プログラミング－言語・基礎・実践－研究報告　9－11
（1992．10．30）

自己反映的並列オブジェクト指向言語 ABCL/R2 コンパイラについて

増原英彦　松岡聡　米澤明憲

東京大学 理学部 情報科学科

自己反映計算の機能を持つ並列オブジェクト指向言語 ABCL/R2 の実現、特にコンパイラと実行時システムに関して述べる。従来の自己反映的言語の実現は、解釈実行に基づくものがほとんどであったが、本実現では、スクリプトを部分コンパイルすることで、並列オブジェクト指向計算に関する自己反映計算の機能を保ちつつ、他の部分は効率的に実行される。また、システムオブジェクトの自己具体化や軽量オブジェクトなどによって、システムのオーバーヘッドを低減している。ベンチマークの結果、今回の実現では、自己反映計算をする部分は従来のものに比べて約二桁の速度の向上が、また、自己反映計算をおこなわない部分では自己反映計算の機能のない言語に匹敵する性能が示された。

Compilation of Object-Oriented Concurrent Reflective Language ABCL/R2

Hidehiko Masuhara　Satoshi Matsuoka　Akinori Yonezawa

Department of Information Science, Faculty of Science,
The University of Tokyo

7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

This report presents our implementation scheme of object-oriented concurrent reflective language ABCL/R2, in particular, its compiler and runtime system architecture. Most previous work of implementation of reflective language is based on interpreters. In our implementation, however, a *partial compiler* translates a script (method) of an object into a mixed code where the reflective operations are still interpreted, while the other operations are executed directly. Furthermore, its runtime system is enhanced by using techniques such as *light-weight objects* and *self-reification*. The benchmark shows that the reflective part of our implementation exhibits two orders of magnitude speed improvements over previous reflective language, and the performance of the non-reflective part is comparable to those of non-reflective languages.

1 はじめに

自己反映計算 (computational reflection) とは、計算システムが、自分自身の計算過程に対しておこなう計算のことである [7, 4]。プログラミング言語において、この自己反映計算は、言語機能の拡張や、システムの動的な最適化のための枠組として、注目されている。特に並列システムにおいては、システムの構成や解こうとする問題領域に応じた特化をおこなうことが重要になってくるため、自己反映計算の機能がいっそう有用である [11]。これまでの ABCL/R, ACT/R, ABCL/R2, RbCl といった言語の研究 [8, 9, 6, 2] を通じて「並列計算」を言語の枠組から制御できることが示されてきた。

1.1 自己反映的並列オブジェクト指向言語 ABCL/R2

本報告では、自己反映計算的並列オブジェクト指向言語 ABCL/R2 [6] の実現、特にコンパイラと実行時システムに関して述べる。ABCL/R2 [6] は *Hybrid Group Architecture*(HGA) に基づいている。HGA は ABCL/R [8] の *individual-based architecture* と ACT/R [9] の *group-wide architecture* の両方の性質をとり入れ、オブジェクト間で共有される「計算資源」を言語の枠組から制御することを目指したものである。特徴的なことは次のとおりである。

オブジェクトのグループ化とグループのメンバ間の共有資源
ABCL/R2 のオブジェクトは必ず一つのグループに属している。グループ内にいるオブジェクトは、メタレベルにある**グループの核 (kernel) オブジェクト**としてあらわされる資源を共有している。このグループはユーザが動的に生成することができる。

メタグループとオブジェクト単位・グループ単位のリフレクティブタワー
ABCL/R2 のオブジェクトは、メタオブジェクトによって表わされており、そのメタオブジェクトもまたオブジェクトであるために、メタオブジェクトのメタオブジェクトによって表わされている、というタワー (individual tower) がある。また、グループのメタレベル (メタグループ) もグループを成しており、メタグループに対するメタグループというタワー (group tower) と二種類のタワーが存在する。前者は主に個々のオブジェクトの構造的側面を決定し、後者はグループの計算的側面を決定する。

図 1 は ABCL/R2 の言語モデルをあらわしている。図中で "meta-gen" とあるのが**メタオブジェクト生成器 (metaobject generator)**、"GMgr" が**グループマネージャ**、"Eval" が**評価器 (evaluator)** と、これら三つがグループ核オブジェクトである。

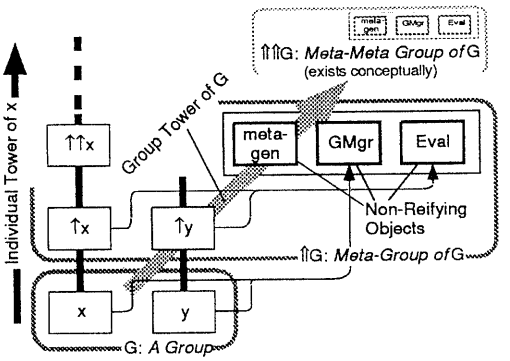

図 1: ABCL/R2 の言語モデル

この核オブジェクトの役割りは次のようになっている。*グループマネージャ*: グループの管理をおこなう。具体的には、他の核オブジェクトへの参照の管理や、新たなグループ生成などである。*メタオブジェクト生成器*: 新たにオブジェクトが生成されるときに、メタレベルにメタオブジェクトを生成する。生成されたメタオブジェクトは、オブジェクトのメッセージキュー・スクリプト・状態変数等を持つ。オブジェクトへ送られたメッセージは、メタオブジェクトが受け取り、パタンマッチをおこなったのちに対応するスクリプトを評価器へ送る。評価器は [:do *Exp Env Id Gid Eval*] という形のメッセージを受け取り、*Exp* に応じた処理をおこなう。

非自己反映的オブジェクト
オブジェクト単位の自己反映計算の能力の必要がないオブジェクトをユーザが指示することで効率よく実現するために、**非自己反映的 (non-reifying) オブジェクト**がある。この非自己反映的オブジェクトと通常のオブジェクトの違いは、前者の方が効率的に実行される代わりに、オブジェクト単位の自己反映計算ができないことである。非自己反映的オブジェクトが、他のオブジェクトやグループに対して自己反映的操作をおこなうことは可能である。

1.2 自己反映的並列オブジェクト指向言語の実現

ABCL/R2 のような自己反映的並列オブジェクト指向言語を実現するためには、(1) メタサーキュラな定義をいかにして実現するか、(2) その実現をいかに効率のよいものにするか、という二つの問題がある。

前者に関しては、ABCL/R2 が HGA に基づいている — オブジェクト単位・グループ単位の二種類の自己反映的タワーが存在し、それぞれ意味的には無限の階

層を持っている — ため問題が複雑になっている。具体的には、タワーを有限の範囲で実現することと、それぞれのタワーでの自己反映計算の因果関係 (causal-connection) を満たす — オブジェクト単位の自己反映的操作は個々のオブジェクトの構造や実行に反映させる一方で、グループ単位の自己反映的操作は、グループの構成員すべてに反映させる — 必要がある。

また、後者の効率に関してであるが、単純には解釈実行 (インタプリタ実行) による実現が考えられるが、これでは十分な効率が得られない。そのため、コンパイルをおこなう必要があるが、(i) コンパイルされる操作と自己反映計算の対象となる操作を混在させる、(ii) 必要以上のコンパイルのために自己反映計算の能力を失わないようにする、といった点を解決する必要がある。

我々はこれらの問題に関して (1) グループ核オブジェクトの自己具体化 (self-reification)・遅延生成、(2) スクリプトの部分コンパイル・軽量オブジェクト・dynamic progression ・レベル間メッセージ転送、といった技法を提案し、実際に ABCL/R2 の処理系を作成した。ベンチマークの結果、この処理系の効率は、非自己反映的部分においては ABCL/1 [10] (自己反映計算の能力を持たない並列オブジェクト指向言語) の疑似並列実装と同等かそれ以上であり、自己反映的部分の実行では、ABCL/R [8] (ABCL/R2 の前身で、individual-based architecture に基づく自己反映的並列オブジェクト指向言語) の解釈実行による実装に対して二桁近く上回っていることが分かっている。

2 ABCL/R2 実装の概要

現在、ABCL/R2 は共有メモリ型並列計算機 OMRON LUNA-88K 上のマルチスレッドの Common Lisp 上で実現されている。また、逐次 Common Lisp 上での疑似並列版は、スケジューリング部分を除いて、マルチスレッド版とほぼ同様の実現をしている。この実現の概要は次のとおりである。(詳細は [5] を参照されたい。)

- **低レベル言語機構**が並列オブジェクト実行のための基本的手続を提供している。具体的には、Lisp のスレッドをオブジェクトへの割り当てや、単純なメッセージ送信の手続きである。
- オブジェクトのスクリプトは**部分コンパイル**によって、自己反映的 (自己反映計算の対象となる) コードと非自己反映的 (自己反映計算の対象とならない) コードが混在したコードへコンパイルされる。
- デフォルトのシステムオブジェクトは、二種類のコンパイルされたスクリプトを持つことで、生成当初は非自己反映的オブジェクトのと同様の効率で実行される。一方で、必要に応じて**自己具体化** (self-reification) することで、メタサーキュラな構造を実現している。
- 通常のオブジェクトの機能を制限することでスケジューリングの費用を小さくした**軽量オブジェクト**によって効率化が図られている。コンパイルされたコードが用いる**継続 (continuation) オブジェクト**や**軽量メタオブジェクト**などがそれである。
- オブジェクト単位のタワー (individual tower) は、自己反映計算の能力を必要に応じて増やしてゆく **dynamic progression** の手法を用いることで実現されている。
- 自己反映計算の能力の異なるオブジェクト間の通信は、レベル間メッセージ転送の機構により、不必要な具体化 (reification) を起こさずに実現されている。
- 非自己反映的オブジェクトのスクリプトは、Lisp の式へ変換することで直接実行される。
- グループ単位のタワーは、グループマネージャの**遅延生成 (lazy creation)** によって実現されている。

以下の章では、これらのうち、実行時システムとコンパイラについて述べる。

3 実行時システム

本章では、ABCL/R2 の実行時システムについて述べる。

3.1 低レベルの言語機構

システムの低レベルでは、オブジェクトは Lisp の構造体データとして表わされており、メッセージキュー、オブジェクト固有のデータ、メッセージ処理用のラムダクロージャ等を保持する。これらのオブジェクトは図 2 のような機構によって実行される。このレベルでは (1) Lisp のスレッドをオブジェクトに割り当てる機構や (2) 基本的なメッセージ送信手続きを提供している。

処理すべきメッセージがあるオブジェクトは、一つのスケジューリングキューに入っている。Lisp のスレッドはそのキューの先頭からオブジェクトを一つとり出して、そのオブジェクトに対応する Lisp のラムダクロージャを実行する。このラムダクロージャは、メッセージキューからメッセージをとり出して、オブジェクトの種類に応じた処理をおこなう。

オブジェクト S が本体:B, 返答先:R なるメッセージをオブジェクト T へ送る場合、メッセージ送信手続きは、組 $\langle B, R, S\rangle$ を T のメッセージキューへ書き込み、必要ならば T をスケジューリングキューへ入

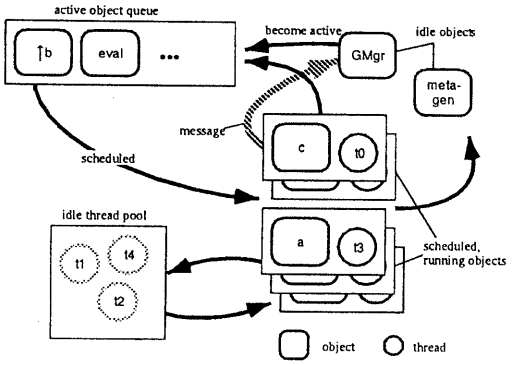

図 2: 低レベル言語機構

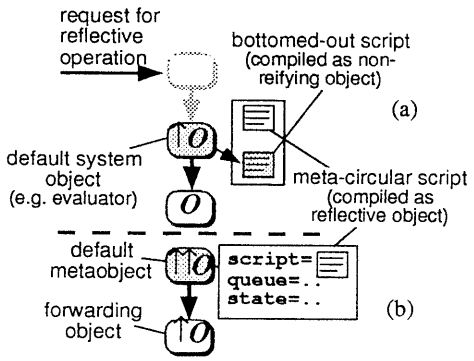

図 3: オブジェクトの実現の構造

れる。(メッセージの送り先のオブジェクトが軽量オブジェクトの場合は、ここに書いたものと異なる送り方をする。これについては後述する。)

3.2 オブジェクト単位のタワー

ユーザーが生成する通常の(自己反映的)オブジェクトは、実際には図 3 のようにデフォルトのメタオブジェクトによって表わされている。

このデフォルトのメタオブジェクトのスクリプトは、非自己反映的オブジェクトと同じようにコンパイルされたコードと、自己反映的オブジェクトとしてコンパイルされたコードの両者を持っている。

オブジェクト O のデフォルトのメタオブジェクトである $\uparrow O$ は生成当初は前者のコードを用いて非自己反映的オブジェクトと同様の効率で実行される。非自己反映的オブジェクトと異なるのは、$\uparrow O$ に対する自己反映的操作が要求された(例えば、$\uparrow\uparrow O$ を通じて $\uparrow O$ のスクリプトを変更する)場合である。このとき、$\uparrow O$ の内部状態を集め、$\uparrow\uparrow O$ によって表わされるオブジェクト

```
;;; 自己反映的オブジェクトの
;;; メタオブジェクトの起動ルーチン
(defun invoke-object (meta-O M R S)
  ;; M R S は meta-O へのメッセージ
  (match M
   (is [:message body reply sender]
     ;; O へのメッセージの場合
     (if (O の状態が dormant)
       (let ((script ⟨body をパタンマッチ⟩))
        ⟨script を評価器へメッセージとして送る⟩)
       ⟨body, reply, sender を O のキューへ保存⟩))
   (is ...)))

;;; 非自己反映的オブジェクトの
;;; メタオブジェクトの起動ルーチン
(defun invoke-object (meta-O M R S)
  ;; M R S は meta-O へのメッセージ
  (match M
   (is [:message body reply sender]
     ;; O へのメッセージの場合
     (let ((script ⟨body をパタンマッチ⟩))
       ⟨script を直接実行⟩))
   (is ...)))
```

図 4: 自己反映的・非自己反映的メタオブジェクトの起動ルーチン

へと自己具体化 (self-reification) する。その後、$\uparrow O$ のスクリプトとしては、自己反映的オブジェクトとしてあらかじめコンパイルされていたものを用いる。

メタオブジェクトのみならず他の核オブジェクトも、この自己具体化の機構を用いて実現されている。そのため、システムが作るデフォルトのオブジェクトは、そのオブジェクトに対する自己反映計算をおこなわない限り、非自己反映的オブジェクトと同様の効率で実行される一方で、メタサーキュラな定義を実現している。

デフォルトのメタオブジェクトが、非自己反映的オブジェクトと同様のスクリプト実行をするので、オブジェクト自身の起動にかかる手順は、そのオブジェクトが非自己反映的オブジェクトである場合と大差がない。図 4 は自己反映的オブジェクトと非自己反映的オブジェクトのスクリプト起動ルーチンの概略であるが、自己反映的オブジェクトと非自己反映的オブジェクトの差は、スクプト実行中に来たメッセージは一度キューに保存されることくらいであり、この部分において解釈実行はおこなわれていない。

3.3 軽量オブジェクト

軽量オブジェクトは、ABCL のオブジェクトの持つ機能のうち、いくつかに制限を加えることで、高速な実現を与えることを目指したものである。

具体的には、軽量オブジェクトはメッセージキューを持たずに、スクリプトをコンパイルした Lisp のラムダクロージャを持っている。あるオブジェクト O が軽量オブジェクト L へメッセージを送る場合、O はメッ

セージをキューに入れる代わりに、L のラムダクロージャを呼び出す。そのため、軽量オブジェクトは低レベルのスケジューリング機構を介さずにスクリプト実行ができる。また、軽量オブジェクトは状態変数を持たないため、通常のオブジェクトよりもメモリ効率がよい。

軽量オブジェクトとして実現可能なオブジェクトには、次のような性質が要求される。(1) 状態を持たないオブジェクトであること。 (2) 一時に一つしかメッセージが来ないこと (メッセージキューがないため)。(3) 単純な操作しかおこなわないこと (ループやメッセージの返事を持つような操作はデッドロックの危険があるため)。

並列オブジェクト指向計算では、計算結果を他のオブジェクトへ転送するためだけや、同期をとるだけのために継続オブジェクトを作ることが多く、このようなオブジェクトは軽量オブジェクトで実現可能な場合が多い。実際に ABCL/R2 システムでも、コンパイルされたコード中の継続オブジェクトや、メッセージを転送するだけの軽量メタオブジェクトなどは軽量オブジェクトによって実現される。また、この軽量オブジェクトは、ユーザープログラムからも生成することができる。

3.4 非自己反映的オブジェクトのコンパイル

非自己反映的オブジェクトのスクリプトのコンパイル方針は、可能な限り (直接) Lisp へ変換することで直接実行することである。ただし、Lisp 上で他のオブジェクトからの返答を待つことがあるために、工夫が必要となる。具体的には、継続オブジェクトによる継続渡し形式へと変更することで実現する。

4 ABCL/R2 コンパイラ

多くの自己反映的言語はメタサーキュラな解釈実行器によって意味モデルが与えられ、実現も解釈実行に基づいている [1, 8]。また、ABCL/R2 の意味モデルもメタサーキュラなによって与えられている [6]。

そのため、最も単純な実現の方針は、メタサーキュラな解釈実行器を考えて、それを遅延生成 (lazy creation) によって有限の範囲で構成する方法である。しかしながら、この方法では効率が非常に悪い。実際、ABCL/R の処理系は、自己反映計算の機能のない言語 ABCL/1 [10] に比べて 1000 倍程度遅くなっている。

4.1 自己反映的オブジェクトのコンパイル

効率化のためには、自己反映的言語においてもコンパイルをおこなう必要がある。その際に重要なことは、過度のコンパイルによって、自己反映的言語の持つ柔軟性を失わないようにすることである。

ABCL/R2 が目指すものは、並列計算における共有資源の管理を自己反映計算によってとり扱うことである。そこで、そのために必要な操作以外をコンパイルして、直接実行させることにする。具体的に、自己反映計算に対象となる操作としては、

- 変数参照
- メッセージ送信 (過去型 / 現在型)
- オブジェクト・グループ生成

の三種とした。(この方法は、CLOS メタオブジェクトプロトコル [3] の実現で、オブジェクト指向に関する計算 (スロットアクセス、総称関数の呼び出し、クラス・インスタンスの生成) 以外は Lisp レベルで直接実行されてしまうのとよく似ている。)

スクリプトの実行は、上に挙げた操作に関しては評価器へのメッセージによって解釈実行的に実現される。一方、他の操作に関しては、Lisp レベルで[1]効率的に実行される。結果として、後者の操作は自己反映計算の対象とならない。

4.2 コンパイル規則

スクリプトがコンパイルされたコードは、式中での上に挙げた自己反映的操作の出現を評価器のメッセージ送信と軽量オブジェクトによる継続渡し形式 (Continuation Passing Style) へと変換し、他の非自己反映的操作は、Lisp の式へと埋め込まれる。

コンパイル規則の一部を紹介する。関数 C は ABCL/R2 の式を評価器が処理するような Lisp のデータへと変換する。評価器が [:compiled f] のような式を受け取ると、その式の文脈の下で Lisp の関数 f を呼び出す。下の変換規則で、メッセージ送信や非自己反映的演算子の引数 $e_1, \ldots, e_n$ のうち、自己反映的操作を含まないもの (定数など) は C を用いずに、対応する Lisp 式で置き換えられる。

$C[\![id]\!]$ =[:variable id] (*Variable lookup*)

$C[\![(e_1$ <= e_2 @ $e_3)]\!]$ (*Message sending*)

```
=[:compiled
  #'(lambda (C Env Id  Gid Eval)
     [Eval <= [:do C⟦e1⟧ Env Id Gid Eval]
      @ [cont v1
           ...
         [Eval <= [:do C⟦ej⟧ Env Id Gid Eval]
          @ [cont vj
             [Eval <= [:do [:send-past u1 u2 u3]
                       Env Id Gid Eval] @ C]]]]])]
```

$C[\![(op\ e_1 \cdots e_n)]\!]$ (*Non-reflective operator*)

```
=[:compiled
```

[1]最終的には、Lisp コンパイラによって機械語レベルで実行される。

```
#'(lambda (C Env Id  Gid Eval)
   [Eval <= [:do C[[e_i]] Env Id Gid Eval]
    @ [cont v_i
       ...
       [Eval <= [:do C[[e_j]] Env Id Gid Eval]
        @ [cont v_j
           [C <= (op u_1 ... u_n)]]]]]]]
```

(ただし u_i は e_i が自己反映的操作を含む場合 v_i、含まない場合は対応する Lisp 式となる。)

4.3 コンパイル例

図 5 に式 [x <= (+ y 2)](これは x の値に (+ y 2) の値をメッセージとして送るという意味) をコンパイルした例を挙げる。ここでの [T <= M @ R] という記法は、Lisp の手続き send-message の呼び出しの略記である。また、[cont ...] も継続オブジェクトを生成する Lisp 関数の呼び出しの略記である。また、C はこの式全体の継続オブジェクトである。

このように生成されたコードには、評価器 (Eval) に対してメッセージを送っている部分が三つある。しかし、これらのメッセージは変数参照およびメッセージ送信の実行のためであり、これらの操作が自己反映的であるとした以上は本質的に必要なものである。また、評価器に送った変数参照などの操作の結果はやはりメッセージによって返されるが、この結果を受け取る継続オブジェクトは軽量オブジェクトなので、このメッセージ送信は手続き呼び出し程度のオーバーヘッドで実現される。

解釈実行をおこなった場合、上に挙げたメッセージ送信に加えて、"2" や "(+ y 2)" の評価のもメッセージによって実行されることになる。しかし、部分コンパイルでは、これらのメッセージは非自己反映的操作であるので、コンパイルコード中では "(+ y-value 2)" のように、Lisp の式によって効率的に実現される。

5 性能評価

ABCL/R2 システムの性能を評価するため、ベンチマークテストをおこなった。測定の対象は (1) 非自己反映的部分の実行、(2) 軽量オブジェクトの効果、(3) 自

```
[Eval <= [:do [:variable 'x] Env Id Gid Eval]
 @ [cont x-value
    [Eval <= [:do [:variable 'y]
                  Env Id Gid Eval]
     @ [cont y-value
        [Eval <= [:do [:past x-value
                              (+ y-value 2) nil]
                      Env Id Gid Eval]
         @ C]]]]]
```

図 5: 式 [x <= (+ y 2)] のコンパイルコード

```
[object parallel-fib-gen  ; the parallel version
 (script
   (=> :new                ; '!' returns the evaluated
    ![object fib           ; expression
       (state [reply := nil] [sub-value := nil])
       (script
        (=> [:ans x]
            (if sub-value
               [reply <= [:ans (+ sub-value x)]]
               [sub-value := x]))
        (=> 0 ![:ans 0])
        (=> 1 ![:ans 1])
        (=> n @ R
            [reply := R]
            [[parallel-fib-gen <== :new]
              <= (- n 1) @ Me]
            [[parallel-fib-gen <== :new]
              <= (- n 2) @ Me]))]))]

[object recursive-fib
  (script
   (=> 0 !0)
   (=> 1 !1)
   (=> n @ R
       [recursive-fib <= (- n 1)
        @ [cont fib-n-1
             [recursive-fib <= (- n 2)
              @ [cont fib-n-2
                   [R <= (+ fib-n-1 fib-n-2)]]]]]))]
```

図 6: フィボナッチ数を計算するオブジェクトの定義

己反映的部分の実行にかかる費用、の三点である。比較のため、自己反映計算の機能のない並列オブジェクト指向言語 ABCL/1 [10] コンパイラと C 言語 + Sun Light-Weight Processes (LWP) ライブラリ[2]上で同じアルゴリズムのプログラムを実行した。ABCL/R2 と ABCL/1 はそれぞれ Kyoto Common Lisp (KCl) 上の処理系を使用し、すべてのプログラムは Sun SPARCstation1+ 上で実行した。

ここで示すベンチマークに使用したプログラムは、図 6 に示したようなフィボナッチ数の並列・逐次計算である。オブジェクト parallel-fib-gen によって作られるオブジェクト fib は、$Fib(n)$ の計算をおこなう際に、新たに $Fib(n-1)$ と $Fib(n-2)$ の計算をおこなうオブジェクト fib を生成する。その後、この二つのオブジェクトから $Fib(n-1)$ と $Fib(n-2)$ の値を受け取り、その和を返すことで計算が完了する。一方、オブジェクト recursive-fib が $Fib(n)$ の計算をおこなう際には、まず $Fib(n-1)$ の計算をおこないその結果を受け取る。次に $Fib(n-2)$ の計算をおこないその結果を受け取って二つの和を返す。なお、ABCL/R2 では、継続オブジェクトとして軽量オブジェクトを用いた。

図 7 はベンチマーク結果である。この結果から、次のようなことがいえる:

- 並列計算では、ABCL/R2 は ABCL/1 に匹敵す

[2] C+LWP では、一つのスレッドを一つのオブジェクトに割り当てる方式をとった。

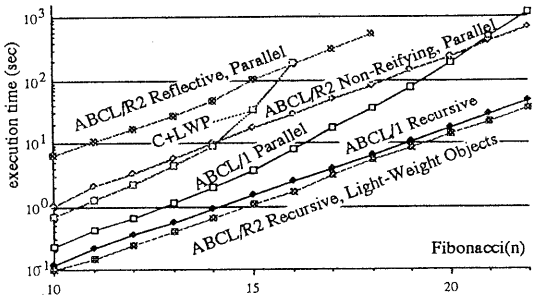

図 7: フィボナッチ数の並列・逐次計算に要した時間

る性能がある。

- 逐次計算では、ABCL/R2 は軽量オブジェクトを用いた結果、ABCL/1 よりも約 30 % の速度向上がある。これは、ユーザが適切に軽量オブジェクトを用いることで、効率を高めている好例である。

- 自己反映的オブジェクトの、非自己反映的オブジェクトに対するオーバーヘッドは 10 倍以下である。また、ABCL/R の速度と比較すると、約二桁の向上を示している。(ABCL/R では *Fib*(12) の計算に 12 分以上かかったが、ABCL/R2 では 22 秒であった。)

- C 言語 + LWP ライブラリでは、オブジェクト数が増加すると極端に遅くなる。スレッドの割り当てにかかる費用が大きいためだと思われる。また、オブジェクト数が少ない場合でも非自己反映的オブジェクトは、これと同等の速度を持っていることが分かる。

また、TimeWarp や並列度制御といった実際の自己反映的プログラミングでは、必要なもののみを自己反映的オブジェクトを用いて、他の部分には非自己反映的オブジェクトを用いることで、自己反映計算によるオーバーヘッドを局所化できることが分かっている。

6 まとめと今後の予定

我々の ABCL/R2 の実現では、部分コンパイルによるスクリプト実行の高速化、軽量オブジェクトや自己具体化などによる実行時の費用の低減により、従来の解釈実行方式による実現に比べて大幅な効率化を達成した。また、非自己反映的部分の実行は、自己反映計算の機能のない言語と同等の性能を持っている。

現在、逐次 Common Lisp 上で動作する疑似並列版の ABCL/R2 システムが、`camille.is.s.u-tokyo.ac.jp` (`133.11.12.1`) から anonymous ftp 可能である。使用できる Common Lisp としては、Kyoto Common Lisp(KCl), AKCl, CMU Common Lisp, Sun Common Lisp, Macintosh Common Lisp などがある。

今後の予定としては、(1) 自己反映計算のための解釈実行を除去する、(2) より効率のよい実行時システムを用いることで、実用的な効率を目指す、(3) いろいろな自己反映的プログラミングを通じて、よりよい言語モデルを提案する、といったことが挙げられる。特に (1) については、動的なコンパイルや、部分計算の枠組の適用などが考えられる。

参考文献

[1] Jim des Rivières and Brian Cantwell Smith. The implementation of procedurally reflective languages. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 1984.

[2] Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. RbCl: A reflective object-oriented concurrent language without a run-time kernel. In *Proceedings IMSA'92 International Workshop on Reflection and Meta-level Architecture*, Tokyo, November 1992.

[3] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, Massachusetts, 1991.

[4] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA'87*, volume 22, pages 147–155. SIGPLAN Notices, ACM Press, October 1987.

[5] Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1992. To appear.

[6] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *Proceedings of ECOOP'91*, number 512 in Lecture Notes in Computer Science, pages 231–250. Springer-Verlag, 1991.

[7] Brian C. Smith. Reflection and semantics in Lisp. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 23–35. ACM Press, 1984.

[8] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of OOPSLA '88*, volume 23, pages 306–315. SIGPLAN Notices, ACM Press, September 1988. (Revised version in [10]).

[9] Takuo Watanabe and Akinori Yonezawa. An actor-based metalevel architecture for group-wide reflection. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages (REX/FOOL), Noordwijkerhout, the Netherlands*, May 1990. also number 489 in Lecture Notes in Computer Science. Springer-Verlag, 1991.

[10] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System.* Computer Systems Series. The MIT Press, 1990.

[11] Akinori Yonezawa and Takuo Watanabe. An introduction to object-based reflective concurrent computations. In *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, volume 24, pages 50–54. SIGPLAN Notices, ACM Press, April 1989.