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 preserves 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.
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. \textit{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 (\texttt{Equal a b}). Thus a non-bottom value \( p :: \texttt{Equal a b} \), 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 [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 \texttt{Equal}:

\begin{verbatim}
data Equal a b = Equal (\forall \varphi. \ (\varphi \ a) \rightarrow (\varphi \ b))
cast :: Equal a b \rightarrow (\forall \varphi. \ (\varphi \ a) \rightarrow (\varphi \ b))
cast (Equal f) = f
\end{verbatim}

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 \( \varphi \). Proofs are useful, since from a proof \( p :: \texttt{Equal a b} \), we can extract functions that cast values of type \( (\forall \varphi. \ (\varphi \ a) \rightarrow (\varphi \ b)) \) to type contexts \( (\forall \varphi) \). For example, we can construct functions \( a2b :: \texttt{Equal a b} \rightarrow a \rightarrow b \) and \( b2a :: \texttt{Equal a b} \rightarrow b \rightarrow a \), 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 [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 \textit{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 \textit{ΩMEGA} [17]. In the remainder of this paper, we shall use \textit{ΩMEGA}, rather than pure Haskell to write our examples. We conjecture that, in principle, whatever it is possible to do in \textit{ΩMEGA}, it is also possible to do in Haskell (plus the usual set of extensions), only in \textit{ΩMEGA} it is expressed more cleanly and succinctly.

The syntax and type-system of \textit{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 \textit{ΩMEGA} and Haskell.

In \textit{ΩMEGA}, the equality between types is not encoded explicitly (as the type constructor \texttt{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 \texttt{forall} with the symbol \( \forall \). We shall write explicitly universally or existentially quantified variables with Greek letters. Arrow types \( (\rightarrow) \) will be written as \( \Rightarrow \), and so on.)

\begin{verbatim}
data Exp e t = Lit Int where t=Int | V (Var e t) data Var e t = forall \gamma. Z where e = (\gamma, t) | \forall \gamma. A (Var \gamma t) where e = (\gamma, a)
\end{verbatim}

Each data-constructor in \textit{ΩMEGA} may contain a \texttt{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 \texttt{Eq a => a \rightarrow Bool} in Section 2, with one important difference. The user is not required to provide any actual evidence of type equality – the \textit{ΩMEGA} type checker keeps track of equalities between types and proves and propagates them automatically.

The mechanism \textit{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 \texttt{Lit} is given the type \( \forall e. t. (t=\texttt{Int}) \Rightarrow \texttt{Exp e t} \). The equation \( t=\texttt{Int} \) is just another form of predicate, similar to the class membership predicate in the Haskell type (for example, \( \texttt{Eq a} \Rightarrow a \rightarrow \texttt{Bool} \)).

\textbf{Tracking equality constraints.} When type-checking an expression, the \textit{ΩMEGA} type checker keeps two sets of equality constraints: \textit{obligations} and \textit{assumptions}.

\textbf{Obligations.} The first set of constraints is a set of \textit{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 (\texttt{Lit 5}). The constructor \texttt{Lit} is assigned the type \( \forall e. t. (t=\texttt{Int}) \Rightarrow \texttt{Exp e t} \). Since \texttt{Lit} is polymorphic in \( e \) and \( t \), the type variable \( t \) can be instantiated to \texttt{Int}. Instantiating \( t \) to \texttt{Int} also makes the equality constraint obligation \( \texttt{Int=Int} \), which can be trivially discharged by the type checker.

\texttt{Lit 5 :: Exp e Int with obligation Int = Int}

One practical thing to note is that in this context, the data-constructors of \texttt{Exp} and \texttt{Var} are given the following types:...
Lit :: Ve t. t=Int ⇒ Exp e t
Z :: Ve e' t. e=(e', t) ⇒ Var e t
S :: Ve 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) 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 env

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

| τ ∈ T ::= | b | τ1 → τ2 |
| Γ ∈ G ::= | ⊢ | Γ, τ |
| e ∈ E ::= | Var n | λ, e | e1 e2 |

Figure 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 λv-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 λv 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 λv, 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 Var n.
2. Shift (↑). The shift substitution adjusts all the variable indices in a term by incrementing them by one. It maps each variable n to the term Var (n + 1).
3. Lift (↑ (σ)). The lift substitution (↑ (σ)) 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 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) → ↑ (σ(n))).

Typing substitutions. The substitution expressions are typed. The typing judgments of substitutions, written Γ1 ⊢ σ : Γ2, indicate that the type of a substitution, in a given type assignment, is another type assignment. The intuition behind
the substitution typing judgment is the following: given a term whose variables are assigned types by \( \Gamma_2 \), applying a substitution \( \sigma \) yields an expression whose variables are assigned types by \( \Gamma_1 \).

**Example.** We describe a couple of example substitutions.

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

2. Consider the substitution \( \sigma = (\uparrow (\text{True}/)) \). \( \sigma \) is the substitution that replaces the variable with the index 1 with the constant \( \text{True} \).

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

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

\[
\begin{align*}
\Gamma & : \text{True} : \text{Bool} \quad \text{Const} \\
\Gamma & : \text{Bool} / : \Gamma, \text{Bool} \quad \text{Slash} \\
\Gamma, \tau & \vdash \uparrow(\text{Bool}/) : \Gamma, \text{Bool}, \tau \quad \text{Lift}
\end{align*}
\]

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 \( \Gamma \), where \( e \) can be given type \( \tau \), the typing rule requires the substitution to work only on expressions in the type assignment \( \Gamma, \tau \), where the 0-index variable is assigned the type \( \tau \). Since the slash substitution also decrements the indexes of the remaining variables, they are all shifted to the right by one place, so that the remaining free variables can be assigned their old types in \( \Gamma \) after the substitution is applied.

\[
\frac{}{\Gamma, \tau \vdash e : \tau} \quad \text{(Slash)}
\]

2. **Shift** \( (\uparrow(\sigma)) \). The shift substitution maps all variables \( n \) to \( \text{Var} \) \((n + 1) \). Thus, given a term whose variables are assigned type \( a \) by \( \Gamma \), 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 \( \tau \) for the variable with the index 0, which cannot occur free in the term after the substitution is performed.

\[
\frac{}{\Gamma, \tau \vdash \uparrow(\sigma) : \Gamma', \tau} \quad \text{(Shift)}
\]

3. **Lift** \( (\uparrow(\sigma)) \). For any variable index \((n + 1)\) in a term, the substitution \( \uparrow(\sigma) \) applies \( \sigma \) 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 \( \sigma \):

\[
\frac{}{\Gamma \vdash \sigma : \Gamma' \quad \Gamma, \tau \vdash \uparrow(\sigma) : \Gamma', \tau} \quad \text{(Lift)}
\]

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

How should \( \text{subst} \) 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 \( \lambda \nu \)-calculus [4]. It is not difficult to show that this reduction strategy indeed does implement capture avoiding substitution sufficient to perform \( \beta \) reductions (see Benaissa, Lescanne & al. [4] for proofs).

4. IMPLEMENTING SUBSTITUTION IN \( \Omega \text{MEGA} \)

Next, we show how to implement this substitution operation in \( \Omega \text{MEGA} \), using expression and substitution judgments instead of expressions and substitution expressions.

4.1 Judgments

The expression and substitution judgments can be easily encoded in \( \Omega \text{MEGA} \). The data-types \( \text{Var} \) and \( \text{Exp} \) encode expression and variable judgments presented in Figure 1.
Substitution on expressions
\[
(\cdot, \cdot) \Rightarrow \cdot \subseteq S \times E \times E
\]

\[
(\sigma, (e_1, e_2)) \Rightarrow e_1' e_2' \\
(\sigma, e_1) \Rightarrow e_1' \\
(\sigma, e) \Rightarrow e'
\]

Substitution on variables
\[
(\cdot, \cdot) \Rightarrow \cdot \subseteq S \times N \times E
\]

\[
\frac{e/0}{\text{Var} n} \\
\frac{(\sigma, n) \Rightarrow e \quad (\cdot, e) \Rightarrow e'}{(\sigma, n + 1) \Rightarrow e'} \\
\frac{\langle \sigma, 0 \rangle \Rightarrow \text{Var} 0}{\langle \sigma, n \rangle \Rightarrow \text{Var} (n + 1)}
\]

Figure 4: Applying substitutions to terms

data Var e t = \forall d. \text{Z} \text{ where } e = (d,t) \\
data Exp e t = \forall t1. t2. S (\text{Var d t}) \text{ where } e = (d, t2)

The judgment \text{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 \text{Z}. This constructor translates the definition of Figure 1 directly: the \text{where}-clause requires the type system of \text{Omega} to prove that there exists some environment \gamma such that the environment \gamma is equal to \gamma \alpha extended by \gamma t.

2. The second constructor, \text{S} takes a judgment of type (\text{Var} \gamma t), and a requirement that the environment \gamma is equal to the pair (\gamma, \alpha), where both \gamma and \alpha are existentially quantified.

The names \text{S} and \text{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 \text{Exp} e t using the constructor \text{V}.

The type of expression judgments (\text{Exp} e t) is constructed in a similar fashion. We shall only explain the abstraction case in some detail. The constructor \text{Abs} takes as its argument a judgment of type (\text{Exp} e (t1 t2)) as 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 \text{Abs} judgment is the function type (t1 \rightarrow t2), as indicated by the \text{where}-clause.

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

data Subst gamma delta = \\
\forall t1. \text{Shift}

4.2 Substitution

Finally, we define the substitution function \text{subst}. It has the following type:

\text{subst} :: \text{Subst gamma delta} \rightarrow \\
\text{Exp delta t} \rightarrow \text{Exp gamma t}

It takes a substitution whose type is \text{delta} in some type assignment \text{gamma}, an expression of type t that is typed in the type assignment \text{delta}, and produces an expression of type t typed in the type assignment \text{gamma}.

We will discuss the implementation of the function \text{subst} (Figure 5) in more detail. In several relevant cases, we shall describe the process by which the \text{Omega} type-checker makes sure that the definitions are given correct types. Recall that every pattern-match over one of the \text{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 \text{gamma} and \text{delta} for notational convenience, but also Skolem constants like \_1. These are an artifact of the \text{Omega} 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 \lambda-abstraction. It may be interesting to examine the types of the various subexpressions in this definition.

| Abs e | Exp \text{delta t}, where \text{t=t1→t2} |
| e | \text{Exp (delta, t1) t2} |
| s | \text{Subst gamma delta} |
| Lift s | \text{Subst (gamma, t1) (delta, t1)} |
| subst (Lift s) e | \text{Exp (gamma, t1) t2} |

The body of the abstraction, e has the type (\text{delta, t1}), where t1 is the type of the domain of the \lambda-abstraction.
In order to apply the substitution \( s \) to the body of the abstraction \((e)\), we need a substitution of type \((\text{Subst} (\gamma,t_1)) (\delta,t_1))\). This substitution can be obtained by applying \(\text{Lift} \) to \(s\). Then, recursively applying \(\text{subst} \) with the lifted substitution to the body \(e\), we obtain an expression of type \((\text{Exp} (\gamma,t_1)) \) from which we can construct a \(\lambda\)-abstraction of the \((\text{Exp} \gamma (t_1 \rightarrow t_2))\).

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

\[
\text{Slash} e :: (\text{Subst} (\gamma) (\gamma,t))
\]

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

\[
\text{Slash} e :: \text{Subst} (\gamma) (\gamma,t) \\
(V (S n)) :: \text{Exp} \delta t
\]

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 \((\text{Lift} s :: \text{Subst} \gamma \delta)\), on the substitution, introduces the following equations:

\[
delta = (d',_1), \\
\gamma = (g',_1)
\]

The pattern on the variable \((V (S n)):: \text{Var} \delta)\) introduces the equation

\[
delta = (d_2,2)
\]

The first step is to apply the substitution \(s\) of type \((\text{Subst} g' d')\) to a decremented variable index \((V n)\) which has the type \(n:: \text{Var} d_2\). To do this, the type checker has to show that \(g'=d_2\), which easily follows from the equations introduced by the pattern, yielding a result of type \((\text{Exp} g' t)\). Applying the \(\text{Shift} \) substitution to this result yields an expression of type \((\text{Exp} (g',a) t)\) (where \(a\) is can be any type).

Now, equations above can be used to prove that this expression has the type \((\text{Exp} \gamma t)\) using the equation \(g'=\gamma\).

5. Variable-shift case (line 9). Pattern matching on the \(t2\). Shift substitution introduces the equation \(\delta=(\delta',1)\). The expression has the type \((\text{Exp} \delta t)\). Applying the successor to the variable results in an expression \((V (S n))\) of type \((\text{Exp} (\delta,a) t)\). Immediately, the type checker can use the equation introduced by the pattern to prove that this type is equal to \((\text{Exp} \gamma t)\).

We have defined type-preserving substitution simply typed \(\lambda\)-calculus judgments. Recall, that since equality proofs can be encoded in Haskell, it should be possible (with certain caveats) to implement the function \(\text{subst} \) in Haskell (with a couple of GHC extensions). It is worth noting that \(\Omega\) 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 \(\lambda\)-calculus. The evaluation relation is given by the following judgment:

\[
e_1 \Rightarrow \lambda e' \Rightarrow (e_2, e') \Rightarrow e_3 \Rightarrow e'' \\
\lambda c \Rightarrow \lambda x \Rightarrow x \Rightarrow x \\
e_1 e_2 \Rightarrow e''
\]

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 \(\lambda\)-abstraction.

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

\[
\text{eval} :: \text{Exp} \gamma t \rightarrow \text{Exp} \gamma t \\
\text{eval} \text{ (App e1 e2)} = \\
\text{case eval e1 of} \\
\text{Abs body \rightarrow eval (subst (Slash e2 body)} \\
\text{eval x = x}
\]

Note that the type of the function \(\text{eval} \) statically ensures that it preserves the typing of the object language expressions it evaluates, with the usual caveats that the \(\text{Exp}s\) faithfully encode well-typed \(\lambda\)-expressions.

Finally, let us apply the big-step evaluator to a simple example. Consider the expression, \(\text{example} \).

\[
\text{example} :: \text{Exp} \gamma (a \rightarrow a) \\
\text{example} = (\text{Abs} (V Z)) \ '\text{App} '((\text{Abs} (\text{Abs} (V Z)))) \\
\ '\text{App} '((\text{Abs} (V Z))) \\
\text{-- example} = (\lambda x. x) ((\lambda y. (\lambda z.z)) (\lambda x. x))
\]

The expression \(\text{example} \) evaluates to the identity function. Applying \(\text{eval} \) to it yields precisely that result:

\[
\text{evExample} = \text{eval example} \\
\text{-- evExample} = (\text{Abs} (V Z)) : \text{Exp} \gamma (a \rightarrow a)
\]
6. RELATED WORK

Integration 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
code 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 Ch-
eneys’s phantom types are designed to address some of the
problems that arise when using equality proofs to represent
pre-indexed data. Their main motivation is to provide
a language in which polytypic programs, as such 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
types 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 seman-
tics 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 Omega’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 translat-
ing this methodology into a more main-stream functional
programming idiom. Although our examples are given in
Omega, most of our techniques can be adapted to Haskell
with some fairly common extensions.

In previous work, we have used the techniques and pro-
gramming 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 com-
munity. Programming with well-typed object-language syn-
tax, 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 ma-
chinery in a language which is arguably more recognizable
to Haskell programmers. By using explicit equality types,
everything can be encoded using the standard GHC exten-
sions to Haskell 98. Omega 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 Omega. Omega, 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 Omega, a language designed to
be as similar to Haskell. We have implemented our own
Omega 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 (Fig-
ure 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 exis-
tential types, we will have to find a way of dealing with type
constructors or type functions as index arguments to judg-
ments, which is difficult to do in Haskell or Omega. We are
currently working on extending the Omega 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 Omega. The examples presented in
this paper succeed because we manage to encode the usual
logical-framework-style inductive predicates into the type
system of Omega. We have acquired considerable experi-
ence 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 Omega 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
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).


λν, a calculus of explicit substitutions which preserves strong
normalisation. Journal of Functional Programming,


Society Press. The conference was held at Cornell
University, Ithaca, New York.


of International Series of Monographs on Computer Science. Oxford University Press, New York, NY,
1990. Currently available online from first authors homepage.


International Conference on Functional Programming (ICFP ’02), Pittsburgh, USA, October 2002. ACM.

deductive systems. In Harald Ganzinger, editor, Proceedings of the 16th International Conference on Automated Deduction (CADE-16), volume 1632 of


author., 2003.


[19] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with constrained types. In FOOL4:

volume 35.9 of ACM Sigplan Notices, pages 58–67,

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,

[22] Hongwei 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.