.

   ```
   Slash e  ::  (Subst (gamma) (gamma,t))
   e        ::  Exp gamma t
   ```

   (b) Variable $n+1$. Pattern matching on the substitution argument introduces the equation `delta=(gamma,t1)`. Pattern matching against the expression `(V (S n))` introduces the equation `delta=(gamma',t)`, for some `gamma'`. The expression result expression `(V n)` has the type `(Exp gamma' t)`. The type checker then uses the two equalities to prove that it has the type `(Exp gamma t)`. It does this by first using congruence to prove that `gamma=gamma'`, and then by applying this equality to obtain `Exp gamma' t = Exp gamma t`.

   ```
   Slash e    ::  Subst gamma (gamma,t)
   (V (S n))  ::  Exp delta t
   ```

4. The variable-lift case (lines 7-8). There are two cases when applying the lift substitution to a variable expression.

   (a) Variable 0. This case is easy because the lift substitution places makes no changes to the variable with the index 0. We are able simply to return `(V Z)` as a result.

   (b) Variable $n+1$. The first pattern (`Lift s :: Subst gamma delta`), on the substitution, introduces the following equations:

   ```
   delta = (d',_1),
   gamma = (g',_1)
   ```

   The pattern on the variable (`V(S n):: Var delta t`) introduces the equation

   ```
   delta = (d2,_2)
   ```

   The first step is to apply the substitution `s` of type `(Subst g' d')` to a decremented variable index `(V n)` which has the type `n :: Var d2 t`. To do this, the type checker has to show that `g'=d2`, which easily follows from the equations introduced by the pattern, yielding a result of type `(Exp g' t)`. Applying the `Shift` substitution to this result yields an expression of type `(Exp (g',a) t)` (where `a` is can be any type). Now, equations above can be used to prove that this expression has the type `(Exp gamma t)` using the equation `gamma=(g',_1)`.

5. Variable-shift case (line 9). Pattern matching on the `Shift` substitution introduces the equation `gamma = (delta,_1)`. The expression has the type `(Exp delta t)`. Applying the successor to the variable results in an expression `(V (S n))` of type `(Exp (delta,a) t)`. Immediately, the type checker can use the equation introduced by the pattern to prove that this type is equal to `(Exp gamma t)`.

We have defined type-preserving substitution simply typed λ-calculus judgments. Recall, that since equality proofs can be encoded in Haskell, it should be possible (with certain caveats) to implement the function `subst` in Haskell (with a couple of GHC extensions). It is worth noting that Ωmega has proven very helpful in writing such complicated functions: explicitly manipulating equality proofs for such a function in Haskell, would result in code that is both verbose and difficult to understand.

5. A BIG-STEP EVALUATOR

Finally, we implement a simple evaluator based on the big-step semantics for the λ-calculus. The evaluation relation is given by the following judgment:

$$\frac{}{\lambda e \Rightarrow \lambda e} \quad \frac{}{x \Rightarrow x} \quad \frac{e_1 \Rightarrow \lambda e' \quad (e_2/, e') \Rightarrow e_3 \quad e_3 \Rightarrow e''}{e_1\ e_2 \Rightarrow e''}$$

Note that in the application case, we first use the substitution $(e_2/, e') \Rightarrow e_3$ to substitute the argument e_2 for the variable with index 0 into the body of the λ-abstraction.

The big-step evaluator is implemented as the function `eval` which takes a well-typed expression judgment of type `(Exp delta t)`, and returns judgments of the same type. The evaluator reduces β-redices using a call-by-name strategy, relying upon the substitution implemented above.

```
eval :: Exp delta t -> Exp delta t
eval (App e1 e2) =
  case eval e1 of
    Abs body -> eval (subst (Slash e2) body)
eval x = x
```

Note that the type of the function `eval` statically ensures that it preserves the typing of the object language expressions it evaluates, with the usual caveats that the `Exp`s faithfully encode well-typed λ-expressions.

Finally, let us apply the big-step evaluator to a simple example. Consider the expression, `example`.

```
example :: Exp gamma (a → a)
example = (Abs (V Z)) ‘App‘ ((Abs (Abs (V Z)))
          ‘App‘ (Abs (V Z)))
-- example = (λ x.x) ((λ y. (λ z.z))) (λ x.x)
```

The expression `example` evaluates to the identity function. Applying `eval` to it yields precisely that result:

```
evExample = eval example
-- evExample = (Abs (V Z)) : Exp gamma (a → a)
```

6. RELATED WORK

Implementations of simple interpreters that use equality proof objects implemented as Haskell datatypes, have been given by Weirich [20] and Baars and Swierstra [2]. Baars and Swierstra use an untyped syntax, but use equality proofs to encode dynamically typed values. Hinze and Cheney [5, 6] have recently resurrected the notion of "phantom type," first introduced by Leijen and Meijer [10]. 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* [21], 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 (extensible kinds and rank-N 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.

Simonet and Pottier [18] proposed a system of *guarded algebraic data types,* which seem equivalent in expressiveness to *phantom types*, *guarded recursive datatype constructors*, and Ωmega's equality qualified (data)types. They present a type system for guarded algebraic data types as an extension to the HM(X) [19] type system, and describe a type inference algorithm. They prove a number of important properties about the type system and the inference algorithm (e.g., type soundness, correctness, and so on).

The technique of manipulating well-typedness judgments has been used extensively in various logical frameworks [7, 15]. We see the advantage of our work here in translating this methodology into a more main-stream functional programming idiom. Although our examples are given in Ωmega, most of our techniques can be adapted to Haskell with some fairly common extensions.

In previous work, we have used the techniques and programming language extensions described above to address the problem of tagless interpreters in meta-programming [14]. Tagless interpreters can easily be constructed in dependently typed languages such as Coq [3] and Cayenne [1]. These languages, however, do not support staging, nor have they gained a wide audience in the functional programming community. Programming with well-typed object-language syntax, applied to the problem of constructing tagless staged interpreters, has been shown possible in a meta-language (provisionally called MetaD) with staging and dependent types [14]. The drawback of this approach is that there is no "industrial strength" implementation for such a language. In fact, the judgment encoding technique presented in this paper is basically the same, except that instead of using a dependently typed language, we encode the necessary machinery in a language which is arguably more recognizable to Haskell programmers. By using explicit equality types, everything can be encoded using the standard GHC extensions to Haskell 98. Ωmega adds further ease of use to these techniques, relieving the programmer of the responsibility of explicitly manipulating equality proofs.

A technique using *indexed type systems* [22], a restricted and disciplined form of dependent typing, has been used to write interpreters and source-to-source transformations on typed terms [21]. The meta-language with guarded recursive datatype constructors, used by Xi & al., seems to be roughly equivalent in expressive power to Ωmega. Ωmega, however, is equipped with additional features, such as staging, which may give it a wider range of useful applications.

7. DISCUSSION AND FUTURE WORK

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 [8].

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 required.

Polymorphism and Binding Constructs in Types. The object-language of the example presented in this paper (Figure 1), 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 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.

8. REFERENCES

[1] Lennart Augustsson and Magnus Carlsson. 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`.

[2] Arthur I. Baars and S. Doaitse Swierstra. 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, October 2002.

[3] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997.

[4] Zine-El-Abidine Benaissa, Daniel Briaud, Pierre Lescanne, and Jocelyne Rouyer-Degli. $\lambda\nu$, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 6(5):699–722, September 1996.

[5] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Proc. of the workshop on Haskell*, pages 90–104. ACM_Press, 2002.

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

[7] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings Symposium on Logic in Computer Science*, pages 194–204, Washington, June 1987. IEEE Computer Society Press. The conference was held at Cornell University, Ithaca, New York.

[8] M. P. Jones. The hugs 98 user manual, 200.

[9] Mark P. Jones. *Qualified types :–theory and practice*. PhD thesis, Keble College, Oxford University, 1992.

[10] Daan Leijen and Erik Meijer. Domain-specific embedded compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 109–122, Berkeley, CA, October 3–5 1999. USENIX Association.

[11] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, April 1980.

[12] Bengt Nordström, Kent Peterson, and Jan M. Smith. *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 authors homepage.

[13] Emir Pašalić. *Heterogeneous Meta-programming*. PhD thesis, Oregon Health and Sciences University, OGI School of Science & Engineering, 2004. Forthcoming.

[14] Emir Pašalić, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *The International Conference on Functional Programming (ICFP '02)*, Pittsburgh, USA, October 2002. ACM.

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

[16] Kristoffer H. Rose. Explicit substitution – tutorial & survey. Technical Report LS-96-3, BRICS, Universitty of Århus, october 1996. BRICS Lecture Series.

[17] Tim Sheard, Emir Pasalic, and R. Nathan Linger. The ωmega implementation. Available on request from the author., 2003.

[18] Vincent Simonet and François Pottier. Constraint-based type inference for guarded algebraic data types. Submitted for publication, July 2003.

[19] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with constrained types. In *FOOL4: 4th. Int. Workshop on Foundations of Object-oriented programming Languages*, January 1997.

[20] Stephanie Weirich. 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*, pages 58–67, N.Y., September 18–21 2000. ACM Press.

[21] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL-03)*, volume 38, 1 of *ACM SIGPLAN Notices*, pages 224–235, New York, January 15–17 2003. ACM Press.

[22] Howgwei Xi and Frank Pfenning. Dependent types in practical programming. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 214–227, New York, NY, January 1999. ACM.

9. ACKNOWLEDGMENT

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