プログラミング 2－9
（1995. 8. 24）

# 並列オブジェクト指向言語 ABCL/$f$ のメタレベルアーキテクチャ

増原英彦* 松岡聡† 米澤明憲

E-mail: {masuhara, matsu, yonezawa}@is.s.u-tokyo.ac.jp
東京大学大学院理学系研究科情報科学専攻
〒113 東京都文京区本郷 7-3-1

並列アプリケーションのための機能拡張や最適化を行う手段として、自己反映計算によるメタレベルプログラミングが有効であることが認められつつあるが、実際のアプリケーションに応用した場合の有効性は、メタアーキテクチャの設計に大きく左右される。現在我々は、並列オブジェクト指向言語 ABCL/$f$ のメタアーキテクチャを設計している。特徴は、メタインタプリタ・メタオブジェクトによる拡張、annotation によるメタレベルへの指示、継承によるメタプログラムの再利用などである。本論文では、いくつかの並列プログラムにおける機能拡張の例を挙げ、それらがどのように記述されるかを検討することで、メタアーキテクチャの有効性を確かめる。

# Meta-Level Architecture of ABCL/$f$ and its Use in Parallel Programs

Hidehiko Masuhara* Satoshi Matsuoka† Akinori Yonezawa

Department of Information Science, University of Tokyo

Meta-level programming via computational reflection has come to be recognized as beneficial for parallel applications. Whether we can clearly program practical meta-programs greatly depends on the design of the language's meta-architecture. This paper presents a design of the meta-architecture of ABCL/$f$, an object-oriented concurrent reflective language. Its features are customization via the meta-interpreters and the meta-objects, annotations that serve as directives that are implemented by the meta-programs, re-use via inheritance in meta-programs, etc. The effectiveness of the architecture is examined through examples from several parallel programs.

*日本学術振興会特別研究員 (JSPS Research Fellow),
†工学系研究科情報工学専攻 (Department of Information Engineering)

## 1 並列言語とメタレベルアーキテクチャ

並列・分散言語における自己反映計算 (reflection) は、言語の機能拡張や最適化といったメタなプログラミングを、本来のアプリケーションから分離して記述できるといった理由から注目を集めている [6, 11]。超並列計算機上のアプリケーションのように、実行性能が重視される分野においても、並列化のための構文の拡張や細かな最適化が重要であるため、こういった自己反映計算によるメタプログラミングは有効であると思われる。そこで我々は、言語 ABCL/$f$ [9] に、自己反映計算の機構を導入することを目指し、そのメタレベルアーキテクチャを設計している。設計目標の一つは、実際的な並列アプリケーションを記述や最適化をする上で不足する機能をメタプログラミングによって実現することである。

一般に自己反映計算によるメタレベルプログラミングを行う場合、メタレベルの構成 — いわゆるメタレベルアーキテクチャ — が、プログラムの簡潔さや再利用性を大きく左右する。CLOS メタオブジェクトプロトコル [1] によって示されたように、オブジェクト指向による設計が有用であることは知られているが、個々の処理系が実際にどのようなプロトコル[1]を提供するかという点に関しては、処理系毎に考える必要がある。そのため、処理系が目指す拡張性を、実例に即して検討しながらプロトコルを定めることが現実的である。

本稿では、ABCL/$f$のメタアーキテクチャ設計について述べる。いくつかの並列プログラム例において、どのようなメタレベルの機能が必要か、あるいは望ましいかを検討する。そして、それらが我々のメタアーキテクチャによって無理なく記述できるか、再利性があるかといった点について考察する。

また、並列アプリケーションの最適化といった目標がある場合、効率のよい処理系が作成できることは重要である。そのため、ABCL/$f$のメタアーキテクチャは、(1) 解釈実行のオーバーヘッドを除去する、部分計算を利用したコンパイル [2] と、(2) 単に拡張性を追求するのではなく、効率的な実現を配慮した設計によって、この問題を解決する。

以下では、まず 2 節で言語 ABCL/$f$ (のメタレベル以外) を簡単に紹介する。次に 3 節で並列プログラムの例とそこに必要となる (あるいは望ましい) 問題とメタレベルの機能を検討する。4 節で、我々のメタアーキテクチャ設計を概要を紹介し、検討した問題が ABCL/$f$ でどのように記述されるかを 5 節で示す。最後に実行効率に関する議論と、関連研究に触れる。

[1]メタレベルにあるオブジェクトがどのように定義され、各メソッドの機能を指定するものをメタオブジェクトプロトコルと呼ぶ。

## 2 言語 ABCL/$f$

本論文で対象とした並列オブジェクト指向言語 ABCL/$f$ [9] の、メタアーキテクチャ以外の部分を簡単に紹介する。ABCL/$f$は、future を基本とした並列化の機能と、オブジェクトのメソッド起動による排他制御を行う言語で、現在までに高並列計算機上で稼動する処理系が作成されている。

並列実行の方法としては、関数呼出 (あるいはメソッド呼出) を `(future (f x))` のように記述する[2]。このときの式の値として reply box が返され、呼出側のその後の計算が、`f` と並行して実行される。関数 (メソッド) `f` の返値は、呼出側の reply box にセットされる。また、明示的に reply box へ値をセットするには `(reply val r)` という形式を用いる。セットされた値は、`(touch r)` によって読み出すことができる。この際、まだ値がセットされていない場合は、セットされるまで待つ (つまり同期をとる)。

オブジェクトは、変更可能な (mutable) データ構造と一連のメソッド定義からなる。このオブジェクトが持つデータ (状態変数) に対するアクセスは、メソッドを通して行われる。さらに、メソッド単位での排他制御があるため、データの一貫性を保つことを容易にする。(1) クラス定義、(2) メソッド定義、(3) メソッド起動はそれぞれ以下のような構文[3]になっている。

(1)(**defclass** ⟨class name⟩ (⟨superclass⟩)
⟨state vars⟩...)
(2)(**defmethod** ⟨class name⟩ ⟨method name⟩
(⟨args⟩...) ⟨body⟩)
(3)(⟨method name⟩ ⟨target⟩ ⟨args⟩ ...)

メソッド起動では、⟨target⟩ のクラスよってコードが選択される。分散メモリ環境において

[2]並列化以外の文法の大部分は Common Lisp と同じものである。

[3]現在の ABCL/$f$には継承機構はないが、ここでは単純なものが導入されていると仮定する。また、一部本来の ABCL/$f$とは異なる記法をとっている。

は、⟨target⟩ が存在するプロセッサ上での実行になる。

## 3 並列プログラム例とその問題点

この節では、並列プログラム例をいくつか示し、そこで必要となる (あるいは望まれる) 言語のメタレベル機能について検討する。

### 3.1 オブジェクトの移動・複製

分散メモリ環境におけるプログラムでは、遠隔メッセージによるオーバーヘッドや遅延が効率に大きく影響する。ある計算の最中に、遠隔プロセッサ上の同じオブジェクトを頻繁にアクセスするような場合には、そのオブジェクトを同一のプロセッサへ**移動 (migrate)** したり複製を作成することによる効率改善の方法が知られている。

例えば、以下のベクトル v1, v2 の内積の計算をする関数を考える。 v1, v2 が遠隔プロセッサ上にある場合、メソッド呼出 nth-element は遠隔メッセージとなる。

```
(defun product (v1 v2)
  (let ((sum 0.0) (size (size-of v1))
    (dotimes (i size)
      (setf sum
        (+ sum (* (nth-element v1 i)
                  (nth-element v2 i)))))
    sum)))
```

このような場合、関数 product の実行の間だけ、v1, v2 の複製を計算中のプロセッサ上に作り、その複製にアクセスすることで効率の改善が図れる。

実際にプログラム中でオブジェクトの移動や複製を利用するには、(1) 移動させるための機構、(2) 移動を指示する構文の拡張、(3) 移動する際の戦略、の 3 点について考える必要がある。

オブジェクトの複製を作る (移動させる) ために、クラス毎に複製メソッドを定義するは不可能ではないが労力が大きいため現実的でない。また、(a) 排他制御の方針 (b) 複製に対する状態変更の方針 (禁止する、書き戻すなど) (c) 恒久的な移動か、一時的な移動か、等いろいろな方針があり得る。そのため、**オブジェクトの排他制御の方針を拡張できることやオブジェクトの状態やメッセージキューをデータとして扱うことで、複製の管理をプログラム可能にすることが望ましい。**

次に、オブジェクトの移動・複製の機能が提供されたとしても、それを利用するためにプログラムを大幅に書き換えるのでは、プログラムの可読性・再利用性の点から好ましくない。むしろ、ある式の中に出現するオブジェクトを移動させる指示を、**annotation** によって本来のアルゴリズムとは別に記述できることが望ましい。

また、移動・複製の戦略 (どういう場合に移動をさせるか) は、アプリケーションの性質や実行環境に依存する。例えば、一定の大きさより小さなベクトルの場合には、複製をせずに実行するような戦略である。ここでは戦略の内容については議論せず、プログラマが適当な条件を与えることを仮定する。そこで、移動を指示する annotation にはそのような条件式が書けることも望まれる。

### 3.2 遅延隠蔽

遠隔プロセッサ上へのデータのアクセスの際に生じる通信遅延を隠すことは重要な最適化技術のひとつである。

ABCL/$f$ では、future/touch によって比較的容易に遅延隠蔽が行えるのだが、それでもプログラムの変更は小さくない。例えば、先の内積計算で、要素のアクセスをループ 1 回分先行させたプログラムは次のようになる。

```
(defun product (v1 v2)
  (let ((sum 0.0)
        (size (size-of v1))
        (e1a (future (nth-element v1 0)))
        (e2a (future (nth-element v2 0)))
        (e1b (future (nth-element v1 1)))
        (e2b (future (nth-element v2 1))))
    (dotimes (i size)
      (setf sum (+ sum (* (touch e1a)
                          (touch e2a))))
      (setf e1a e1b
            e2a e2b
            e1b (future
                  (nth-element v1 (+ i 2))
            e2b (future
                  (nth-element v2 (+ i 2))))))
    sum))
```

ここでは {advanced exp ...} という annotation を用意し、先行して要求するメッセージとそのタイミングを明示的に記述させることにする。この annotation は、

- 式 exp の中に現われるメソッド呼出を、future 呼び出しとしてあらかじめ実行。
- 続く計算で annotation 中と同じメソッド呼び出しがあった場合、先に行われた future 呼出で得た reply box に touch する。

という効果を生むものとする。これを用いて、内積の計算の遅延隠蔽は次のように書ける。

```
(defun product (v1 v2)
  (let ((sum 0.0) (size (size-of v1)))
    (dotimes (i size)
      (setf sum (+ sum (* (nth-element v1 i)
                          (nth-element v2 i))))
      (advanced (nth-element v1 (+ i 2))
                (nth-element v2 (+ i 2))))
    (advanced
      (nth-element v1 0) (nth-element v2 0)
      (nth-element v1 1) (nth-element v2 1))
    sum))
```

### 3.3 終了判定

並列探索問題のようなアプリケーションでは、プログラム実行が非同期的な沢山のスレッドから成るため、計算の終了を知ることが難しい。単純には、計算を並列に起動した(以下 fork したいう)箇所において、各計算の終了をメッセージによって通知する方法がとられる。それ以外にも、重みを用いる方法 [7] や、ハードウェアを使う方法などが考えられる。

こういった終了判定のアルゴリズムは、探索のアルゴリズムと独立している。従って、プログラム上も両者が独立した記述が行えることが望ましい。

一方、このような終了判定を行うには、終了時の返答先や重みなどの情報を受け渡す必要がある。そのためには、**関数やメソッド呼出に際してベースレベルとは別に引数を受け渡す**ことが必要となる。

メッセージや重みを用いるような終了判定を行う場合には、(ハードウェアなどで行われる場合を除いて)通常の計算よりもコストがかかるため、プログラム中に「終了判定を行う fork」を明示的に記述し、それらのみを終了判定の対象とすることが妥当である。ここでは、future 呼出を拡張した形式として、fork 形式を定義する。基本的には future 呼出のように並行実行するが、終了判定を自動的にする点が異なる。構文は以下の通りで、それぞれ開始および途中での fork に用いる。

(init-fork ⟨メソッド / 関数呼出形式⟩)
(fork ⟨メソッド / 関数呼出形式⟩)

これらを使って、例えば $N$-Queens 問題は以下のように記述される。

```
;;; function definition
(defun n-queens (size col rows)
  (if (= size col)
    (print *console-object* rows)
    (dotimes (row size)
      (if (not-checked? size col rows row)
        (fork (n-queens size (1+ col)
                        (cons row rows)))))))
;;; the caller
(init-fork (n-queens size 0 '()))
```

### 3.4 スケジューリング

並列探索問題でも、特に最適解を探す問題においては、解の予測値を使って枝刈りや探索の順序を制御する A*- 探索という手法が知られている。

特別な言語機構を使わずにこのような制御を行うには、(1) ユーザ定義のスケジューラを作り、(2) 関数(あるいはメソッド)を fork するかわりにスケジューラに実行要求を送る、(3) 実行が終了したらスケジューラに通知する (4) 通知を受けたスケジューラが優先度の高い実行要求を選んでそれを起動する、といったプログラムになる。また、プロセッサ間の負荷分散をスケジューラどうしの通信で実現する。結果として、通常の探索プログラム(例えば 3.3 節のもの)と大きく異なったスタイルになる。

むしろ、通常の探索プログラムとして書いたものに、予測値などの情報を加えるだけで、スケジューラとの通信などは明示的に記述しないで済ます機構が望ましい。

KL1[10] のような、スケジューリングの優先度を指示できる処理系では、このような制御がある程度記述できる。しかし、実際には負荷分散や枝刈りといったことを行うため、スケジューラ自身がユーザによって定義できることが望ましい。

## 4 ABCL/$f$のメタアーキテクチャ

ABCL/$f$のメタアーキテクチャは、メタインタプリタによる言語拡張と、メタオブジェクトによる実行時システムの拡張の両方の機能を備えている。特徴的なこととして、

- 状態やメッセージが一級 (first class) のデータとして扱える。
- "メタ引数" によって、メタレベル間での情報のやりとりを、ベースレベルから独立して行える。
- Annotation によってメタレベルへの指示をベースプログラムから独立して記述できる。

といった点が挙げられる。

**メタインタプリタによる拡張** ABCL/$f$ のメタインタプリタは、基本的には 3-Lisp[8] などに見られ

るメタサーキュラインタプリタと同種の形式をしている。メタインタプリタは、クラス primary-eval のメソッド群として定義され、サブクラスを定義することで拡張できる。プログラマの変更を少なくするため、インタプリタの定義は複数のメソッドに細分化され、継承機構によって再利用できる。

以下に、メタインタプリタのプロトコルを示す[4]。

**(eval-entry exp env):** メソッド(関数)の開始時に呼ばれる。メソッド本体は exp に、引数や状態変数は env に束縛されている。実際の式の評価にはメソッド eval を呼び出す。

**(eval exp env):** 式 exp を環境 env の下で評価する。式の形によって、eval-const, eval-var などのメソッドを呼び出す。

**(eval-variable var env):** 変数 var の値を評価する。

**(eval-methodcall exp env):** メソッド呼出形式を評価する。内部では (1) 呼出のタイプ(future 等)を決め、(2) 受信者と引数を eval によって求め、(3) meta-args によって「メタ引数(後述)」を求め、(4) do-method-call によってメッセージを送る。

**(do-method-call type target method args meta-args env):** 実際にメソッド呼出をする。具体的には、make-message によってメッセージデータを作り、target のメタオブジェクトに message メソッドによってメッセージを送る。

**メタ引数**とは、関数やメソッド呼出の際に、通常の引数に加えてメタレベルで渡される引数のことである。呼出側は名前と値のリストによってメタ引数を指定し、被呼出側では環境から値をとり出すことができる。これによって、メタプログラム間での情報の受け渡しを、ベースレベルから独立して行える。

式に対する **annotation** は、式の後ろに "{ }" に囲まれて記述される。この情報は、メタレベルでは式 (exp) に付随するデータとしてアクセスできる。

**メタオブジェクトを通した実行時システムの拡張**

ABCL/R[11] や ABCL/R2[3] と同様、個々のオブジェクトのメタレベル表現を持つメタオブジェクトが存在する。メタオブジェクトは、クラス metaobject のオブジェクトとして定義され、メッセージ受信や排他制御、状態の管理などを行っている。クラス metaobject の状態変数には、 class, methods, state-vars(状態変数の名前と値), message-queue, evaluator, mode(オブジェクトの排他制御に関する状態) などがある。メソッドのプロトコルは以下のようになっている。

**(message m):** メッセージ m の受信。通常は排他制御のためにメッセージキューに保存される。実際にメソッドを実行するのは次のメソッド accept である。

**(accept message):** メソッドの起動。オブジェクトの状態を "active" にし、message 中のメソッド名から適切なメソッド定義を選び、メタインタプリタ eval-entry を呼び出す。

**(become env):** メソッド実行の終了時に呼び出されるメソッドで、env 中から状態変数に対して行われた更新を、実際の状態変数に反映させる。

**(process-next):** メッセージキューを調べ、メッセージがあればそれを accept する。そうでない場合は、状態を "dormant" にする。

## 5 メタレベルプログラミングの実際

3 節に示した問題が、ABCL/$f$ のメタアーキテクチャによってどのように解決されるかを示す。

### 5.1 オブジェクトの移動

**移動のための機構** メタレベルでは、オブジェクトの状態変数は state-vars というメタオブジェクトの状態変数としてアクセスできるため、プロセッサ p にオブジェクトの複製を作るメソッドは以下のように比較的単純に定義できる[5]。

```
(defmethod metaobject
           copy-object (p &reply-to r)
  (future
    (make-replica-meta
      :class class :state-vars state-vars
      :original self)
    :on p :reply-to r))
```

ここでクラス replica-meta は、複製されたオブジェクトのためのメタクラスであり、変更された状態をオリジナルのオブジェクトに書き戻すためのメソッドなどが定義されている。

[4]実際のメソッド呼出では、第 1 引数が eval オブジェクトになるが、ここの記述では略してある。

[5]"&reply-to r" は reply box を明示的に変数 r に束縛する指示である。また future 呼出中の ":on p" はオブジェクトを生成するプロセッサを指定している。

移動されたオブジェクトの排他制御は、メタオブジェクトのメソッド message, accept を再定義することによって実現できるが、ここでは割合する。

**移動を指示するための構文**　ここでは移動を指示するための構文として、{cache (⟨変数...⟩) :when ⟨条件式⟩ } という annotation を用意する。これは、条件式が成立した場合に、指定された変数に束縛されるオブジェクトの複製を作り、式の中におけるそのオブジェクトのアクセスはその複製に対して行うものである。先のベクトルの内積の例では、以下のように使われる。

```
(dotimes (i size)
  (setf sum
    (+ sum (* (nth-element v1 i)
              (nth-element v2 i)))))
{cache (v1 v2) :when (size < 20)}
```

この annotation を解釈は、クラス cache-eval のメソッド eval として図 1 のように定義される。ここでは簡単のために、移動の対象を示す変数は 1 つしか書けないとした。継承機構によって、簡潔に定義できている。

### 5.2 遅延隠蔽

遅延隠蔽の実現には、適切なタイミングでメッセージを先行して送信することと、実際に値を使う場面で先行して送ったメッセージの結果を利用することの 2 点つに分かれる。

先行してメッセージを送るタイミングに関しては、⟨式⟩{advanced exp...} という構文によって指定される。Annotation の引数に現われる式 exp の解釈は、通常の計算と区別するためにフラグを立てておいてから eval によって評価するだけである。

```
(defmethod LH-eval eval (exp env)
  (if (advanced-annotation? exp)
    (let ((a-exp (annotation-expression exp)))
      (set-in-advanced-evaluation env)
      ;; evaluate exp in the annotation
      (eval a-exp env)
      (reset-in-advanced-evaluation env)))
  (eval super exp env)) ; evaluate the body
```

一方、メソッド呼出の解釈は、状況によって 3 種類に分かれる。

1. Annotation の中だった場合は、future 型のメソッド呼出を行い、返値の reply box と、メソッド名と引数を環境に記録しておく。

2. 普通 (annotation の中でない) の式の中だった場合で、メソッド名と引数の組み合わせが、環境の中に記録されていた場合、記録されている reply box に touch する。

3. 普通の式の中だった場合で、メソッド名と引数の組み合わせが、環境の中に記録されていない場合、普通にメソッド呼出をする。

実際のメソッド呼出は、do-method-call が行う。これが以下のように変更される。

```
(defmethod LH-eval do-method-call
    (type target method args meta-args env)
  (if (in-advanced-annotation? env)
   (let ((rbox                          ; case 1
          (do-method-call
            super 'future target
            method args meta-args env)))
     (remember-method env
      type target method args meta-args rbox)
     nil)
   (let ((rbox                  ; case 2 or 3
          (look-for-method env
           type target method args meta-args)))
     (if rbox
       (touch rbox)                     ; case 2
       (do-method-call
         super type target method
         args meta-args env)))))        ; case 3
```

### 5.3 終了判定

ここでは終了メッセージを回収する方式と、重みを使う方式の 2 つの方式を考える。両者に共通する計算パターンとして、(1) 関数 (メソッド) 起動時に、reply box や重みを受け取る。(2) fork の際に、新たに reply box を作ったり、重みを分割したりして、(3) それを fork されるスレッドに渡す。(4) 関数 (メソッド) 終了時に、fork したスレッドの終了を待ったり、重みを返却したりする、といった点である。

(1), (3) に関しては、メタレベルでの引数の受け渡しの機能を用いることで、また (2) は eval の、(4) は eval-entry の変更で、それぞれ実現できる。

ここでは、2 つの終了判定方式に共通する定義と、終了メッセージを回収する方式の定義のみを示す。まず、共通する部分は新たに導入した fork/init-fork の解釈を与えるが、基本的には future 呼出と同様に扱う。

```
(defclass TD-eval (primary-eval))
(defmethod eval TD-eval (exp env)
 (cond ((exp-fork? exp)
        (eval-fork self exp env))
       ((exp-init-fork? exp)
        (eval-init-fork self exp env))
       (t (eval super exp env))))
(defmethod eval-fork TD-eval (exp env)
  (eval-method-call exp env))
(defmethod eval-init-fork TD-eval (exp env)
  (eval-method-call exp env))
```

```
(defmethod cache-eval eval (exp env)
  (if (cache-annotation? exp)                                    ; (1)
    (if (eval (cache-annotation-condition exp) env)              ; (2)
      (let* ((obj (lookup (cache-annotation-target exp) env))
             (copy (copy-object-request (meta obj)))             ; (3)
             (new-env (extend-env env target copy)))             ; (4)
        (let ((result (eval super exp new-env)))                 ; (5)
          (copy-back (meta copy))                                ; (6)
          result))
      (eval super exp env)) ; eval as usual
    (eval super exp env)))  ; eval as usual
```

(1) は annotation の有無を検査し、(2) は条件式の評価をする。(3) でオブジェクトの複製を行い ((meta obj) はメタオブジェクトを得る形式)、それを (4) で環境にセットした後に (5) で式本体を評価する。(6) では、式の評価終了後に、変更された状態を書き戻す指示。変数 self, super は、それぞれ eval オブジェクト自身とスーパークラスのメソッド起動を行うための疑変数である。

図 1: オブジェクト移動の annotation の解釈

終了メッセージ回収方式の定義は、クラス ack-eval のメソッドになる。このクラスは終了判定共通のクラス TD-eval のサブクラスである。ここで定義される動作は、(a) メソッド評価の入口で、rboxes という変数を用意する。(b) メソッド評価の最後に、rboxes の全要素に touch して、その後 (c) 自身に渡された reply box に返答を送る。(d) fork 形式の実行のときに、reply box を作り、(e) rboxes に記録し、(f) メタ引数として渡す。

```
(defclass ack-eval (TD-eval))
(defmethod ack-eval eval-entry (exp env)
  (let* ((new-env (extend-env env      ; a
                    :meta 'rboxes '()))
         (result
          (eval-entry super exp new-env)))
    (dolist (rbox (lookup-meta
                   'rboxes new-env))
      (touch rbox))                    ; b
    (reply (lookup-meta env 'ack) t)   ; c
    result))
(defmethod ack-eval meta-args
     (type method target args options env)
  (let ((margs (meta-args super type method
                 target args options env)))
    (cond ((eq type 'fork)
           (let ((rbox (make-reply-box))) ; d
             (push-meta-env env          ; e
                            'rboxes rbox)
             (list* 'ack rbox margs))) ; f
          (t margs)))) ; for the other forms
```

実際のプログラムでは、この終了判定機能を利用する関数・メソッドを annotation によって明示する。それ以外の関数・メソッドにはこれらの evaluator は関与しない。

### 5.4 スケジューリング

スケジューリングのためには、メソッド (関数) 呼出形式のところで、メッセージを送る代わりに fork される計算の情報をスケジューラに送る必要がある。そのためには、do-method-call でメタオブジェクトに送るメッセージをスケジューラに送るように変更すればよい。

```
(defmethod sched-eval do-method-call
     (type target method args meta-args env)
  (let ((m (make-message
            target method args meta-args)))
    (request *scheduler*
             (cons m (get-priority env)))))
```

また、メソッド (関数) 終了時にスケジューラへの通知を送る必要があるが、これには eval-entry を変更する。

```
(defmethod sched-eval eval-entry (exp env)
  (eval-entry super exp env)
  (run-next *scheduler*))
```

## 6 議論

多くの言語処理系は、処理系の実現方式に関わる制御を部分的にプログラマに公開している。こういったメタな機能を利用すれば、本論文に示した問題が解決できることも多い。例えば、メッセージを一級 (first class) データとして扱うことができれば、3.4 節で述べたスケジューリングを行うことは比較的容易と思われる。

しかし、このようなメタな機能を利用するためには、プログラムの構造を大きく変更する[6]ことが必要となり、制御と本来のアルゴリズムの分離は実現できないことが多い。

他方、メタインタプリタの変更 (あるいはコンパイラの変更) が可能な言語処理系では、新たな構文を用意することが容易なので、メタな機能によって新しく導入された機能を自然な形で提供で

[6]例えば 3.2 節に示したような書き換えである。

きる。つまり、自己反映言語の利点は、(1) 新しい機能を作成することと、(2) その機能を自然な形で提供するという 2 点を、統一的に実現できることにあると言える。

**実行効率について**　本論文で示したようなメタプログラムの実行効率を良くすることは、自己反映言語の重要な課題の一つである。まず、evaluator オブジェクトへの変更に対しては、すでに部分計算によるコンパイル方式を提案している [2]。これによって、例えば図 1のメタインタプリタと内積のプログラムから、以下の様なコードを得ることができ、解釈実行オーバーヘッドのない効率を得ている。

```
(let ((t1 (copy-object-request (meta v1)))
      (t2 (copy-object-request (meta v2))))
  (dotimes (i size)
    (setf sum
      (+ sum (* (nth-element t1 i)
                (nth-element t2 i)))))
  (copy-back (meta t1)) (copy-back (meta t2)))
```

一方、メタオブジェクトの拡張に関しては、多くが実行時の操作に依存しているため、個々の操作の効率的な実現を考えてゆくしかない。ただし、本論文で示したような、メッセージや状態変数を扱うといった操作の場合は、遠隔メッセージのためのコード化のルーチンを利用するなどの方法により効率的な実現が充分可能であると思われる。

**関連研究**　AL-1/D は分散環境のための自己反映言語で、multi-model reflection framework (MMRF)[6] に基いており、記述力の高いメタアーキテクチャを作成すること目指している。また、AL-1/D 処理系を用いて、オブジェクトの移動に関する考察と実験を行っている [5] が、比較的ネットワーク遅延の大きな環境を前提としており、超並列計算機などの環境での効率化には課題が多いと思われる。

CodA は、Smalltalk のメタアーキテクチャであり、部品化のために、オブジェクトによって細分化されたメタアーキテクチャを持つ [4]。特に分散 framework の提供などを目的しているが、効率的な実現方式までは示されていない。

## 7　まとめ

本論文では、言語 ABCL/$f$ のメタレベルプログミングを、いくつかの例題に即して検討した。メタインタプリタ・メタオブジェクトによる拡張、annotation によるメタレベルへの指示、継承による差分プログミングなどによって、簡潔で再利性の高いメタプログラミングが達成されている。今後は、プロトコルのさらなる強化、メタオブジェクトの効率的な実現方式、実際の処理系作成を行う。

**謝辞**　日頃から活発な議論をしていただいている米澤研究室の諸氏に感謝いたします。

## 参考文献

[1] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[2] H. Masuhara, S. Matsuoka, K. Asai, and A. Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In *Proc. of OOPSLA'95*, 1995. to appear.

[3] S. Matsuoka, T. Watanabe, and A. Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *Proc. of ECOOP'91*, 1991.

[4] J. McAffer. Meta-level programming with CodA. In *Proc. of ECOOP'95*, 1995. to appear.

[5] H. Okamura and Y. Ishikawa. Object location control using meta-level programming. In *Proc. of ECOOP'94* (LNCS 821), pp. 299–319. July 1994.

[6] H. Okamura, Y. Ishikawa, and M. Tokoro. AL-1/D: Distributed programming system with multi-model reflection framework. In *Proc. of IMSA Workshop on Reflection and Meta-Level Architecture*, pp. 36–47, Nov. 1992.

[7] K. Rokusawa, N. Ichiyoshi, T. Chikayama, and H. Nakashima. An efficient termination detection and abortion algorithm for distributed processing systems. In *International Conference on Parallel Processing* (I), pp. 18–22, 1988. also published as ICOT-TR 341.

[8] B. C. Smith. Reflection and semantics in Lisp. In *Proc. of POPL'84*, pp. 23–35, 1984.

[9] K. Taura, S. Matsuoka, and A. Yonezawa. ABCL/$f$: A future-based polymorphic typed concurrent object-oriented language – its design and implementation –. In *Proc. of DIMACS workshop on Specification of Parallel Algorithms*, 1994.

[10] K. Ueda and T. Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, 33(6):494–500, 1990.

[11] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *Proc. of OOPSLA'88*, pp. 306–315. 1988.