記　号　処　理　60－3
プログラミング—言語・基礎・実践—　2－3
（1991. 6. 21）

リフレクティブな言語の代数的仕様

栗原 正仁　　大内 東
北海道大学 情報工学科

因果結合により計算システムが自分自身の計算に関して計算することを可能にする自己反映（リフレクション）計算機能をもつ関数型言語 Brown の代数的仕様を示す。仕様の大部分は、環境、継続、メタ継続をモデル化する表示的意味論に基づく等式で表わされた公理の集合から成っている。等式は左辺のインスタンスを対応する右辺のインスタンスに置き換える書き換え規則とみなせるので、この仕様は項書き換えにより実行できる。

AN ALGEBRAIC SPECIFICATION OF A REFLECTIVE LANGUAGE

Masahito Kurihara　　Azuma Ohuchi

Department of Information Engineering, Hokkaido University

Kita 13, Nishi 8, Kita-ku, Sapporo 060, Japan

We present an algebraic specification of a functional programming language (called Brown) with 'reflective' computation mechanisms which enable the computational system to compute about its own computation in a causally connected way. The major part of the specification consists of a set of equational axioms based on a denotational semantics which models environment, continuation and 'metacontinuation.' The specification is executable by term rewriting, because the equations can be regarded as the rewrite rules which replace instances of the left-hand sides with the corresponding instances of the right-hand sides.

1 Introduction

Computational reflection is the activity performed by a *reflective* computational system which computes about its own computation in a causally connected way [5]. Many reflective languages have been proposed in the frameworks of functional programming [3,8,12], logic programming [9,11] and object-oriented programming [10,13,14]. In particular, Wand and Friedman [12] presented a clear semantics of reflection, based on a denotational semantics which models environment, continuation and 'metacontinuation'. Two basic processes, *reification* and *reflection*, were introduced and implemented in Scheme, a dialect of Lisp, to describe a reflective language called Brown which is a dialect of Lisp with reflective facilities.

The implementation was done essentially in lambda calculus which, although suitable for mathematically rigorous description of computation, seems unsuitable for communication among people, and thus for specification. Unlike Lisp and other practical languages, all functions are fully 'curried' and higher-order. Forms are evaluated in normal order (call-by-name) unless call-by-value is simulated by applicative-order Y combinators. No practical data types are introduced. As a result, the description seems to be a little complicated as a specification, especially when we try to extend and implement the language (Brown) in other languages such as C, Modula-2, etc.

In this paper we try to specify Brown in the framework of algebraic specification. Algebraic specification techniques [2] are now considered as one of the most promising approaches to specification writing in every phase of software development. For example, in the field of programming language design, the method combination scheme in CLOS (Common Lisp Object System) was specified by an algebraic specification [7]. For more current trends, consult Futatsugi [4]. Part of the emphasis in this paper is put on how we can specify the reflection abstractly in equational logic and, in other words, how we can specify the reflective facilities as abstract data types.

2 Specification (Part I)

In this and the subsequent sections, we describe a Lisp-like reflective language called Brown [12] in an algebraic specification language. The syntax of Brown is similar to a classical Lisp. It has identifiers, abstractions for which the notation `(lambda (id ...) body)` is used, and application of a function to any number of arguments. Unlike Common Lisp (and like Scheme), the first element (function part) of application forms is evaluated and the result is used as a function. For simplicity, we do not specify the side effects on environments (e.g., by set! operation) and I/O (by read/write), because the emphasis in this paper is put on reflection. But remember that such side effects could be specified either formally or informally, being accommodated in the implementation.

In the specification, we introduce the following sort symbols:

`BF`	: Brown functions
`Exp`	: Brown expressions
`Exp*`	: lists of Brown expressions
`Id`	: identifiers
`Id*`	: lists of identifiers
`Env`	: environments
`K`	: continuations
`MK`	: metacontinuations
`A`	: some domain of answers
`V`	: Brown values
`V*`	: lists of Brown values
`N`	: natural numbers

Since an identifier can be an expression, `Id` and `Id*` are subsorts of `Exp` and `Exp*`, respectively. `N` is a subsort of `Exp`. Since Brown is a dialect of Lisp, a Brown value can be a program and vice versa, so `Exp` is equivalent to `V`.

The rest of this section consists of a set of declarations of function symbols, and a system of commented equations which specifies Brown without reflective facilities. (The declarations of variables are omitted, and reflective facilities will be specified in the next section.) The specification is executable by term rewriting [1,6], because the equations can be viewed as the rewrite rules which rewrite terms by using the pattern matching and

the subterm replacement.

2.1 Continuation and metacontinuation

A continuation is a computational object that describes the control context. It is typically modelled by a function which receives a value from the previous calculation and then finishes the entire calculation of the current interpreter.

A metacontinuation represents the states of the interpreters above the current level in the reflective tower. It is typically modelled by a function which receives a value from the lower interpreter and then finishes the entire calculation of all the interpreters above the current level.

In our framework of algebraic specification, a continuation and a metacontinuation are modelled *abstractly* (in the sense of abstract data types) through the function pass:

```
pass : V K MK -> A
```

pass[v,k,mk] passes the value v to the continuation k, and the output from k further to the metacontinuation mk, to obtain the final answer.

The primitive operations on continuations and metacontinuations are meta-car, meta-cdr and meta-cons:

```
meta-car  : MK -> K
meta-cdr  : MK -> MK
meta-cons : K MK -> MK
```

Since in Brown a metacontinuation is essentially a stack of continuations, we can specify the operations with the following equations:

```
meta-car[meta-cons[k, mk]] = k
meta-cdr[meta-cons[k, mk]] = mk
```

2.2 Environment

An environment is a set of bindings of identifiers with their values, and is constructed from the empty environment empty-env and the extend operation. extend[r, pl, vals] extends the environment r by binding the formal parameters pl with the actual values vals.

Environments are accessed with the function rib-lookup.
rib-lookup[id,r,k,mk] looks up the identifier id in the environment r, and passes to the continuation k the place ('cons' cell) in (the 'car' part of) which the associated value is stored; the value from k is further passed to the metacontinuation mk. The following specification introduces the auxiliary function lookup to define rib-lookup:

```
empty-env : -> Env
extend : Env Id* Exp* -> Env
rib-lookup : Id Env K MK -> A
lookup : Id Id* Exp* Env K MK -> A

rib-lookup[id, empty-env, k, mk]
 = pass[(UNASSIGNED), k, mk]

rib-lookup[id, extend[r,names,vals],k,mk]
 = lookup[id, names, vals, r, k, mk]

lookup[id, (), (), r, k, mk]
 = rib-lookup[id, r, k, mk]

lookup[id, (name . names), (val . vals),
       r, k, mk]
 = if eq[id, name]
   then pass[(val . vals), k, mk]
   else lookup[id, names, vals, r, k, mk]
```

where eq, which we assume is imported from an appropriate module, checks the equality of identifiers. The expression "if x then y else z" is identical to the term if[x,y,z] which satisfies the following equations:

```
if[true, x, y] = x
if[false, x, y] = y
```

2.3 Initial environment

The global environment, which will be shared by all the interpreter levels, are constructed by installing several primitives in the empty environment. In the following, we include only eight functions as an example set of primitives for the illustrating purposes:

```
global-env : -> Env
prim-names : -> Id*
prim-vals : Id* -> V*
primitive : Id -> BF

global-env
  = extend[empty-env,
    (T NIL MEANING . prim-names),
    (T NIL mean . prim-vals[prim-names])]

prim-names
 = (CAR CDR CONS + MAKE-REIFIER)

prim-vals[()] = ()

prim-vals[(name . names)]
  = (primitive[name] . prim-vals[names])
```

The primitive functions are specified by call. call[id,vals,k,mk] calls the primitive (built-in) function id with the list of actual arguments vals, and passes the result to k and further to mk.

```
call : Id V* K MK-> A

call[CAR, ((x . y)), k, mk]
 = pass[x, k, mk]
call[CDR, ((x . y)), k, mk]
 = pass[y, k, mk]
call[CONS, (x y), k, mk]
 = pass[(x . y), k, mk]
call[+, (x y), k, mk]
 = pass[x + y, k, mk]
call[MAKE-REIFIER, (bf), k, mk]
 = pass[reifier[bf], k, mk]
```

where we assume that x + y represents the (built-in) addition. The specification of reifier and mean will be given in the next section (Part II).

2.4 Denotation

The functions that play the central role in the specification are denote and apply as well as pass.

```
denote : Exp Env K MK -> A
apply : BF Exp* Env K MK -> A
```

denote[e,r,k,mk] evaluates the Brown expression e in the environment r and passes the result to the continuation k. The output from the continuation is further passed to the metacontinuation mk, to obtain the final answer.

```
denote[n, r, k, mk]
 = pass[n, k, mk]  when n is number
```

This equation means that if the given expression is a number, it should be simply passed to the continuation, because a number is a self-evaluating object.

```
&deref : K -> K

denote[id, r, k, mk]
 = rib-lookup[id, r, &deref[k], mk]
   when id is symbol

pass[(val . x), &deref[k], mk]
 = pass[val, k, mk]
```

If the expression is a symbol (an identifier), rib-lookup is used to find the value associated with the symbol in the given environment. The cell containing the value is passed to the continuation &deref[k] which then passes to the continuation k the value contained in the cell.

These two equations indicate how we represent continuations in our framework of algebraic specification. In our specification, a continuation is implicitly specified abstractly by the interactions with the function pass, rather than explicitly by a lambda expression. To state more specifically, we introduce particular functions[1] (such as &deref) to make a continuation from necessary information (such as k). Then the equation for pass which takes this continuation as the second argument specifies how the continuation will continue the computation when it receives a datum from the previous computation.

```
closure : Exp Id* Env -> BF

denote[(LAMBDA pl body), r, k, mk]
 = pass[closure[body, pl, r], k, mk]
```

[1] For convenience, the names of the functions which make a continuation begin with the letter '&'.

If the expression is a lambda expression, the closure which encapsulates the body, the parameter list and the environment is passed to the continuation.

```
 &apply : Exp* Env K -> K

denote[(fun . args), r, k, mk]
 = denote[fun, r, &apply[args, r, k], mk]
   when fun <> LAMBDA

pass[bf, &apply[args, r, k], mk]
 = apply[bf, args, r, k, mk]
```

If the expression is a function application form, the first element of the form (which is expected to be evaluated to a Brown function) is evaluated, and the result is passed to the continuation `&apply[args, r, k]` which, when it receives a Brown function, applies the function to the unevaluated arguments `args` in the environment `r`, and passes the result to the continuation `k`.

Now we specify `apply`. `apply[bf,e,r,k,mk]` applies the Brown function `bf` to the list of unevaluated expressions `e` in the environment `r`, and passes the result to `k`, and further to `mk`.

```
 &prim-call : Id K -> K
 &eval-body : Exp Id* Env K -> K

apply[primitive[id], args, r, k, mk]
 = eval-args[args,r,&prim-call[id,k],mk]

pass[vals, &prim-call[id, k], mk]
 = call[id, vals, k, mk]

apply[closure[body,pl,r1],args,r,k,mk]
 = eval-args[args,r,
             &eval-body[body,pl,r1,k],mk]

pass[vals, &eval-body[body,pl,r1,k], mk]
 = denote[body,extend[r1,pl,vals],k,mk]
```

When the function is either a primitive (with name `id`) or a closure, the arguments are evaluated from left to right and passed to a suitable continuation. If the function is a primitive, it is called with the evaluated arguments. If the function is a closure, its body is evaluated in the environment which extends the encapsulated environment by binding the parameters with the evaluated arguments.

The argument evaluation process is defined by `eval-args`:

```
 eval-args : Exp* Env K MK -> A
 &eval-args : Exp* Env K -> K
 &cons : V Env K -> K

eval-args[(), r, k, mk]
 = pass[(), k, mk]

eval-args[(arg . args), r, k, mk]
 = denote[arg,r,&eval-args[args,r,k],mk]

pass[val, &eval-args[args, r, k], mk]
 = eval-args[args,r,&cons[val,r,k],mk]

pass[vals, &cons[val, r, k], mk]
 = pass[(val . vals), k, mk]
```

If the argument list is empty, the empty list is passed to the continuation. Otherwise, the first element of the list is evaluated and passed to the continuation `&eval-args[args, r, k]`, which then evaluates recursively the rest of the arguments and passes the result to the continuation `&cons[val, r, k]`, which constructs and passes to the continuation `k` the final list of the evaluated arguments.

3 Specification (Part II)

In this section, we augment the specification in the previous section with the reflective facilities: reification and reflection.

3.1 Reification

Reification is the process in which the current expression, the current environment and the current continuation contained in the interpreter registers are passed to the program itself, suitably packaged (or reified) so that the program can manipulate them. Since environments and continuations are

represented by functions in Brown, the reification process must turn the interpreter environment and the continuation into Brown functions.

```
reifier : BF -> BF
reify-env : Env -> BF
reify-cont : K -> BF

apply[reifier[bf], args, r, k, mk]
 = apply[bf, (E R K),
         extend[r, (E R K),
                (args
                 reify-env[r]
                 reify-cont[k])],
         meta-car[mk], meta-cdr[mk]]
```

A reifier, or a reifying procedure, represented by the term `reifier[bf]`, is a Brown function turned from an ordinary Brown function `bf` with three arguments. Recall the last equation of subsection 2.3 to see that this transformation is done by the primitive `MAKE-REIFIER` in a Brown program.

When a reifier is applied, the list of the arguments is passed to `bf` as the first argument. In addition, the environment and the continuation are turned into the corresponding Brown functions by `reify-env` and `reify-cont`, and passed to `bf` as the second and the third arguments, respectively. Note that Brown functions take a list of unevaluated arguments, so actually, the list `(E R K)` is passed to `bf` in a suitable environment. At the same time, the reflective tower is "shifted up" to continue the computation under the new continuation `meta-car[mk]`, because the old continuation `k` was reified to `reify-cont[k]` which will be controlled by the Brown program.

Since we have now introduced two more constructors of Brown functions, we have to augment apply and pass with the following equations:

```
&lookup : Env K -> K

apply[reify-env[r1], (arg), r, k, mk]
 = denote[arg, r, &lookup[r1, k], mk]

pass[id, &lookup[r1, k], mk]
 = rib-lookup[id, r1, k, mk]
```

`reify-env[r1]` is the Brown function which represents the reified environment `r1`. When it is applied to a single argument, the argument is evaluated (to an identifier) and passed to the continuation which, when it receives an identifier, looks it up in `r1`, and passes the associated cell to `k`.

```
apply[reify-con[k1], (arg), r, k, mk]
 = denote[arg, r, k1, meta-cons[k, mk]]
```

`reify-con[k1]` is the Brown function which represents the reified continuation `k1`. When `reify-con[k1]` is applied to a single argument, the argument is evaluated and the resultant value is passed to the continuation `k1` after "shifting down" the reflective tower by `meta-consing` `k` to `mk`.

3.2 Reflection

According to the definition of Wand and Friedman, *reflection* is the process by which an expression, an environment and a continuation all of which are expressed in Brown are passed back to the interpreter to reinstall them in the interpreter registers. The reflection is invoked by the Brown function `mean`:

```
mean : -> BF
reflect-env : BF -> Env
&reflect-cont : BF -> K
&reflect : K -> K

apply[mean, args, r, k, mk]
 = eval-args[args, r, &reflect[k], mk]

pass[(e rb kb), &reflect[k], mk]
 = denote[e, reflect-env[rb],
          &reflect-cont[kb],
          meta-cons[k,mk]]
```

`reflect-env` and `&reflect-cont` turns a Brown environment and a Brown continuation (both of which are Brown functions) into the interpreter environment and the continuation. When the meaning function is applied to the list of arguments, the arguments are expected to be evaluated to an expression (`e`), an Brown environment

(rb) and a Brown continuation (kb), respectively; and their list is passed to the continuation which, when it receives the list, evaluates the expression e under the new environment reflecting rb, and passes the result to the new continuation reflecting kb and further to the metacontinuation meta-cons[k,mk]. Note that the environment r might be thrown away.

Since we have now introduced new constructors of environments and continuations, we have to add two more equations to rib-lookup and pass.

```
rib-lookup[id, reflect-env[rb], k, mk]
 = apply[rb,(E),
         extend[global-env,(E),(id)],
         k,mk]
```

To look up an identifier in the reflected environment reflect-env[rb], we apply the Brown environment function rb to the list of a symbol E under the environment in which E is bound to the identifier.

```
pass[v, &reflect-cont[kb], mk]
 = apply[kb,(E),
         extend[global-env,(E),(v)],
         meta-car[mk], meta-cdr[mk]]
```

To start the continuation &reflect-cont[kb] with a value v, we apply the Brown continuation function kb to the list of a symbol E under the environment in which E is bound to the value. At the same time, the reflective tower is "shifted up", so the result of the application is passed to the lowest element of the metacontinuation.

3.3 Reflective tower

Let &R-E-P[n] be the initial continuation (a read-eval-print loop) for the interpreter at level n. Thus each interpreter begins with a continuation which is a read-eval-print loop:[2]

```
&R-E-P : Exp -> K

print&prompt&read : Exp V -> Exp

pass[v, &R-E-P[prompt], mk]
 = denote[print&prompt&read[prompt, v],
          global-env, &R-E-P[prompt], mk]

print&prompt&read[prompt, v]
 = begin
    writeln[v];
    write[prompt]; read[]
   end
```

Then the infinite layer of the reflective tower is defined by the following equation augmented by two equations for meta-car and meta-cdr:

```
tower : -> MK
tower-above : N -> MK

tower = tower-above[0]

meta-car[tower-above[n]] = &R-E-P[n]

meta-cdr[tower-above[n]]
 = tower-above[n + 1]
```

You can start the system by calling boot-tower:

```
boot-tower : -> A

boot-tower[]
 = pass[STARTING-UP,
        meta-car[tower], meta-cdr[tower]]
```

where STARTING-UP can be an arbitrary value used as a dummy.

4 Conclusion

Brown was originally specified and implemented in the lambda calculus by Wand and Friedman [12]. In this paper, we have presented an abstract form of the specification in equational logic. Apparently, the implementation by Wand and Friedman is one of (and the earliest of) the possible models that satisfy our abstract specification. Our specification is first-order, consisting of a set of simple

[2] Note, however, that this is not part of the formal specification, because it contains the side effects on I/O and our specification prohibits side effects.

equations that are rigorous and easier to understand. Also, it is executable by term rewriting in an intuitively clear way. Therefore, we believe that it will help make the reflective language easier to be understood, implemented, extended and reasoned about.

Acknowledgment

This work is partially supported by the Grants from Ministry of Education, Science and Culture of Japan, #02249104; and a donation of Toshiba Corporation.

References

[1] Avenhaus, J. and Madlener, K., Term rewriting and equational reasoning, in: Banerji, R.B. Ed: *Formal Techniques in Artificial Intelligence*, North-Holland (1990)

[2] Bergstra, J.A., Heering, J. and Klint, P., Ed: *Algebraic Specification*, ACM Press (1989)

[3] Danvy, O. and Malmkjær, K., Intensions and extensions in the reflective tower, *Proc. ACM Conf. on LISP and Functional Programming*, 327–341 (1988)

[4] Futatsugi, K., Trends in formal specification methods based on algebraic specification techniques — from abstract data types to software processes: a personal perspective —, *Proc. InfoJapan '90*, 59–66 (1990)

[5] Maes, P., Issues in computational reflection, in: Maes, P. and Nardi, D. Ed.: *Meta-Level Architectures and Reflection*, 21—35, North-Holland (1988)

[6] O'Donnel, M.J., *Equational Logic as a Programming Language*, MIT Press (1985)

[7] Olthoff, W. and Kempf, J., An algebraic specification of method combination for the Common Lisp Object System, *Lisp and Symbolic Computation*, 2, 115–152 (1989)

[8] Smith, B.C., Reflection and semantics in Lisp, *Conf. Rec. 11th ACM Symp. on Principles of Programming Languages*, 23–35 (1984)

[9] Sugano, H., Reflective computation in logic language and its semantics, *Report IEICE Japan*, COMP89-96, 67–76 (1989)

[10] Tanaka, T., Actor-based reflection without meta-objects, *IBM Technical Report*, RT-0047 (1990)

[11] Tanaka, J., An experimental reflective programming system written in GHC, *J. Information Processing*, 14, 1 (1991)

[12] Wand, M. and Friedman, D.P., The mystery of the tower revealed: a non-reflective description of the reflective tower, in: Maes, P. and Nardi, D. Ed.: *Meta-Level Architectures and Reflection*, 111–134, North-Holland (1988)

[13] Watanabe, T. and Yonezawa, A., An actor-based metalevel architecture for group-wide reflection, *Proc. REX/FOOL*, Lecture Notes in Comp. Sci. (1991), to appear.

[14] Yokote, Y., Teraoka, F. and Tokoro, M., A reflective architecture for an object-oriented distributed operating system, *Proc. ECOOP '89*, 89–106 (1989)