Self-Interpretation and Reflection in a Statically Typed Language

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

Martin Odersky
Universität Karlsruhe
odersky@ira.uka.de

August 5, 1993

In Proc. OOPSLA '93 Workshop on Reflection and Metalevel Architectures

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 [9].

By contrast, we consider self-interpretation and reflection in a statically typed metalanguage along the lines of [10] and [7]. 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: If a representation is well-typed, then 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 [11]. 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:

$$\text{APP} \qquad \frac{\vdash e : \tau' \to \tau \qquad \vdash e : \tau}{\vdash e\ e' : \tau}$$

3 The Metalanguage

We use the lazy functional language Haskell [6] as the metalanguage for the implementation of our metacircular interpreter. Haskell static typing with parametric polymorphism and systematic overloading. We briefly describe lazy evaluation and systematic overloading.

Lazy evaluation

In lazy or call-by-need evaluation, expressions are not evaluated until they are actually needed. This includes actual parameter expressions in function calls. 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`**.

Systematic overloading

Haskell provides *type classes* [12] 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`**.

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 [9]. The interpreted program is a simple SK combinator expression, *represented* by its abstract syntax tree of type **`Rep`**, defined in Figure 2. An abstract syntax tree represents either a basic

```
data Rep = Zero | Succ | Cond | S | K | Y | Eval | Reif | Refl
         | Rep @ Rep

intp    :: Rep -> Rep
apply   :: Rep -> Rep

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

Figure 2: Representation and self-interpretation of SK terms

combinator or the application of one expression to another, as expressed by the left-associative infix constructor **`@`** in the last variant of **`Rep`**. 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 can be reduced 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.

There is no guarantee that the syntax trees we build from the value constructors of type **`Rep`** represent well-typed SK expressions. For example, the syntax tree **`Zero @ Zero`** can be constructed and has type **`Rep`**, but does not represent a well-typed SK expression. The next section describes how this problem is solved.

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`**), but also the expression itself (of some type **`a`**). Reflection then simply corresponds to the selection of the second component. The extended type **`Exp a`** 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

```
data Exp a = Exp Rep 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)

instance (Reify a) => Reify (Exp a) where
  reify e = reif & e
```

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

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
```

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.

We saw in the previous section that even ill-typed combinator expressions could be represented within the type **Rep**. Fortunately, our extended representation **Exp** allows us to enforce that only well-typed expressions can be constructed. For each basic combinator, we declare an extended representation consisting of the corresponding constructor of type **Rep** and the corresponding λ-expression in the metalanguage. Explicit type declarations are not required for the desired typing of the combinators, 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))
```

```
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

```
=> 2

reflect (reflect (reflect (reif & (reif & (reif & zero)))))
=> zero

zero & zero
=> [65] Cannot unify types: a -> b and Int
=> in (&) zero
```

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 [10] an embedding of the types of F_2 in F_3 but stop short of true reflection of a language in itself.

We argued [7] 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 [8]. While such a language was not available at that time, 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, static typing, and systematic overloading, 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] K. Läufer and M. Odersky. Reflection in type systems. In *Proc. OOPSLA Workshop on Reflection and Metalevel Architectures*. ACM, October 1991.

[8] J. Mitchell and G. Plotkin. Abstract types have existential type. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988.

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

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

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

[12] 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.