並列自己反映言語システムの部分計算によるコンパイル技法

増原英彦[†] 松岡 聡[††] 米澤明憲[†††]

並列自己反映言語システムは，並列アプリケーションの最適化等を簡潔に記述するメタプログラミングの機能を持つ一方，解釈実行に基づくモデルから来る効率上の問題を持つ．本論文では部分計算を用いた並列自己反映言語のコンパイル技法を提案する．この技法では，副作用について拡張された部分計算やプログラム変換を適用することで基本的に解釈実行を除去し，直接実行のみとする．並列計算機上の実験では，並列アプリケーションのメタレベルに記述された最適化が，7～17%のオーバヘッドで実行できるという結果が得られている．

A Compilation Technique for Parallel Reflective Language Systems Using Partial Evaluation

HIDEHIKO MASUHARA,[†] SATOSHI MATSUOKA[††]
and AKINORI YONEZAWA[†††]

Meta-programmability of parallel reflective language systems is beneficial for parallel applications to describe optimizations, etc. On the other hand, their execution model based on interpretation is an obstacle to efficient implementation. We propose a compilation technique for parallel reflective languages using partial evaluation. The technique, which effectively eliminates program interpretation, includes partial evaluation extended for side-effects, and several program transformation techniques. Benchmarks on a MPP show that parallel applications with meta-level optimizations can be executed with small overhead.

1. はじめに

超並列計算機・共有メモリ型並列計算機・ワークステーションクラスタなどで，大規模かつ不規則な並列アプリケーションを実行するため，並列オブジェクト指向言語によるプログラミングが注目されている．しかし，その効率的な実行のためには，負荷分散・データ配置・スケジューリングといった，**メタな制御**を実行計算機のアーキテクチャ・アプリケーション・実行時データの性質等に応じて適切に行う必要がある．

従来このような制御は，主にアプリケーションプログラム中に制御のためのコードを混在させることで実現されており，(1) プログラムの可読性が悪くなる，(2) プログラムの再利用性が悪くなるといった問題が生じていた．例として，並列オブジェクト指向言語 ABCL/f[1] で記述された探索問題のアプリケーションに並列実行のための制御を加えた場合を示す．**図1** では，探索木の節点を並列オブジェクト（**search-task**）として生成し，それらを非同期メソッド呼出しする（**(past (do-search ...))**）ことで並列に探索する．これに対して**図2** は，(1) リモート通信を減らすために探索木の深い部分では同一プロセッサにオブジェクトを生成し，(2) 重みを使った終了判定する，という制御を

```
(defclass search-task () ...)
(defmethod search-task do-search (p1...pn)
  (if 〈答えを見つけたか?〉
    〈答えを返す〉
    (while 〈子供を生成できる間〉
      ;; 子供のためのパラメータ生成
      (let ((q1 ...) ... (qn ...))
        ;; 子オブジェクトを生成・非同期
        ;; メッセージによって並列に探索を実行
        (past (do-search (new 'search-task)
                q1 q2 ... qn))))))
```

図1 並列言語による単純な探索問題の記述
Fig. 1 Simple parallel search problem in concurrent object-oriented language.

† 東京大学教養学部情報・図形科学教室
Department of Graphics and Computer Science, College of Arts and Sciences, University of Tokyo
†† 東京大学大学院工学系研究科情報工学専攻
Department of Information Engineering, University of Tokyo
††† 東京大学大学院理学系研究科情報科学専攻
Department of Information Science, University of Tokyo

```
(defmethod search-task do-search (p1 ... pn depth weight)          ; 引数に深さと重みが加わる
  (if 〈答えを見つけたか?〉
    (progn 〈答えを返す〉 (return-weight term-master weight))      ; 重みを返す
    (let ((next-depth (1+ depth))
          (weight-for-child (max 1 (/ weight max-num-children))))  ; 子供に分配する重みを計算
      (while 〈子供を生成できる間〉
        (let ((q1 ...) ... (qn ...))                               ; 子供のためのパラメータ生成
          (if (< weight weight-for-child)                          ; 重みが足りるかチェック
            (setq weight (request-weight term-master))             ; 足りない場合は要求を送る
            (setq weight-for-child (/ weight ave-num-children)))
          (past (do-search (new 'search-task      ; ↓オブジェクト生成プロセッサの指定
                                :on (if (< *threshold* depth) (this-node-id) (random-node-id)))
                           q1 q2 ... qn next-depth weight-for-child)) ; 深さと重みを一緒に送る
          (setq weight (- weight weight-for-child))))              ; 子に与えた分の重みを減らす
      (if (< 0 weight) (return-weight term-master weight))))))     ; 重みを返す
```

図2 局所性の制御と重み付き終了判定のコードを図1に埋め込んだ様子
Fig. 2 Parallel search problem with locality control and weighted termination detection.

アプリケーションコード中に埋め込んだものであり，プログラムが複雑化している様子が見てとれる．

この問題に対して，High Performance Fortran (HPF) の指示子 (directives) のような試みがある．これは，annotation として配列のデータ配置等に関する指示を記述することで，アプリケーションから分離したメタな記述を可能にしている．しかしHPFの記述は処理系が提供する配置方式からの選択であり，annotation 自身の適切な抽象化手段も提供されていない．そのため，配置方法をモジュール化したり，新たなデータ配置方法をユーザが記述することは困難である．コンパイラ開発の負担も大きい．特に不規則な並列アプリケーションでは，動的な条件に応じた制御を行う必要があるため，アプリケーション本体からの指示はHPFのように宣言的に記述しつつも，制御記述そのものは**操作的 (operational)** に記述でき，拡張可能な形でライブラリ化できることが望ましい．

また，コンパイラの内部構造を拡張可能にすることで，メタな制御を実現する試みもある[2)~4)]が，メタな制御はコンパイル時のソースプログラムの変換やコード生成方法の変更として記述されるため，実行時に変化するような制御を記述することが難しい．むしろ，動的なメタプログラミングを直接行える方が，ユーザにとっては扱いやすい．

そのため，メタアーキテクチャを適切に設計した言語処理系を構築する必要がある．**自己反映 (reflective) プログラミング**は，そのような処理系構築の一手段である．

これまでにも，分散環境や並列環境を対象とした自己反映言語システムが提案されてきた[5)~9)]．これらのシステムは**解釈実行 (interpretation)** に基づくメタレベルのモデルをユーザに与えているため，メタな制御を容易に記述できる一方で，実行時のオーバヘッドが大きいという問題を持っている．そのため，メタな制御によって得られる性能向上が相殺されてしまうことが多い．この効率の問題を解決しない限り，並列アプリケーションにの性能向上に用いることはできないといえる．

我々は，超並列計算機等の種々のプラットフォーム上での実用的な並列計算が可能な，自己反映計算モデルに基いた並列オブジェクト指向言語 ABCL/R3 を開発している．特に本論文では解釈実行に基づくメタレベル記述を，**部分計算 (partial evaluation)** によってコンパイルする技法を提案し，効率の問題を解決する．ABCL/R3 の特徴は，

メタレベルの操作的記述による拡張

オブジェクトのメソッド実行は，メタレベルにある**評価器 (evaluator)** によって操作的に定義されている．この評価器は，委譲形式 (delegation) によって拡張することができる．これにより，アプリケーションから分離された形で種々の制御を記述できる．たとえば，前出の局所性制御・重みによる終了判定のような制御は，それぞれの制御を行う評価器オブジェクトを定義し，図1のベースレベルプログラムに与えることで実現できる．

部分計算コンパイルによる効率的実行

ユーザによって拡張された評価器を持つプログラムは，部分計算を用いた技法によってコンパイルされる．メタレベルの評価器のコードはベースレベル☆の

☆ 本来のアプリケーションが記述実行されるレベルをメタレベルに対してベースレベルと呼ぶ．

コードに関して特化され，結果としてメタレベルの変更・拡張がベースレベルに埋め込まれたようなコードが生成される．これにより，ユーザにはメタレベルが解釈実行しているモデルを与えつつも，実際には解釈実行によらない効率的な実行が可能になっている．

という点である．後者は，ベースレベルのメソッド単位の部分計算，副作用を扱う枠組みの提案，いくつかのプログラム変換といった技法によって可能になった．

並列アプリケーションの最適化に応用した場合の結果では，非自己反映言語の直接実行と比べて7%～17%のオーバヘッドしかかからなかった．従来のインタプリタに基づく処理系では，最適化されたものでも，非自己反映言語の約10倍遅かった[6]ことに比べ，我々の提案する実行方式が大きな拡張性と高い効率を同時に達成しているといえる．

2. 部分計算によるコンパイル

ABCL/R3システムにおける部分計算を用いたコンパイル技法について説明する．まず部分計算の基本的な仕組みと，それを自己反映言語のコンパイルに用いるアイデアについて述べ，次にABCL/R3システムに適用する際の問題点を述べる．

2.1 部分計算とは

部分計算とは，プログラムの入力の一部にあらかじめ値を与えて，その値に特化されたプログラムを生成することである[10),11)]．特化されたプログラムは，あらかじめ与えられた値に関する計算を終えているため，元のプログラムに同じデータを与えた場合よりも速度が向上している．簡単のために，2入力のプログラム $p(x,y)$ を考える．このプログラム p の第1引数 x に定数 c を与えて部分計算したプログラム（**residualプログラム**と呼ばれる）を $PE(p,c)=p_c$ と表す．このとき，プログラム p_c は $p_c(y)=p(c,y)$ を満たす．さらに，x の値に依存する計算や条件分岐をすでに終えているため，$p_c(y)$ の実行は $p(c,y)$ よりも高速になっている．

今回我々が使用した部分計算器は，部分計算器Fuseの研究[12)]で提案された技術である，グラフを用いたコード重複の回避や部分的に静的な（partially static）構造の扱いなどを踏襲している．

部分計算の過程は**図3**のような，インタプリタに類似した式の評価規則によって定義される．部分計算関数（$\mathcal{PE}$）は式と環境を受け取り，**記号値（symbolic value）**を返すものとして定義されている．記号値は，部分計算時に計算される（静的な）数値・真偽値・ペア

$$\mathcal{PE} : Exp \to Env \to Sval$$
$$Sval = Num + Bool + Pair + \cdots + Top$$

$$\mathcal{PE}[\![var]\!]\rho = \rho(var)$$
$$\mathcal{PE}[\![num]\!]\rho = Num(num)$$
$$\mathcal{PE}[\![(\texttt{if } e_1\ e_2\ e_3)]\!]\rho$$
$$= \text{case } (\mathcal{PE}[\![e_1]\!]\rho) \text{ of}$$
$$\quad Bool(\text{true}) : \mathcal{PE}[\![e_2]\!]\rho$$
$$\quad Bool(\text{false}) : \mathcal{PE}[\![e_3]\!]\rho$$
$$\quad\quad s : Top((\texttt{if } s\ \mathcal{PE}[\![e_2]\!]\rho\ \mathcal{PE}[\![e_3]\!]\rho))$$
$$\mathcal{PE}[\![(\texttt{let } ((v_1\ e_1)\cdots(v_n\ e_n))\ e)]\!]\rho$$
$$= \mathcal{PE}[\![e]\!]\rho[s_i/v_i] \quad \text{ただし } s_i = \mathcal{PE}[\![e_i]\!]\rho\ (i=1,\ldots,n)$$
$$\mathcal{PE}[\![(+\ e_1\ e_2)]\!]\rho$$
$$= \text{case } (\mathcal{PE}[\![e_1]\!]\rho, \mathcal{PE}[\![e_2]\!]\rho) \text{ of}$$
$$\quad (Num(n_1), Num(n_2)) : Num(n_1+n_2)$$
$$\quad\quad (s_1, s_2) : Top((+\ s_1\ s_2))$$

図3 部分計算の規則（一部）
Fig. 3 Rules for partial evaluation (abridged).

（consセル）などに加え，実行時に計算される（**動的**な）値（ここでは**Top値**と呼ぶ）から成る．Top値は，$Top(exp)$ のように書き，「式 exp を実行して得られる値」を表す．条件分岐や関数適用については，引数が静的な値の場合には実際に分岐や計算を行い，動的な値の場合は適切なコードを含んだTop値を生成する．このようにして最終的に得られた記号値から，プログラムを復元したものが部分計算の結果である（**residualコード**と呼ぶ）．

2.2 部分計算によるメタレベルのコンパイル

自己反映言語を，部分計算を用いてコンパイルする方法は，二村射影[10)]に基づいている☆．以下にそれを説明する．

インタプリタプログラム（int）は，ベースレベルのプログラム（$prog$）と実行時データ（$data$）の2つの入力をとり，結果（$result$）を返すようなプログラムであり，$int(prog,data)=result$ と書かれる．プログラム int を $prog$ に関して部分計算したプログラムは $PE(int,prog)=int_{prog}$ と書かれる．これは，(1)$int_{prog}(data)=int(prog,data)$，つまり，インタプリタ実行と同様の結果を与える．また，部分計算が十分効果的だった場合，(2) $int_{prog}(data)$ の実行は $int(prog,data)$ よりも高速化され，$prog$ を直接実行したものと同程度になっており，解釈実行のオーバヘッドを除去できる．

一方，自己反映言語は「変更が可能なインタプリタを持つ処理系」と見なせる．素朴に作成した処理系では，変更されたインタプリタを int' として，実際にインタプリタを走らせる，$int'(prog,data)$ という実

☆ 正確には3つある二村射影の第1番目である．

行をする．部分計算を用いれば，インタプリタをベースレベルプログラムに対して部分計算したプログラム $PE(int', prog) = int'_{prog}$ を自動的に生成することができる．このプログラムは，インタプリタに対する変更を反映しつつ，ベースプログラムを直接実行したときと同程度の速度が期待できるものになっている．

しかしながら，これは理想的な場合にすぎない．実際には，部分計算をそのまま適用することは困難である．以下，それに関して問題点と我々のとった解決策を述べる．

- **メタシステムが並列オブジェクトによって構成されている**：ABCL/R3 のメタレベルは，複数の並列オブジェクトから成る．前出の部分計算によるコンパイルの枠組みを直接あてはめると，部分計算する対象言語が，並列オブジェクト指向言語になるが，そのような部分計算は困難である．

解決策：メタレベルの評価器（evaluator）と，ベースレベルプログラムのメソッド本体部分に限定した部分計算を個別に行う．スケジューリング・メソッド選択機構・排他制御などは，部分計算の対象とせず☆，また，オブジェクト間のメッセージ送信などは，I/O 副作用と見なすことで，既存の部分計算の技術の多くが適用できるようになる．

- **プログラムが副作用を含む**：メソッド単位に分割した部分計算を行う場合，オブジェクトの生成やオブジェクト間の相互作用（メッセージ送受信）などの操作は副作用となる．また，ベースレベルプログラムにおける変数代入は，通常メタレベルの環境に対する破壊的変更（Lisp の **rplacd**）になる．これらの副作用は関数型言語の部分計算ではうまく扱えない☆☆．

解決策：オブジェクト間の相互作用については，**preaction による部分計算の拡張を提案する**．これによって，元のプログラムにおける副作用の順序や回数を保存するような部分計算結果を生成することが可能になる．また，変数代入に関しては，前処理でインタプリタを**状態渡しの形式に変換し，後処理（post-processing）で実際の代入形式を再構成**することで解決している．

- **メタシステムの定義が動的に変更可能**：自己反映システムでは，メタシステムの定義を実行時に変更することができる．このように，動的に変化するプログラムを部分計算することは事実上不可能である☆☆☆．

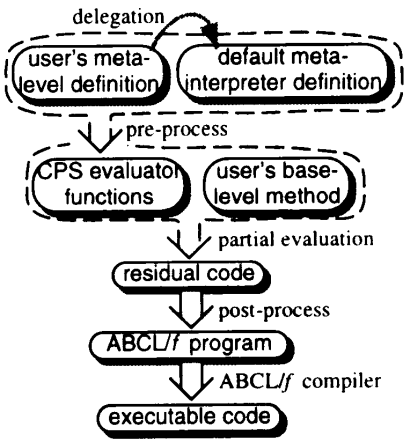


図 4 ABCL/R3 システムのコンパイル手順
Fig. 4 Overview of ABCL/R3 compiler.

解決策：コードバージョニングによる実現をする．実行時に使用される評価器オブジェクトについて，静的であるとして部分計算によるコンパイルを行っておく．さらに，実行時に決定される評価器オブジェクトに応じてコンパイルコードを選択する．

3. 部分計算コンパイル技法の実際

ABCL/R3 システムは，図 4 のような手順でコンパイルを行う．

(1) （前処理）メタレベルの評価器のメソッド定義を，継続渡し・状態渡しの関数定義に変換する．
(2) 変換された関数定義を，ベースレベルプログラムの各メソッド本体に関して部分計算をする．
(3) （後処理）部分計算の結果のコードから，代入形式を復元する．さらにメソッドインタフェースを加えて ABCL/*f* プログラムを生成する．
(4) ABCL/*f* コンパイラによってコンパイルし，実行形式を得る．

以下では，コンパイルの際に用いられる技法について説明する．

3.1 メソッド本体への部分計算の限定

ABCL/R3 システムのメタレベルでは，メソッドの解釈実行は複数の評価器オブジェクトの委譲形式（delegation）によって行われている．これらを部分計算するために，以下の前処理によって部分計算可能な関数群へと変換する．

(1) あるベースレベルメソッドについて，そのメソッドを実行する評価器オブジェクト群を決定する．

☆ 部分計算の対象を，これらの操作にまで拡大することも可能であると思われる．たとえば Open C++[13] や Self[14] において行っているメソッド選択機構のメタレベルに関する最適化は，部分計算の一種と見なせる．

☆☆ 手続型言語の部分計算[15)~17)]には，副作用を扱えるものもあるが，インタプリタをコンパイルできるほど強力であるかは不明である．

☆☆☆ 仮にできたとしても，その結果は元々のインタプリタと同じものになってしまうと予想される．

```
————————(a) 評価器の定義：変換前————————
;;; クラスeval-Aのメソッド定義
(defmethod eval-A eval (exp env)
  (print exp)
  ;; eval-Bのメソッドevalへの委譲
  (eval super exp env))
;;; クラスeval-Bのメソッド定義
(defmethod eval-B eval (exp env)
     ;; eval-Aのメソッドを再帰呼び出し
  .. (eval self ..) ..)
————————(b) 評価器の定義：変換後————————
(defun eval-A-eval (exp env cont store)
  (print exp)
  (eval-B-eval exp env cont store))
(defun eval-B-eval (exp env cont store)
  .. (eval-A-eval ..) ..)
```

図5 評価器メソッドへの前処理
Fig. 5 Preprocess for evaluator methods.

この評価器オブジェクト群は，静的に決定されているものとする．これらが動的に変更される場合の対処は3.4節で述べる．

(2) 各評価器オブジェクトのメソッド定義を一意（unique）な名前の関数定義に置き換える．

(3) 委譲形式の部分（**self**や**super**に対するメソッド呼び出し）を，対応する関数呼び出しに置き換える．

たとえば，あるメソッドが図5(a)の2つの評価器オブジェクト**eval-A**, **eval-B**の委譲によって実行されているとする．これらのメソッドは，前処理によって図5(b)のような関数群へ変換される．

3.2 副作用(1)：メッセージ・同期等のI/O

部分計算をメソッド本体の実行部分に限定した場合，ベースレベルプログラム中のメッセージの送信やオブジェクトの生成といった操作は，入出力に関わる副作用と見なせる（以下，代入のような副作用と対比して**I/O副作用**と呼ぶ）．このような副作用は，関数型言語の部分計算を素朴に利用し，動的な関数呼び出しとして扱えばよいように思える．しかし，このままでは，図6の例のように副作用の順序や回数が正しく保たれない場合がある．

我々は，関数型言語の部分計算器をI/O副作用に対して拡張する**preaction**という枠組みを提案することで，この問題を解決する．ある記号値のpreactionとは，その記号値を生成するまでに行われた副作用の履歴である．たとえば，

```
(progn (send x :hello) 123)
```

という式の値は123であるが，それまでに`(send x :hello)`という副作用を起こしている．部分計算器の内部では，この値を

(1) 副作用の消滅

```
(* 5 (progn (send a :foo) 2)) ⟹10
```

(2) 順序の入れ替え

```
(let ((x (send a :first)))
  (cons (send a :second) x))
⟹ (cons (send a :second) (send a :first))
```

(3) 副作用の複製

```
(let ((x (send a :foo))) (cons x x))
⟹ (cons (send a :foo) (send a :foo))
```

図6 I/O副作用が正しく扱われない例
Fig. 6 Examples of incorrect treatment of I/O type side-effects.

$$\mathcal{PE} : Exp \to Env \to Sval^* \times Sval$$

$$\mathcal{PE}[\![var]\!]\rho = {}^{\langle\langle\rangle\rangle}\rho(var)$$
$$\mathcal{PE}[\![num]\!]\rho = {}^{\langle\langle\rangle\rangle}Num(num)$$
$$\mathcal{PE}[\![(+\ e_1\ e_2)]\!]\rho$$
$$= \text{case } (\mathcal{PE}[\![e_1]\!]\rho, \mathcal{PE}[\![e_2]\!]\rho) \text{ of}$$
$$({}^{\langle\langle p_1\rangle\rangle}Num(n_1), {}^{\langle\langle p_2\rangle\rangle}Num(n_2)) : {}^{\langle\langle p_1 \cdot p_2\rangle\rangle}Num(n_1 + n_2)$$
$$({}^{\langle\langle p_1\rangle\rangle}s_1, {}^{\langle\langle p_2\rangle\rangle}s_2) : {}^{\langle\langle p_1 \cdot p_2\rangle\rangle}Top((+\ s_1\ s_2))$$
$$\mathcal{PE}[\![(\text{send } e_1\ e_2)]\!]\rho = {}^{\langle\langle p_1 \cdot p_2 \cdot s\rangle\rangle}s$$
$$\text{ただし}\quad s = Top((\text{send } s_1\ s_2)),\quad {}^{\langle\langle p_i\rangle\rangle}s_i = \mathcal{PE}[\![e_i]\!]\quad (i = 1, 2)$$
$$\mathcal{PE}[\![(\text{let } ((v_1\ e_1)\cdots(v_n\ e_n))\ e)]\!]\rho$$
$$= {}^{\langle\langle p_1 \cdots p_2\rangle\rangle}\mathcal{PE}[\![e]\!]\rho[s_i/v_i]$$
$$\text{ただし} {}^{\langle\langle p_i\rangle\rangle}s_i = \mathcal{PE}[\![e_i]\!]\rho\quad (i = 1, \ldots, n)$$
$$\mathcal{PE}[\![(\text{progn } e_1\ \ldots\ e_n)]\!]\rho = {}^{\langle\langle p_1 \cdots p_n\rangle\rangle}s_n$$
$$\text{ただし} {}^{\langle\langle p_i\rangle\rangle}s_i = \mathcal{PE}[\![e_i]\!]\rho\quad (i = 1, \ldots, n)$$

図7 拡張された部分計算の規則（一部）
Fig. 7 Extended rules of partial evaluation.

$${}^{\langle\langle}\texttt{(send x :hello)}{}^{\rangle\rangle}Num(123)$$

という形で保持する（《》に囲まれた部分がpreactionである）．

このようなpreactionによって拡張された部分計算の規則は図7のようになる．直感的には，ある式の引数に含まれるpreactionを，その結果のpreactionにコピー（実際にはグラフ構造におけるノードの共有）するように拡張されている．

副作用の回数は，preactionを含めた記号値を1つのグラフ（DAG）構造によって表現する[12]ことで対処する．つまり，preactionの中や記号値本体の複数箇所に現れる値は，グラフ上で共有されるノードとして表現し，部分計算後に記号値から実行可能なコードを再構成する際に，let形式によって束縛するプログラムを出力する．

たとえば，図6の例(1)を拡張された規則に従って

部分計算すると

$$\begin{aligned}
&\mathcal{PE}[\![\texttt{(* 5 (progn (send a :foo) 2))}]\!]\rho \\
&= \mathrm{apply}(*,\ \mathcal{PE}[\![\texttt{5}]\!]\rho, \\
&\qquad \mathcal{PE}[\![\texttt{(progn (send a :foo) 2)}]\!]\rho) \\
&= \mathrm{apply}(*, Num(5), {}^{\ll\texttt{(send a :foo)}\gg}Num(2)) \\
&= {}^{\ll\texttt{(send a :foo)}\gg}\mathrm{apply}(*, Num(5), Num(2)) \\
&= {}^{\ll\texttt{(send a :foo)}\gg}Num(10)
\end{aligned}$$

のようになり，これをプログラムの形に直すと

```
(progn (send a :foo) 10)
```

を得る．

3.3 副作用 (2)：オブジェクトの状態変数

ベースレベルプログラムの持つもう 1 つの副作用は，オブジェクトの状態変数を更新する代入操作である．これを部分計算で扱うために，評価器は状態渡し (store passing) の形式へと変換し，ベースレベルの代入操作をメタレベルではリストのコピーによって表現する．さらに，更新された変数を，実際の状態変数の値へ反映させるために，部分計算時のメソッドの継続に疑似代入形式を置き，部分計算の後処理で本来の代入操作を再構成する．

評価器をベースレベルメソッドに関して部分計算する場合，メソッド本体の実行は，メソッドの引数とオブジェクトの状態変数を環境として評価関数 **eval** を呼び出し，終了時に更新された状態変数を書き戻す (**update**) ような式として扱われる．たとえば，状態変数 **a**, **b** を持つオブジェクトのメソッドが起動される場合は，図 **8** に示すような式が部分計算の対象となるように変換される☆．

3.4 メタレベルの動的な変更

ABCL/R3 システムでは，メタレベルの評価器オブジェクトを実行時に入れ替えることで，ベースレルプログラムの解釈のされ方を実行時に変更できる．これによって，たとえば負荷の状況に応じてオブジェクト生成の挙動を変更することができる．しかし，このことは，部分計算の対象が静的に決定できないことに相当するので，効果的な部分計算は困難である．

ABCL/R3 システムでは，**コードバージョニング**によってこれを解決する．これは，あるベースレベルメソッド (M) を解釈実行する評価器オブジェクトが複数 (たとえば E_1 と E_2) あり，どれが用いられるかが実行時に決定されるような場合，各評価器ごとに部分計算したコード ($PE(E_1, M), PE(E_2, M)$) を用意しておき，実行時にそれらを選択するものである．

☆ 実際には，このメソッド本体が部分計算されるので，環境が作られたりすることはなく，後処理によって通常の変数参照・更新へ変換される．

——元プログラム——

```
;;; クラスfooの定義
(defclass class-foo () (a ...) (b ...))
;;; メソッド定義
(defmethod class-foo foo (x y) ⟨fooの本体⟩)
```

——部分計算される式——

```
(eval '⟨fooの本体⟩
  ;; 評価環境（変数名→アドレス）
  (make-env '((a . 0) (b . 1) (x . 2) (y . 3)))
  ;; 終了時の継続
  #'(lambda (result-val result-store)
      ;; 状態変数更新のための疑似代入形式
      (update 'a (nth 0 result-store))
      (update 'b (nth 1 result-store))
      result-val)
  ;; ストア（アドレス→値）
  (list a b x y))
```

図 8 ベースレベルプログラムと部分計算される式の対応
Fig. 8 A base-level program and a target expression of partially evaluation.

3.5 部分計算によるコンパイルの例

例として，メタレベルの評価器を変更した場合の部分計算の例を示す．まず，特定の変数 **client**, **worker2** の参照をモニタする評価器のクラス **monitor-eval** を **default-eval** のサブクラスとして定義する (図 **9** (i))．クラス **monitor-eval** では，変数参照の解釈をするメソッド **eval-variable** だけを再定義し，変数 **client**, **worker2** を参照した際に，コンソールへ通知を送る．それ以外の式は，委譲によって通常の解釈がなされる．

ベースレベルのプログラム (図 9 (ii)) は，2 つの **worker** オブジェクトに見積りを聞き，大きい値を返した方のオブジェクトを **client** に渡している．

部分計算の結果は，図 9 (iii) のようになる．参照される変数名がモニタされているかどうかのチェックは部分計算時に決定できるため，結果の中には現れず，実際に通知をするコードだけが埋め込まれている．プログラム中の下線 a，b，c の変数参照に対応するのが注 (※) のついた行である．他の変数 (**worker1** や一時変数) に関しては，通知が行われず普通に参照されていることに注意されたい．また，ベースレベルのメッセージ送信 (**estimate** と **start-job**) とメタレベルのメッセージ送信 (**notify**) が混在しているが，これらの間の順序も正しく保たれている．

4. 性能測定

部分計算コンパイルの技法を用いて，現在までにプロトタイプシステムが作成されている．部分計算は十分効果的に行われており，実際，ベンチマークプログラムを，デフォルトの評価器定義の下で部分計算した

```
―――――――――(i) メタレベル定義―――――――――
;;; クラス定義(default-eval を継承)
(defclass monitor-eval (default-eval))
;;; 変数評価メソッド
(defmethod monitor-eval eval-var (var env)
  (if (member var '(worker2 client))
    ;; モニタされている変数の場合
    (progn (notify *console* var) ; 通知をして
           (lookup var env))      ; 変数参照
    (lookup var env))) ;; 他の変数の参照
―――――――――(ii) ベースレベル―――――――――
(defmethod server request
                    (client worker1 worker2)
  (let ((e1 (estimate worker1))
        (e2 (estimate worker2(a))))
    (start-job client(b)
     :with (if (> e1 e2) worker1 worker2(c)))))
―――――――――(iii) 部分計算の結果―――――――――
(defmethod server request
                    (client worker1 worker2)
  (let ((t313 (estimate worker1)))
    (notify *console* 'worker2)          ; ※ a
    (let ((t314 (estimate worker2(a))))
      (notify *console* 'client)         ; ※ b
      (if (> t313 t314)
        (start-job client(b) :with worker1)
        (progn
          (notify *console* 'worker2)    ; ※ c
          (start-job client(b)
            :with worker2(c)))))))
```

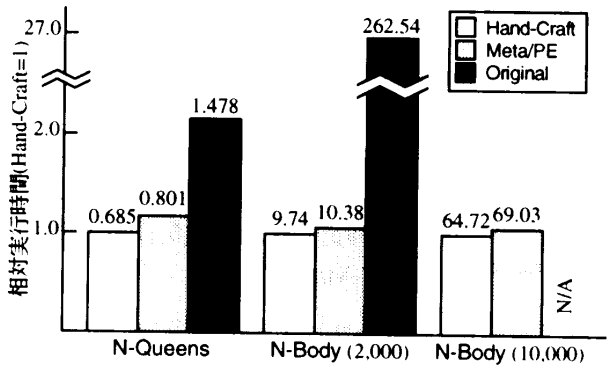
図9 コンパイルの例
Fig. 9 Example of our compilation.

ものは，同じベンチマークプログラムを非自己反映言語のコンパイラによって直接実行したものと同等の速度で実行できている（詳細は別の文献 18), 19) を参照されたい）．

ここでは，並列アプリケーションに応用した場合の性能を測定した結果を示す．まず，素朴に記述された並列アプリケーションを用意し，それへの最適化を ABCL/R3 のメタレベルに記述し，部分計算コンパイルによって実行する（Meta/PE）．また，メタレベルを用いずにアプリケーションを書き換えることで同じ最適化を行ったもの（Hand-Craft）および最適化を施さないもの（Original）をそれぞれ ABCL/f で記述し，実行時間を比較する．

実行したアプリケーションとその最適化は，n-Queens 探索問題に 1 章で例示した最適化を施したもの（N-Queens）と，N 体シミュレーションの近似計算アルゴリズムを並列化したものに，遠隔プロセッサにある空間データをキャッシュする最適化を施したもの（N-Body）である．

実際に富士通 AP1000（64 × 25 MHz Sparc）で実行した結果を図 10 に示す．グラフは，11-Queens，粒子数 2,000 および 10,000 の N 体シミュレーションの各アプリケーションについて，Hand-Craft を 1 とした相対実行時間☆を示している．グラフから，Meta/PE は Hand-Craft に対して 7%（N 体シミュレーション）・17%（n-Queens）のオーバヘッドしかなく，Original と比べて十分な効率改善になっていると言える．



棒の上の数値は実行時間（秒）を表す．

図10 並列アプリケーションでのオーバヘッド
Fig. 10 Overhead of our compilation scheme in parallel applications.

部分計算コンパイルによる実行のオーバヘッドの原因は，主として (1) 元のプログラムの 1 つのメソッドが複数のメソッドに分割されてしまう，(2) 冗長な状態変数の更新がある，といった点であると推測される．これらは，部分計算を行った後に，インライン化やデータフロー解析のような技法を利用することで低減できると思われる．

5. 関連研究

ABCL/R3 の前身である ABCL/R2 システムでは，決められた種類の式は実際にメタレベルで解釈実行し，それ以外の式はコンパイル実行する，部分コンパイルという技法が用いられている[6]．この方式の問題は，メタレベルの拡張性（解釈実行される式の種類）と実行効率がトレードオフになっていることである．一方，今回提案した部分計算コンパイルでは，ユーザに対してはすべての式が解釈実行されているモデルを与えつつも，実際には解釈実行を行わない効率で実行できている．

自己反映言語 Open C++の最適化として，無駄な reify-reflect 対を減らすものが提案されている[13]．これは，解釈実行の除去に相当するが，メタインタプリ

☆ N 体シミュレーションに関しては，粒子の移動の計算に要した時間のみで，データの初期化等にかかる時間は含まない．また，使用した ABCL/f 処理系は大域 GC が未定義であったため，N-Body (10,000) Original 版は実行できなかった．

タの定義が変更可能であるような一般的な場合への適用に関しては知られていない.

言語拡張の手段として,コンパイラのメタレベルをオブジェクト指向的に公開し,コンパイル時のプログラム変換や,コード生成の方法を容易にユーザが変更できるようにする,Metaobject Protocol に基づくコンパイラ[2)~4)]の研究がさかんになってきている.しかし,この方法で動的な制御を記述するのは,自己反映プログラミングと比べて煩雑である☆.

AL-1/D のような解釈実行に基づく自己反映言語システムを用いても,分散アプリケーションのような,遠隔メッセージ送受信のコストが大きい環境では,そのコストを減らす最適化をメタレベルに記述することで,解釈実行のオーバヘッドを上回る性能向上が得られる[7)].しかし,並列アプリケーションでは,解釈実行のオーバヘッドは相対的に大きな割合を占めてくるため,これを除去するコンパイル技法の意義は大きい.

6. まとめ

本論文では並列自己反映言語システムの部分計算によるコンパイル技法を提案し,ABCL/R3 用のプロトタイプシステムを作成した.これまで並列言語・自己反映言語のメタレベルの部分計算は困難であったが,副作用を扱うための拡張と,メソッド単位の適用の枠組みによって,解釈実行を除去することが可能になった.これによって,(1) 従来,アプリケーションに埋め込まれていたような制御を,性能をほとんど落とさずにメタレベルに分離して記述することが可能になり,(2) 従来の自己反映言語の実現方式と比べて大きい拡張性と高い性能を同時に達成できた.実際に並列計算機 AP1000 上での実験では,メタレベルに分離された制御記述を行った場合でも,7~17%のオーバヘッドにすぎないという結果を得ている.

謝辞 浅井健一氏は部分計算や自己反映計算に関する数々の議論をしてくださいました.田浦健次朗氏からは多くのコメントに加え,ABCL/f 使用の便宜を図っていただきました.また,一杉裕志,千葉滋の各氏から有意義なコメントをいただきました.ここに感謝の意を表します.

参考文献

1) Taura, K., Matsuoka, S. and Yonezawa, A.: ABCL/f: A Future-Based Polymorphic Typed Concurrent Object-Oriented Language – Its Design and Implementation, *Proc. DIMACS Workshop on Specification of Parallel Algorithms*, Blelloch, G., Chandy, M. and Jagannathan, S. (Eds.), pp.275–292, American Mathematical Society (1994).

2) Rodriguez, Jr., L.: A Study on the Viability of a Production-Quality Metaobject Protocol-Based Statically Parallelizing Compiler, *Proc. IMSA'92 Workshop on Reflection and Meta-Level Architecture, Tokyo*, Yonezawa, A. and Smith, B.C. (Eds.), pp.107–112 (1992).

3) Lamping, J., Kiczales, G., Rodriguez, L. and Ruf, E.: An Architecture for an Open Compiler, *Proc. IMSA'92 Workshop on Reflection and Meta-Level Architecture, Tokyo*, Yonezawa, A. and Smith, B.C. (Eds.), pp.95–106 (1992).

4) Chiba, S.: A Metaobject Protocol for C++, *Proc. OOPSLA'95, Austin, Texas*, pp.285–299, ACM (1995).

5) Watanabe, T. and Yonezawa, A.: Reflection in an Object-Oriented Concurrent Language, *Proc. OOPSLA'88, San Diego, California*, pp.306–315, ACM (1988).

6) Masuhara, H., Matsuoka, S., Watanabe, T. and Yonezawa, A.: Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently, *Proc. OOPSLA'92, Vancouver, British Columbia*, pp.127–145, ACM (1992).

7) Okamura, H. and Ishikawa, Y.: Object Location Control Using Meta-level Programming, *Proc. ECOOP'94*, Tokoro, M. and Pareschi, R. (Eds.), pp.299–319, Springer-Verlag (1994).

8) Tomokiyo, T., Konaka, H., Maeda, M., Hori, A. and Ishikawa, Y.: Meta-level Architecture in OCore, *Proc. OOPSLA'93 Workshop on Object-Oriented Reflection and Metalevel Architectures, Washington, D.C.* (1993).

9) Yokote, Y.: The Apertos Reflective Operating System: The Concept and Its Implementation, *Proc. OOPSLA'92, Vancouver, British Columbia*, pp.414–434 (1992).

10) Futamura, Y.: Partial Evaluation of Computation Process – An Approach to a Compiler-compiler, *Systems, Computers, Controls*, Vol.2, No.5, pp.45–50 (1971).

11) Jones, N.D., Gomard, C.K. and Sestoft, P.: *Partial Evaluation and Automatic Program Generation*, Prentice Hall (1993).

12) Weise, D., Conybeare, R., Ruf, E. and Seligman, S.: Automatic Online Partial Evaluation, *Proc. FPCA'91*, Hughes, J. (Ed.), pp.165–191, Springer-Verlag (1991).

☆ たとえば実行時に条件をテストし,分岐を行うようなコードを生成する規則を記述しなければならない.

13) Chiba, S. and Masuda, T.: Open C++ and Its Optimization, *Proc. OOPSLA'93 Workshop on Object-Oriented Reflection and Metalevel Architectures, Washington, D.C.* (1993).
14) Chambers, C.: Towards Efficient Implementation of Computational Reflection, *Proc. OOPSLA'91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming* (1991).
15) Meyer, U.: Techniques for Partial Evaluation of Imperative Languages, *Proc. PEPM'91*, pp.94–105 (1991).
16) Baier, R., Glük, R. and Zöchling, R.: Partial Evaluation of Numerical Programs in Fortran, *Proc. PEPM'94, Orlando, Florida*, ACM (1994). Published as Technical Report 94/9, Department of Computer Science, University of Melbourne.
17) Andersen, L.O.: Program Analysis and Specialization for the C Programming Language, PhD Thesis, DIKU, University of Copenhagen (1994). Published as DIKU Report 94/19.
18) 増原英彦，松岡　聡，米澤明憲：並列自己反映システムの部分計算によるコンパイル技法，並列処理シンポジウム（JSPP），福岡，pp.273–280 (1995).
19) Masuhara, H., Matsuoka, S., Asai, K. and Yonezawa, A.: Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation, *Proc. OOPSLA'95, Austin, Texas*, pp.300–315, ACM (1995).

(平成7年 9 月1日受付)
(平成7年12月8日採録)

増原　英彦（正会員）

1970年生．1992年東京大学理学部情報科学卒業．1995年同大学大学院博士課程退学．同年より，同大学教養学部助手．1994～1995年日本学術振興会特別研究員．並行計算，リフレクティブ言語，部分計算などに興味を持つ．日本ソフトウェア科学会，ACM各会員．

松岡　聡（正会員）

1963年生．1986年東京大学理学部情報科学科卒業．1989年同大学大学院博士課程中退．同年同大学助手．現在，同工学部情報工学専攻講師．理学博士．オブジェクト指向言語，並列システム，リフレクティブ言語，制約言語，ユーザ・インタフェースソフトウェアなどの研究に従事．情報処理学会，ACM，IEEE-CS各会員．ACM日本支部書記．

米澤　明憲（正会員）

1947年生．1977年 Ph.D. in Computer Science (MIT)．1989年より東京大学理学部情報科学科教授．超並列ソフトウェアアーキテクチャ，ソフトウェア基礎論，人工知能基礎論などに興味を持つ．著書「算法表現論」，「モデルと表現」（岩波書店），編著書「ABCL: An Object-Oriented Concurrent System」，「Research Directions in Concurrent Object-Oriented Computing」(MIT Press) 等．現在 IEEE Parallel & Distributed Technology 編集委員，ACM Transaction on Programming Languages and Systems (TOPLAS) 副編集長．1992年よりドイツ国立情報処理研究所（GMD）科学顧問．現在日本ソフトウェア科学会理事長．