Self-Interpretation and Reflection in a Statically Typed Language

Extended Abstract

Konstantin Läufer
Loyola University of Chicago
laufer@math.luc.edu

Martin Odersky
Universität Karlsruhe
odersky@ira.uka.de

July 31, 1993

1 Introduction

Reflection is the ability of a system to perform a computation about itself. This ability typically includes a way of representing programs as data ("reification") and of executing representations of programs ("self-interpretation"). The interpreter is accessible to the interpreted program in the form of an "eval" function. Reflection is traditionally studied in untyped or dynamically typed languages such as LISP [4] [2], Smalltalk [3], or the λ-calculus [12].

By contrast, we consider self-interpretation and reflection in a statically typed metalanguage along the lines of [13] and [10]. Since the language is statically typed, the data structure used as a representation for programs is statically typed as well. Reflection in a statically typed context can be characterized as follows:

Type-preserving representation: A representation is well-typed iff the represented program is well-typed.

Type-preservation of self-interpretation: A representation of a program of some type is reduced to a representation of the result of the program of the same type.

Reuse: Features of the metalanguage, for instance the type checker or garbage collector, are reused directly without reprogramming them. (A lack of reuse is called redundancy.)

Static typing in reflective systems is useful since it precludes breaches of type safety through self-interpretation; we cannot construct a syntax tree that causes a run-time type error when we interpret it. This goal is captured by the principle that the interpreted program should be as safe as the original one.

2 The SK Combinator Calculus

We explore reflexivity in a typed version of the SK combinator calculus [14]. Unlike the equivalent, more familiar λ-calculus, the SK calculus provides a Turing-complete language without variables or scope rules. This eliminates the need to keep track of variables in an environment. SK combinator expressions are applicative expressions consisting of the combinators S, K, and Y, defined in Figure 1.

$S\ f\ g\ x$	$= f\ x\ (g\ x)$	$\vdash S :\ (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$
$K\ x\ y$	$= x$	$\vdash K :\ \alpha \to \beta \to \alpha$
$Y\ f$	$= f\ (Y\ f)$	$\vdash Y :\ (\alpha \to \alpha) \to \alpha$

Figure 1: SK combinators and their types

An SK expression said to be well-typed if a typing can be derived for it, using the type rules in Figure 1 and the following rule for function application:

$$\text{APP} \quad \frac{\vdash e : \tau' \rightarrow \tau \qquad \vdash e : \tau}{\vdash e \; e' : \tau}$$

3 The Metalanguage

The metalanguage we use to implement our metacircular interpreter is a version of the statically typed lazy functional language Haskell [6]. While Haskell already provides parametric polymorphism and systematic overloading, we extend the language to include existential quantification [11] and polymorphic recursion [7]. In this section, we briefly describe the essential features of our metalanguage. A comprehensive treatment of an extension of Haskell with existential quantification can be found in [8] [9], including examples demonstrating that the combination of type classes and existential quantification gives expressive power comparable to that of object-oriented programming languages.

Systematic overloading

Haskell provides *type classes* [15] for the systematic overloading of functions and operations. For example, to overload the multiplication operation, we first declare a class containing the operation.

```
class Mult a where
  (*) :: a -> a -> a
```

We then declare *instance types* of the type class for which we want to use the overloaded operation. We must implement the operation for each instance type.

```
instance Mult Int   where (*) = primIntMult
instance Mult Float where (*) = primIntMult
instance Mult Bool  where (*) = (&&)
```

We can now use the overloaded operation in the definition of new functions, for example:

```
square :: (Mult a) => a -> a
square x = x * x
```

Given this definition, `square True` evaluates to `True`, and `square 3` evaluates to `9`.

Lazy evaluation

In lazy or call-by-need evaluation, expressions are not evaluated until they are actually needed. This includes actual parameter expressions in a function call. Consequently, we can define a fixpoint combinator within the language.

```
fix   :: (a -> a) -> a
fix h = h (fix h)
```

The factorial function is the fixpoint of the following functional:

```
h     :: (Num a) => (a -> a) -> a -> a
h f x = if x == 0 then 1 else x * f (x - 1)
```

Thus the expression `fix h 4` evaluates to `24`.

Existential quantification

The declaration

```
data KEY = Key a (a -> Int)
```

introduces a new (parameterless) type `KEY` and a value constructor `Key` of type

```
Key :: a -> (a -> Int) -> KEY
```

We use pattern matching to extract the two components of a value of type **KEY**. The type variable **a** in the component type of the constructor **Key** is *existentially quantified*, and the component type is written $\exists\alpha.\,\alpha\times(\alpha\to int)$. Thus the argument type of **f** and the type of **x** in the following function definition are identical *private* types, and the function has the type **KEY -> Int**:

```
whatkey (Key x f) = f x
```

Such private types must not escape the scope in which they are introduced; thus the following function definition is ill-typed:

```
value (Key x f) = x
```

Assuming that **k = Key [1,2,3] length**, the expression **whatkey k** evaluates to **3**.

Polymorphic recursion

In most statically typed polymorphic languages, recursion must be *monomorphic* in the sense that all recursive applications of the (polymorphic) function being defined must have the same (monomorphic) typing. Some languages, including most Haskell implementations, allow *polymorphic recursion* if an explicit type declaration was given for the function. We can thus define the following length function for lists:

```
len        :: [a] -> Int
len []      = 0
len [x]     = 1
len (x:xs) = len [1] + len [True] - 1 + len xs
```

As expected, **len [1,4,9]** evaluates to **3**, as does **len [[],[1],[1,2]]**.

4 Representation and Self-Interpretation

We are now ready to present the data structures for our statically-typed metacircular interpreter, following the framework given in [12]. The interpreted program is a simple SK combinator expression, *represented* by its abstract syntax tree. Unlike in metacircular interpreters for untyped languages such as LISP, our syntax trees are typed. A syntax tree of type **Rep a**, defined in Figure 2, denotes an expression of type **a**, for every type **a**. As a corollary of this correspondence, we have:

> If a syntax tree is well-typed, then the expression it represents is well-typed.

A syntax tree of type **Rep a** represents either a combinator of type **a** or the application of an expression of type **b -> a** to an expression of type **b**. Application is expressed by the left-associative infix constructor **@** in the last variant of **Rep**. The type variable **b** in the component type of **@** is existentially quantified since we are only interested in the result type **a** of the application. In addition to the basic combinators **S**, **K**, and **Y**, we provide combinators to represent natural numbers. For metacircularity, we add the combinators **Eval**, **Reif**, and **Refl**, representing applications of the corresponding metafunctions defined in the next section.

A *self-interpreter* for the SK calculus is a function that reduces a representation of an expression e to a representation of an expression e' whenever e reduces to e' in the SK calculus itself. The self-interpreter **intp** for our representation of SK expressions is defined in Figure 2 along with its auxiliary functions. The function **coerce** is needed to convert a representation of a private type arising from existential quantification to an equivalent expression of a suitable type. Polymorphic recursion is used in the definitions of **coerce** and **intp**. Our self-interpreter enjoys the following type-preservation property guaranteed by the type system of the metalanguage:

```
data Rep a = Zero | Succ | Cond | S | K | Y | Eval | Reif | Refl
           | (Rep (b -> a)) @ (Rep b)

coerce   :: Rep a -> Rep b
intp   :: Rep a -> Rep a
apply  :: Rep a -> Rep a

coerce Zero     = Zero
coerce Succ     = Succ
coerce Cond     = Cond
coerce S        = S
coerce K        = K
coerce Y        = Y
coerce Eval     = Eval
coerce Reif     = Reif
coerce Refl     = Refl
coerce (x @ y)  = coerce x @ y

intp (Reif @ x) = Reif @ x
intp (x @ y)    = apply (intp x @ intp y)
intp e          = e

apply (Succ @ x)                  = Succ @ x
apply (Cond @ Zero @ c @ f)       = intp(coerce c)
apply (Cond @ (Succ @ x) @ c @ f) = intp(coerce f @ coerce x)
apply (S @ f @ g @ x) =
                intp(coerce f @ coerce x @ (coerce g @ coerce x))
apply (K @ x @ y)                 = intp(coerce x)
apply (Y @ f)                     = intp(coerce f @ (Y @ f))
apply (Eval @ (Reif @ x))         = Reif @ intp x
apply (Refl @ (Reif @ x))         = intp(coerce x)
apply e                           = e
```

Figure 2: Representation and self-interpretation of SK terms

If **`intp e`** evaluates to **`e'`**, where **`e`** is of type **`Rep t`**, then **`e'`** is also of type **`Rep t`**.

5 Reflection and Reification

Reflection is the ability to convert data (representations of SK expressions) into programs (SK expressions themselves). To implement the function **`reflect`**, we extend the representation of an SK expression to contain not only the representation (of type **`Rep a`**), but also the expression itself (of type **`a`**). Reflection then simply corresponds to the selection of the second component. The extended type **`Exp`** is defined in Figure 3.

Self-interpretation is now handled by the function **`eval`**, which calls **`intp`** on the first component and retains the second component of the extended representation. Lazy evaluation ensures that the second component is not evaluated until it is extracted by a call to **`reflect`**.

Reification, the ability to convert programs back to data, is provided only from natural numbers to representations of natural numbers. Although every program in the metalanguage can be represented as an SK expression, the reification of functions would require reflective capabilities of the metalanguage itself. Systematic overloading in Haskell is used to provide a function **`reify`** of the polymorphic type

```
reify :: (Reify a) => a -> Exp a
```

```
data Exp a = Exp (Rep a) a

eval    :: Exp a -> Exp a
reflect :: Exp a -> a
(&)     :: Exp (a -> b) -> (Exp a) -> Exp b

eval (Exp e v) = Exp (intp e) v
reflect (Exp e v) = v
(Exp e u) & (Exp f v) = Exp (e @ f) (u v)

class Reify a where
  reify :: a -> Exp a

instance Reify Int where
  reify 0 = zero
  reify x = succ & reify (x - 1)
```

Figure 3: Extended representation, reflection, and reification of SK terms

although **`reify`** is implemented only on natural numbers. Without this mechanism, we could not represent the metacircular combinator **`reif`** in the extended representation with its correct type.

Independently of the type preservation property of **`intp`**, there is no guarantee that the syntax trees constructed using the value constructors of type **`Rep`** represent a well-typed SK expression. For example, the syntax tree **`Y @ K`** can be constructed and can be coerced to have any type **`Rep t`**, but does not represent a well-typed SK expression. Fortunately, we can enforce that only well-typed syntax trees are constructed using our extended representation. For each basic combinator, we declare an extended representation consisting of the combinators representation by a constructor from type **`Rep`** and the SK expression of the combinator given in the metalanguage. The explicit type declarations are not necessary to give the combinators the desired types, but are included for clarity. The representations of the extended combinators are declared in Figure 4

Well-typed syntax trees can now be constructed from the extended representations of the combinators and the infix operation **`(&)`**, a wrapper around the constructor **`@`**, or by reifying a natural number. The subsequent examples demonstrate the functionality of the interpreter. The representations for **`plus`** and **`fact`** were obtained by an automated translation of their λ-expressions to SK expressions.

```
eval (plus & (reify 2) & (reify 1))
=> succ & (succ & (succ & zero))

reflect (plus & (reify 2) & (reify 1))
=> 3

eval (refl & (i & (reif & two)))
=> succ & (succ & zero)

reflect (i & fact) 7
=> 5040

reflect ((i & refl) & ((i & reif) & two))
=> 2

y & k
=> [63] Cannot unify types (occurence): a and b -> a
=> in y & k
```

```
zero    :: Exp Int
succ    :: Exp (Int -> Int)
cond    :: Exp (Int -> a -> (Int -> a) -> a)
s       :: Exp ((a -> b -> c) -> (a -> b) -> a -> c)
k       :: Exp (a -> b -> a)
i       :: Exp (a -> a)
y       :: Exp ((a -> a) -> a)
evl     :: Exp (Exp a -> Exp a)
reif    :: (Reify a) => Exp (a -> Exp a)
refl    :: Exp (Exp a -> a)

zero    = Exp Zero 0
succ    = Exp Succ (+1)
cond    = Exp Cond (\x c f -> if x == 0 then c else f (x - 1))
s       = Exp S    (\f g x -> f x (g x))
k       = Exp K    (\x y -> x)
i       = s & k & k
y       = Exp Y    fix where fix h = h (fix h)
evl     = Exp Eval eval
reif    = Exp Reif reify
refl    = Exp Refl reflect
```

Figure 4: Types and values of the representations of the basic SK combinators

6 Conclusion

To our knowledge, we are the first to implement a fully metacircular interpreter for a statically typed, Turing-complete language. By contrast, the typed interpreter of [5] is based on a “types as values” assumption and hence relies on some form of dynamic typing. Pfenning and Lee present [13] an embedding of the types of F_2 in F_3 but stop short of true reflection of a language in itself.

We argued [10] that conventional statically typed object-oriented languages were not sufficient to guarantee well-typed expression syntax trees and described an alternative metalanguage based on F-bounded polymorphism [1] and existential quantification. While such a language was not available initially, we were now able to use a reasonably wide-spread “stock” functional language for our purposes.

We demonstrated that each feature of the metalanguage, lazy evaluation, systematic overloading, existential quantification, and polymorphic recursion, plays a crucial role in the implementation of our metacircular interpreter. We were able to avoid redundancy by relying completely on the type system of the metalanguage.

References

[1] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proc. Functional Programming and Computer Architecture*, pages 273–280, 1989.

[2] P. Cointe. Metaclasses are first-class: The ObjVlisp model. In *Proc. ACM Conf. Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, pages 156–167, 1987.

[3] B. Foote and R. Johnson. Reflective facilities in Smalltalk-80. In *Proc. ACM Conf. Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, pages 156–167, 1987.

[4] D. Friedman and M. Wand. Reification: Reflection without metaphysics. In *Proc. ACM Symposium on Lisp and Functional Programming*, pages 327–336, 1984.

[5] M. Hagiya. Meta-circular interpreter for a strongly typed language. *J. Symb. Comp.*, (8):651–680, 1989.

[6] P. Hudak, S. Peyton-Jones, P. Wadler, et al. Report on the programming language Haskell A non-strict, purely functional language Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[7] M. Jones. Polymorphic recursion in Haskell, July 1993. Posted to the Haskell mailing list.

[8] K. Läufer. *Polymorphic Type Inference and Abstract Data Types*. PhD thesis, New York University, Department of Computer Science, July 1992. Appeared as NYU-CS Report 622.

[9] K. Läufer. An extension of Haskell with first-class abstract types, July 1993. Submitted to JFP.

[10] K. Läufer and M. Odersky. Reflection in type systems. In *Proc. OOPSLA Workshop on Reflection and Metalevel Architectures*. ACM, October 1991.

[11] J. Mitchell and R. Harper. The essence of ML. In *Proc. Symp. on Principles of Programming Languages*. ACM, January 1988.

[12] R. Muller. On self-interpretation and reflection in lambda-calculus. In *Proc. OOPSLA Workshop on Reflection and Metalevel Architectures*. ACM, October 1991.

[13] F. Pfenning and P. Lee. Metacircularity in the polymorphic lambda-calculus. Technical Report CMU-CS-89-207, CMU, December 1989.

[14] D. Turner. A new implementation technique for applicative languages. *Software — Practice and Experience*, 9:31–49, 1979.

[15] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, pages 60–76. ACM, January 1989.