Duplication and Partial Evaluation to Implement Reflective Languages (Extended Abstract)

Kenichi Asai Satoshi Matsuoka Akinori Yonezawa

Department of Information Science, Faculty of Science,
The University of Tokyo
7-3-1 Hongou, Bunkyo-Ku, Tokyo, 113, Japan
{asai,matsu,yonezawa}@is.s.u-tokyo.ac.jp

1 Introduction

Ichisugi et al. have presented in [4] a reflective language Rscheme, in which user programs can redefine the functions comprising the interpreter which is actually executing the programs at one level above[1]. As we can redefine (parts of) the interpreter, we can extend or change the behavior of the interpreter in various ways. By documenting the protocols and behavior of the functions, we can provide users a reflective language with high flexibility and extensibility.

In general, how to implement such a flexible language, however, is not obvious, and the structure of Rscheme is complicated. The purpose of our present work is to give a general structure which enables such flexible reflective languages. In fact, we show that it can be constructed by the combination of three simple concepts: a meta-circular interpreter, duplication of the metalevel, and partial evaluation. We illustrate this method using a reflective language called *Black*, but it is general enough to be applied to other reflective languages.

2 A Reflective Language Black

The syntax of Black is the same as Scheme with one additional reflective construct `(exec-at-metalevel E)`. In Black, a baselevel expression (*level 0*) is interpreted by the interpreter running at the level above (*level 1, metalevel*), which is in turn interpreted by the interpreter running at level 2, and so on. Conceptually, there is an infinite tower of interpreters in

[1]We call these functions *evaluator functions*.

Black. The construct `(exec-at-metalevel E)` is used to evaluate the expression `E` one level above.

Figure 1 shows the interpreter at level n, which interprets Black programs of level $n-1$. The interpreter is basically an usual continuation passing style Scheme interpreter except for the function $\texttt{eval-exec-at-metalevel}_n$. It executes the body of `exec-at-metalevel` at metalevel by calling $\texttt{eval}_{n+1}$ running one level above. We write this interpreter $\texttt{L}_n^{env_n}(\cdot)$. We can then write an infinite tower of interpreters as follows:

$$\ldots(\texttt{L}_3^{env_3}(\texttt{L}_2^{env_2}(\texttt{L}_1^{env_1}(exp))))\ldots.$$

3 Implementation Difficulties

The major challenges in constructing reflective languages are:

- How can we implement an infinite tower?
- How do we implement it efficiently?

Conventionally, these have been solved by creating *directly implemented* interpreters *lazily*. For the former, Black uses the lazy creation technique as usual. For the latter, however, the naive direct implementation technique does not work, because Black allows the redefinition of evaluator functions. If we directly executed the functions in Figure 1 instead of interpreting them on the metalevel interpreter, redefinition of evaluator functions would not work since their behavior is determined in machine code. To implement evaluator functions directly without losing the ability to redefine them, we *duplicate* the metalevel interpreter.

4 Duplicating the Metalevel Interpreter

Suppose we are constructing the directly implemented version of the level n interpreter. To do this, we first duplicate the metalevel interpreter $\texttt{L}_{n+1}^{env_{n+1}}$. The duplicated interpreter, written as $\mathcal{L}_{n+1}^{env_{n+1}}$, is the same as $\texttt{L}_{n+1}^{env_{n+1}}$ except for a single point. It does not include the function $\texttt{eval-exec-at-metalevel}_{n+1}$, that is, it consists of the functions in $\texttt{L}_{n+1}^{env_{n+1}}$ other than $\texttt{eval-exec-at-metalevel}_{n+1}$[2]. Since it does not include $\texttt{eval-exec-at-metalevel}_{n+1}$, it is the same as the conventional meta-circular interpreter. Thus, we can compile it into machine code to obtain directly implemented version of $\mathcal{L}_{n+1}^{env_{n+1}}$.

Given the duplicated interpreter $\mathcal{L}_{n+1}^{env_{n+1}}$, we now show that $\mathcal{L}_{n+1}^{env_{n+1}}(\texttt{L}_n^{env_n}(\cdot))$ is the interpreter that we want: the interpreter which can be directly implemented and still allows the redefinition of the component functions. Reconsider $\texttt{L}_n^{env_n}$ shown in Figure 1. $\texttt{L}_n^{env_n}$ itself does not use `exec-at-metalevel`, which means that $\mathcal{L}_{n+1}^{env_{n+1}}$ can be used instead

[2]We would also need to throw away the following clause in the $\texttt{eval}_n$:

```
((eq?  (car exp_n) 'exec-at-metalevel) (eval-exec-at-metalevel_n exp_n env_n cont_n))
```

```
(define (eval_n exp_n env_n cont_n)
  (cond ((number? exp_n) (cont_n exp_n))
        ((symbol? exp_n) (eval-var_n exp_n env_n cont_n))
        ((eq? (car exp_n) 'quote) (eval-quote_n exp_n env_n cont_n))
        ((eq? (car exp_n) 'if) (eval-if_n exp_n env_n cont_n))
        ((eq? (car exp_n) 'set!) (eval-set!_n exp_n env_n cont_n))
        ((eq? (car exp_n) 'lambda) (eval-lambda_n exp_n env_n cont_n))
        ((eq? (car exp_n) 'exec-at-metalevel)
         (eval-exec-at-metalevel_n exp_n env_n cont_n))
        (else (eval-list_n exp_n env_n
                          (lambda (l) (apply_n (car l) (cdr l) env_n cont_n))))))
(define (eval-var_n exp_n env_n cont_n) (cont_n (get exp_n env_n)))
(define (eval-quote_n exp_n env_n cont_n) (cont_n (car (cdr exp_n))))
(define (eval-if_n exp_n env_n cont_n)
  (eval_n (car (cdr exp_n)) env_n
         (lambda (pred)
           (if pred (eval_n (car (cdr (cdr exp_n))) env_n cont_n)
                    (eval_n (car (cdr (cdr (cdr exp_n)))) env_n cont_n)))))
(define (eval-set!_n exp_n env_n cont_n)
  (let ((var (car (cdr exp_n)))
        (body (car (cdr (cdr exp_n)))))
    (eval_n body env_n (lambda (data) (set-cdr! (assq var env_n) data)
                                      (cont_n var)))))
(define (eval-lambda_n exp_n env_n cont_n)
  (let ((body (cdr (cdr exp_n)))
        (params (car (cdr exp_n))))
     (cont_n (list 'lambda params body env_n))))
(define (eval-exec-at-metalevel_n exp_n env_n cont_n)
  (eval_n+1 (car (cdr exp_n)) env_n+1 cont_n))
(define (eval-list exp_n env_n cont_n)
  (if (null? exp_n)
      (cont_n '())
      (eval_n (car exp_n) env_n
             (lambda (val1)
               (eval-list_n (cdr exp_n) env_n
                           (lambda (val2) (cont_n (cons val1 val2))))))))
(define (apply_n operator operand env_n cont_n)
  (cond ((procedure? operator)
         (cont_n (scheme-apply operator operand)))
        ((and (pair? operator)
              (eq? (car operator) 'lambda))
         (let ((params        (car (cdr operator)))
               (body     (car (cdr (cdr operator))))
               (env (car (cdr (cdr (cdr operator))))))
           (eval-list_n body
                       (extend env params operand)
                       (lambda (l) (cont_n (car (last-pair l)))))))
        (else (error "Not a function" operator))))
```

Figure 1: The level n interpreter

of $\mathtt{L}_{n+1}^{env_{n+1}}$ to interpret $\mathtt{L}_n^{env_n}$. Because $\mathcal{L}_{n+1}^{env_{n+1}}$ is directly implemented, $\mathcal{L}_{n+1}^{env_{n+1}}(\mathtt{L}_n^{env_n}(\cdot))$ can also be directly executed. Thus, if we regard $\mathcal{L}_{n+1}^{env_{n+1}}(\mathtt{L}_n^{env_n}(\cdot))$ as the level n interpreter, we get the directly executable interpreter which allows us to redefine the functions in $\mathtt{L}_n^{env_n}$. The whole tower structure becomes like this:

$$\ldots(\underbrace{\mathcal{L}_4^{env_4}(\mathtt{L}_3^{env_3}}_{level3}(\underbrace{\mathcal{L}_3^{env_3}(\mathtt{L}_2^{env_2}}_{level2}(\underbrace{\mathcal{L}_2^{env_2}(\mathtt{L}_1^{env_1}}_{level1}(exp))))))\ldots.$$

Here, $\mathcal{L}_{n+1}^{env_{n+1}}(\mathtt{L}_n^{env_n}(\cdot))$ behaves like a primitive procedure, rather than being interpreted by $\mathcal{L}_{n+2}^{env_{n+2}}(\mathtt{L}_{n+1}^{env_{n+1}}(\cdot))$, since $\mathcal{L}_{n+1}^{env_{n+1}}(\mathtt{L}_n^{env_n}(\cdot))$ is directly running. Each level is connected by calling $\mathtt{eval}_{n+1}$ of $\mathtt{L}_{n+1}^{env_{n+1}}$ from $\mathtt{eval\text{-}exec\text{-}at\text{-}metalevel}_n$ of $\mathtt{L}_n^{env_n}$. By creating upper interpreters lazily, we can implement the infinite tower.

Using $\mathcal{L}_n^{env_n}(\cdot)$ as the level n interpreter is entirely different from using $\mathcal{L}_{n+1}^{env_{n+1}}(\mathtt{L}_n^{env_n}(\cdot))$. If we use $\mathcal{L}_n^{env_n}(\cdot)$, its behavior is determined in machine code. Even if we redefine functions at metalevel, it would not affect $\mathcal{L}_n^{env_n}(\cdot)$. If we use $\mathcal{L}_{n+1}^{env_{n+1}}(\mathtt{L}_n^{env_n}(\cdot))$, on the other hand, we can change $\mathtt{L}_n^{env_n}$, because $\mathtt{L}_n^{env_n}$ is actually interpreted by $\mathcal{L}_{n+1}^{env_{n+1}}$.

5 Employing Partial Evaluation Technique

The interpreter presented in the previous section is quite slow, because $\mathtt{L}_n^{env_n}$ is always interpreted by $\mathcal{L}_{n+1}^{env_{n+1}}$. To get more efficient interpreter, we partially evaluate $\mathcal{L}_{n+1}^{env_{n+1}}$ by specializing on the argument $\mathtt{L}_n^{env_n}$. Let the result be $\mathcal{L}_n^{env_n,env_{n+1}}$. As a function, it is exactly the same as $\mathcal{L}_{n+1}^{env_{n+1}}(\mathtt{L}_n^{env_n}(\cdot))$, but it runs much more efficiently. By carefully inspecting $\mathcal{L}_n^{env_n,env_{n+1}}$, it turns out to be very much similar to $\mathtt{L}_n^{env_n}$, except for accessing the metalevel environment and doing some work which should be done in the metalevel. Using $\mathcal{L}_n^{env_n,env_{n+1}}$, the infinite tower becomes:

$$\ldots(\mathcal{L}_3^{env_3,env_4}(\mathcal{L}_2^{env_2,env_3}(\mathcal{L}_1^{env_1,env_2}(exp))))\ldots.$$

The resulting interpreter is quite efficient, and we have actually constructed a Black interpreter based on this method, which runs almost as efficiently as the conventional metacircular interpreter.

6 The Effect of Duplication

To see the effect of duplication, let us consider the difference between direct execution and interpretation. When expressions are interpreted, we can freely change the expressions by redefining them at the same level they are running. When expressions are directly executed, however, they cannot be redefined because their behavior is determined in machine code. Then, why can the Black interpreter be directly executed without losing the ability to redefine evaluator functions? This is because $\mathtt{L}_{n+1}^{env_{n+1}}$ and $\mathcal{L}_{n+1}^{env_{n+1}}$ share the metalevel environment (env_{n+1}).

In the Black interpreter, $\mathtt{L}_n^{env_n}$ is interpreted by $\mathcal{L}_{n+1}^{env_{n+1}}$, instead of $\mathtt{L}_{n+1}^{env_{n+1}}$. This means that changes made on $\mathtt{L}_{n+1}^{env_{n+1}}$ do not affect the behavior of $\mathtt{L}_n^{env_n}$. Because $\mathtt{L}_{n+1}^{env_{n+1}}$ and

$\mathcal{L}_{n+1}^{env_{n+1}}$ share the environment env_{n+1}, however, changes to env_{n+1} will affect the behavior of $\mathcal{L}_{n+1}^{env_{n+1}}$ and thus that of $\mathtt{L}_n^{env_n}$. So, the effect of duplication can be said to be giving up changeability in favor of direct implementation.

But we cannot give it all up for direct implementation. By sharing the environment, we leave the system flexible. Is it enough to share only the environment? We believe yes, because if we can change the environment, we can redefine all the evaluator functions freely. Various changes and extensions are possible by redefining them, and if we want, we can entirely replace the interpreter with the new user-defined interpreter.

7 Related Work

Jefferson and Friedman have implemented in [5] a finite reflective tower by executing a meta-circular interpreter on top of itself. By duplicating and partially evaluating their interpreter, we can obtain an infinite reflective tower which is directly executable. The global environment is shared by all the levels in [5] because of the efficiency consideration, but in Black, each level has its own global environment.

Brown[3, 7] and Blond[1] are Lisp-based reflective languages which also achieve the infinite tower of directly executed interpreters. However, they do not allow to redefine the interpreter. In the Black framework, this is the case where nothing is shared when the metalevel interpreter is duplicated. We have achieved a more flexible interpreter by sharing the metalevel environment.

Black is invented independently of 3-LISP[2, 6], but the implementations of both seem to be quite similar. We hope that our framework will provide a clear understanding of the implementation of 3-LISP.

8 Conclusion

By duplicating a conventional meta-circular interpreter and partially evaluating it, we can implement flexible reflective languages which allow user programs to redefine the interpreter actually running at one level above. We have implemented the language Black in this framework and obtained a system which runs almost as efficiently as the conventional meta-circular interpreter when reflection is not used. The technique shown here is simple enough and we think that it can serve for deepening general understanding of reflection.

Acknowledgements

We are grateful to Hidehiko Masuhara for his suggestions and extensive discussions on reflection. We would also like to thank Yuuji Ichisugi, Takuo Watanabe, and Shigeru Chiba for their helpful comments.

References

[1] Danvy, O., and K. Malmkjær "Intensions and Extensions in a Reflective Tower," *Conference Record of the 1988 ACM Symposium on Lisp and Functional Programming*, pp. 327–341 (July 1988).

[2] des Rivières, J., and B. C. Smith "The Implementation of Procedurally Reflective Languages," *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pp. 331–347 (August 1984).

[3] Friedman, D. P., and M. Wand "Reification: Reflection without Metaphysics," *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pp. 348–355 (August 1984).

[4] Ichisugi, Y., S. Matsuoka, and A. Yonezawa "RbCl: A Reflective Object-Oriented Concurrent Language without a Run-time Kernel," *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, pp. 24–35 (November 1992).

[5] Jefferson, S., D. P. Friedman "A Simple Reflective Interpreter," *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, pp. 24–35 (November 1992).

[6] Smith, B. C. "Reflection and Semantics in Lisp," *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, pp. 23–35 (January 1984).

[7] Wand, M., and D. P. Friedman "The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower," *Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming*, pp. 298–307 (August 1986).