プロセスとチャネルに基づく並列言語のための部分計算

細谷晴夫　増原英彦　田浦健次朗　米澤明憲
E-mail: {haruo, masuhara, tau, yonezawa}@is.s.u-tokyo.ac.jp
東京大学理学系研究科情報科学専攻

部分計算は、パラメータの一部に既知の値を与えてプログラムを特化し、効率を向上させる手法である。これまで逐次言語の部分計算の手法が多く提案されて、その有効性が示されてきた。しかし、並列言語の部分計算の研究はあまり多くはない。本稿では、プロセスとチャネルに基づいた並列計算モデルを対象とした部分計算を提案する。この計算モデルは、単純で well-defined なセマンティクスを持ち、かつ並列オブジェクト指向言語のようなより高レベルな並列言語を実現するに十分強力である。さらに部分計算時にできるだけ非決定性を決定するというアプローチをとることにより、より効果的な部分計算ができることを示す。

Partial Evaluation for a Concurrent Language based on Processes and Channels

Haruo HOSOYA　Hidehiko MASUHARA　Kenjiro TAURA　Akinori YONEZAWA

Department of Information Science, University of Tokyo

Partial evaluation is a technique to improve efficiency by specializing a program with respect to part of its parameters whose values are known. While many partial evaluation techniques for sequential languages have been proposed and their effectiveness has been proved, very few techniques exist for concurrent languages. In this paper, we propose a partial evaluation for a concurrent calculus based on processes and channels. The calculus has simple and well-defined semantics, yet is powerful enough to encode higher-level languages such as concurrent object-oriented languages. Moreover, we show that if any non-determinism in a program is allowed to be resolved at partial evaluation time, the partial evaluation will be more effective.

1 はじめに

部分計算は、プログラムを部分計算時に値が既知なパラメータに関して特化したプログラムを自動的に生成する手法である。これまで関数型、もしくは手続き型の逐次言語において部分計算の手法が多く提案されており、効率を大幅に向上させてその有効性を示してきた。[3,11]。

並列言語においては、部分計算を行うことによる利益はさらに大きいと予想される。これは、並列プログラムにおける計算パターンを露出することでより低レベルな静的最適化の効果を向上させたり、通信を最適化したりすることなどができると期待できるからである。しかし現時点では、並列言語の部分計算の研究はあまり多くはない。

我々は、並列オブジェクト指向言語など明示的に並列性を記述するような並列言語に対する部分計算を一般的に（つまり特定の言語のコンストラクトからは独立して)、かつ見通しよく論じたい。そこで、プロセスがチャネルを介して通信しあいながら並列に計算を行う計算モデル[4,10]を部分計算を論じる対象とする。これらの計算モデルは、単純でかつ、さまざまな並列言語のコンストラクトを実現するに十分強力である。実際このような単純な計算モデルの上で並列オブジェクト指向言語のさまざまなコンストラクトが自然に実現できること[5]や、解析・最適化を行うことにも適していること[6]が示されている。一般の並列言語に対して部分計算を行うにはまずこのような計算モデルに基づく言語に変換してから行えばよい。このような設定のもう一つの利点は、厳密でかつ単純なセマンティクスがこれらの言語に定義されているため、部分計算の正当性が論じやすいことである。本稿では、そのような並列な計算モデルの一つである、HACL[4]のサブセットを対象とした部分計算を提案する。

並列言語の部分計算を行う際に問題となるのが、非決定性の扱いである。具体的に我々の対象言語における非決定性には、次のようなものがある。

- 並列実行の非決定性：どのプロセスが先に実行されるかは分からない
- 通信の非決定性：同じチャネルに対してどの送信プロセスと受信プロセスが先に通信が行われるかは分からない

特に、同じチャネルに対し複数の送信プロセス、受信プロセスが存在し得るため、通信における非決定性が問題となる。

このような並列言語で、非決定性を厳密に保存するような部分計算を行うとすると、効果的に部分計算できない場合がある。例えば、部分計算時の情報から、次のPとQ２つのプロセスが同じチャネルmに対して送受信することが分かったとする。

```
P:      (send m v)
Q:      (receive m x ...)
```

この２つのプロセスの通信を部分計算時に実行して除去してもよいためには、このmに対する他の送信または受信プロセスが実行時に存在しないことがいえなければならない。しかし部分計算時に、他の送信・受信プロセスがあることが分かったり、もしくはmがどこで使われるか特定できなかったりすると、この通信を部分計算時に実行することはできなくなる。

そこで我々は、並列プログラムが含む非決定性の中でも、部分計算時の情報のみで実行可能なものが一つでもあれば、それに決定するというアプローチをとる。非決定性を持つ言語の意味としては、上記のプログラムは「PがQと通信しても他のプロセスと受信しても正しい実行」なので、ありうる実行の一つとしてPとQの通信を部分計算時に実行する。非決定性を厳密に保存しようとすれば除去できない上の例のような通信も、このアプローチでは除去可能である。すると送信された値に基づいてさらに部分計算を続けることができるため、より効果の高い部分計算が可能になる。

本稿で示す部分計算器は、従来の逐次言語のための部分計算器を基本として、非決定性を部分計算時に決定する手続きを拡張したものと見ることができる。基本的には、言語の意味に即して解釈実行するインタプリタに、「未知の値」を扱えるように拡張したものであり、また部分計算時の情報によって部分計算の挙動を決めるonline[3]と呼ばれる方式をとっている。基本的な実装の方針は、既存の逐次言語 Schemeのためのonline部分計算器Fuse[11]の技術に基づいている。また工夫点として、除去できなかった受信の後のプログラムも部分計算することや、具体化されていないチャネルも変数名で識別できるという性質を利用することなどがある。

現在、上述の並列言語のための部分計算器のプロトタイプがほぼできており、簡単なプログラムについて部分計算が正しく行われていることを確認した。

本稿の残りでは、2節では対象言語の文法とセマンティクスを説明する。0節では我々の部分計算のアルゴリズムを述べる。4節では並列言語の部分計算の正当性について触れる。5節では関連研究、6節ではまとめと今後の課題を述べる。

2 対象言語の文法とセマンティクス

本稿で述べる部分計算の対象言語は、明示的に並列性を記述する、プロセスとチャネルに基づく並列言語HACLから、関数やchoiceを取り除いたサブセットである。

チャネルは、プロセスが他のプロセスと通信を行うための通信媒体である。チャネルに対する送信と受信のコンストラクトを用意する。チャネルには次のような性質がある。

- 同じチャネルに複数の送信プロセス、受信プロセスが存在し得る
- 一度通信を行った送信プロセス、受信プロセスは消滅する
- チャネルは一級の値で、パラメータとして渡したりデータ構造に保存することができる

言語の文法を図 1に示す。Pはプロセスであり、再帰的にプロセスが定義されている。この言語はStandard ML of New Jerseyのコンパイラにおける Continuation Passing Style[1]と類似したコンパイラの中間コードの形をしており、一つの式（またはプロセス）はただ一つの機能を持つ。すなわち、引数やチャネルの位置にあるxやvは変数かアトミックな定数に限られる。プロセス自体は値を持つわけではなく、計算した結果の値はチャネルを通じて次の計算のステップへ渡す。引数および、fixで定義できるプロセスは実際には複数個書けるが、図 1では簡単のため一つとしている。

HACLの厳密なセマンティクスは、[4]に遷移規則として記述されているが、ここではそれぞれのコンストラクトの意味を直感的に説明する。|で区切られたプロセスは並列実行を表す。$x.Pは、チャネルxを作ってプロセスPを実行する。x(v)はチャネルxに値（またはメッセージ）vを送信する。x(y)=>Pはチャネルxから値を受信して（値が来るまで待ち）、その値を変数yにバインドしてPを実行する。プロセスTはすべての計算が終了させる。_は何もせずに終わるプロセスである。

```
P ::=P|P|...|P                    並列実行
    $x.P                        チャネル生成
    x(v)                      チャネルに送信
    x(y)=>P                 チャネルから受信
    T                         プログラム終了
    _                           何もしない
    let x=op(v,...) in P      プリミティブ
    if v then P else P          条件分岐
    fix f(x)=P in P           プロセス定義
    f @ (v)                   プロセス生成
```

図 1: 対象言語の文法

```
(define (fib n)
  (if (< n 2)
    1
    (let ((f1 (future (fib (- n 1))))
          (f2 (future (fib (- n 2)))))
      (+ (touch f1) (touch f2)))))
(fib 10)
```

図 2: 並列Schemeでかかれたfib

letはプリミティブ関数opを実行した後、Pを実行する。プリミティブ関数の呼び出しは一つのletに一つのみ書け、その引数には変数またはアトミックな定数のみがとれる。ifは分岐であるが、ここでも条件に式が入らない。fixは再帰的なプロセス定義で、ユーザ関数の定義に相当する。プロセス生成@はfixで定義されたプロセスを引数付きで生成するもので、関数呼び出しに相当するが、これ自体が値を持つわけではない。

さて、いろいろな高レベルな並列言語がこの言語に変換できる[1]。例えば、非同期な関数呼び出しやオブジェクトの排他制御など並列言語のコンストラクトは、チャネルを用いて表現される。また逐次的な実行ではCPS変換[1]と同様の変換を行う。このときチャネルがコンティニュエーションの代わりをする。例として、並列 Schemeで書かれた図 2の並列フィボナッチ数計算がどのように我々の言語に変換されるかを示す。

```
(future (fib (- n 1)))
```

は、(fib (- n 1))を並列に実行し、その答えを待つ代わりに、計算が終わり次第答えが入るような「箱」を返す。その箱から実際に値を取り出すにはtouchを使う。このプログラムを変換すると、図 3にあるようなコードが得られる。fibはnのフィボナッチ数を計算してチャネルrに答えを返す。fibは中で、再帰的にfibをn-1とn-2についてそれぞれ並列に実行するが、その際に新しくチャネルsとtをつくってfibに渡す。そして、答えがそれぞれチャネルsとtに返ってくるまで待ち、返ってきたら2つの答えを足してfibの答えとしてチャネルrに返す。チャネルsとtが、2回のfibの呼び出し後のコンティニュエーションの役割を果たしていることが分かる。

[1] 実際、我々の研究室では並列 Schemeのコンパイラの中間コードとしてこの言語を使うことが行われている。

```
fix fib(n,r) =
  let y=(x<2)
    in if y
       then r(1)
       else $s.$t.let l=n-1 in fib(l,s)
                | let m=n-2 in fib(m,t)
                | s(x)=>t(y)=>let w=x+y
                                in r(w)
in $r.fib(10,r)
```

図 3: fib を変換した例

3 部分計算のアルゴリズム

部分計算とは、プログラムの入力または自由変数などのパラメータに特定の値を与えると、それを部分計算時に静的な値として特化されたプログラムを自動生成する手続きである[3]。

本稿で述べる部分計算は、並列プログラムにおける非決定性の中で、部分計算時の情報のみで実行可能なものが一つでもあれば、それに決定するというアプローチをとる。これは、部分計算の効果を向上させるということ以外に、非決定的な実行のうちどれを実行しても並列言語の意味としては正しいという考えに基づいている。

しかし、並列プログラムでもスケジューリングの fairness に依存したプログラムに対しては、期待される実行が部分計算によって失われる可能性がある。例えば、探索問題で、投機的に並列探索をして最も早く終わったものを解とするようなプログラムを我々の部分計算にかけると、実行時に投機的な実行が行われなくなる可能性がある。したがって、本稿ではそのようなプログラムは想定せず、どのようなスケジューリングが行われても期待する答えが出るようなプログラムのみを扱うものとする。

本稿で示す部分計算のアルゴリズムは、従来の逐次言語のための部分計算の手法を拡張して、非決定性を扱えるようにしたものと見ることができる。並列実行の非決定性、および通信の非決定性の 2 種類は、補助関数群（図 4）で決定する。非決定性の決定以外に工夫点として、

- 通信ができずに残った受信プロセス `x(y)=>P` について、受信後にあたる内側の式 `P` を再び部分計算する
- 名前でチャネルを識別することにより、部分計算時に生成されていないチャネルについても同値性を判定する

ということがある。

部分計算器本体 *pEval*（図 5）は従来の手法に基づいている部分である。基本的にはインタプリタを「未知の値」も扱えるように拡張し、部分計算時に計算可能な式は計算し、未知の値があるために計算できない式は、その計算を行うようなコード（**residual コード**という）として残す。

我々の言語では、プログラムの実行は「並列に実行可能なプロセスの集合」の遷移として表される。部分計算器は、このプロセスの集合を「状態」として、状態を引数に渡す形をしている。*pEval* および補助関数群の引数に現れる *state*（または組<*ready*, *pending*>）がプロセスの集合を表しており、*pEval* の呼び出し一つが一つの遷移に相当すると考えてよい。

変数名と値のバインディングを保持する環境のようなものは用いず、代わりに変数への代入を使用する。例えば、`P[v/x]` は `P` に自由に出現する変数 `x` を `v` に置き換えるという意味である。

以下、補助関数群と *pEval* についてそれぞれ説明する。

3.1 補助関数群

我々の部分計算器で特徴的な「部分計算時の非決定性の決定」は、補助関数群（図 4）に記述する。非決定性には (a)並列実行の非決定性、(b)通信の非決定性の 2 種類があり、それぞれを決定するために以下のような（プロセスを集めた）データ構造および補助関数を用意する。

(a)並列実行の非決定性を決定するためのデータ構造および補助関数はそれぞれ、*state*（=<*ready*,*pending*>）の要素 *ready*、および補助関数 *Fork*, *Schedule* である。*Fork* は並列実行可能なプロセスを *ready* に挿入する。*Schedule* は *ready* にあるプロセスから一つ取り出し *pEval* にかける。もし *ready* にプロセスがなければ、実行できるプロセスはもはや存在しないので、*pending* に残ったプロセスに関する処理を *Post_peval* に委ねる。

(b)通信の非決定性を決定するためのデータ構造および補助関数はそれぞれ、*state* の要素 *pending*、および補助関数 *Send*, *Receive* である。*Send* は、与えられた送信プロセスに対し、同じチャネルでメッセージを待っている受信プロセスを *pending* から一つ取り出し、それらの通信を実行する関数 *Commun* に渡す。もし対応する受信プロセスが存在しない場合は、与えられた送信プロセスを *pending* に挿入し、*Schedule* に処理を移す。*Receive* も、与えられた受信プロセスについて *Send* と同様のことを行う。

上記の手続きにおいて、次の点が部分計算時の非決定性の決定に相当する：

- *ready* および *pending* から取り出せるプロセスが一意でないにも関わらず、そのうち一つだけを取り出す
- *Send* および *Receive* で、送信プロセスと受信プロセスがそろった時点で、すぐに通信を成立させて実行を始める

Schedule にて *ready* が空になり、実行できるプロセスがなくなると、*Post_peval* が呼ばれる。*Post_peval* では、この時点で *pending* に残っている通信できなかった送信プロセスおよび受信プロセスのうち、受信プロセスについて、受信後にあたる内側の式も再び部分計算する。この受信プロセスは、受信の時には他にどのようなプロセスが存在するか分からないため、引数を未知、*state* を空として、再び部分計算するという保守的な方法をとっている。例えば次のプログラムを部分計算するとする。

```
n(a) | m(x)=>((n(y)=>P)|n(b))
```

まずは、`n(a)` と `m(x)=>...` の両プロセスは通信できないので残る。次に *Post_peval* が後者の内側 `((n(y)=>P)|n(b))` を部分計算する。このとき `n(y)=>P` が外側の `n(a)` と通信し得るかは、実行時に `m` に送信がある前に `n(a)` が消費されるかどうかに依存するため、部分計算時には判断できない。これらの残った受信プロセスに関する処理は、関数型言語の部分計算でλ式が残るとその内側も部分計算することに相当する。最後に、部分計算された受信プロセスも含めて、残ったプロセスを並列コンストラクトで結合する。

Send および *Receive* で、「同じチャネルかどうか」をどう判断するかが問題である。チャネルが生成されると、その度にスコープ内に現れるそのチャネル変数が一意な名前で置き換えられる。しかしチャネルがそのように具体化されなくても、同じスコープで同じ変数名のチャネルで受信プロセスと送信プロセスがある場合は、それらは通信し得ると考えられる。そこで我々の部分計算器では、改名されたチャネルの名前、および（同じスコープなら）プログラム上の変数名の両方でチャネルの同値性を判断する。上記の例では、内側にある `n(y)=>P` と `n(b)` のチャネルは具体化されていないが、同じスコープで同じ名前の変数なので、これらのプロセスは通信させてもよいわけである。

3.2 部分計算器本体

部分計算器本体 *pEval*（図 5）は従来の部分計算に基づいている部分で、インタプリタを「未知の値」も扱えるように拡張したものである。*pEval* は、ターゲットのプロセスと *state* を受け取り、部分計算を行い、その結果を並列なプロセスの集合として返す。ここで部分計算が扱う値として記号値(sval)を導入する。記号値は、本来値を持つような式を表し、定数、未知な値を表す自由変数、プリミティブの呼び出し、λ式がある。

```
Fork <ready, pending> procs =
    Schedule <ready ∪ procs, pending>

Schedule <ready, pending> =
    if P ∈ ready
    then pEval [P] <ready - {P}, pending>
    else Post_peval pending

Post_peval [P1,...,Pn] =
    let [P1',...,Pn'] =
        map (λQ.if Q = x(y)=>P
                then let P'=pEval P empty_state
                     in x(y)=>P'
                else Q) [P1,...,Pn]
    in [P1'|...|Pn']

Send <ready, pending> [x(y)] =
    if x(v)=>P ∈ pending
    then Commun [x(y)=>P] [x(v)]
                <ready, pending - {x(v)=>P}>
    else Schedule <ready, pending ∪ {x(y)}>

Receive <ready, pending> [x(y)=>P] =
    if x(v) ∈ pending
    then Commun [x(y)=>P] [x(v)]
                <ready, pending - {x(v)}>
    else Schedule <ready, pending ∪ {x(y)=>P}>

Commun [x(v1,...,vn)] [x(y1,...,yn)=>P] state =
    pEval [P[v1/y1, ..., vn/yn]] state
```

図 4: 補助関数群

pEval の中は一つの大きな case 式になっていて、各言語のコンストラクトに関して分岐している。並列実行と送信・受信プロセスについてはそれぞれ補助関数 *Fork*, *Send*, *Receive* をそのまま呼ぶ。プログラムの終了を表す `T` については、これ以上何も実行せずに `T` を返す。何もしないプロセスを表す `_` については、別のプロセスを実行させるために *Schedule* が呼ばれる。

チャネル生成では、別のスコープの同じ変数名のチャネルと区別するために、スコープ内の自由な変数 `x` をグローバルに一意な新しい名前に改名する。

この先は逐次言語の部分計算の手法とほぼ同じである。`let` は、プリミティブ関数の実行をするものである。引数がすべて既知の値の場合は計算し、もし引数に未知な値が含まれていたらプリミティブ呼び出しの記号値を作る[2]。`if` は、もし条件値が既知なら、その値の真偽に基づいて選択した分岐を部分計算する。

[2] 部分的に既知なデータ構造[3]はまだ扱っていない。

```
pEval proc state =
    case proc of
    [P1|...|Pn]     =>  Fork state [P1,...,Pn]
    [x(v)]          =>  Send state proc
    [x(y)=>P]       =>  Receive state proc
    [T]             =>  [T]
    [_]             =>  Schedule state
    [$x.P]          =>  let x' = newname()
                        in pEval [P[x'/x]] state
    [let x=op(v1,...,vn) in P]
                    =>  if known(v1, ..., vn)
                        then let x' = eval_prim(op, [v1,...,vn])
                             in pEval [P[x'/x]] state
                        else let x' = sval_prim(op, [v1,...,vn])
                             in pEval [P[x'/x]] state
    [if v then P1 else P2]
                    =>  if known(v)
                        then if v=true
                             then pEval [P1] state
                             else pEval [P2] state
                        else let P1' = pEval [P1] state
                                 P2' = pEval [P2] state
                             in [if v then P1' else P2']
    [fix f(x) = P in Q]
                    =>  letrec f' = sval_lambda(x, P[f'/f])
                        in pEval [Q[f'/f]] state
    [f @ v]         =>  if known(f)
                        then if unfold?
                             then let sval_lambda(x,P) = f
                                  in pEval [P[v/x]] state
                             else let f' = specialize(f,v)
                                      R = Schedule state
                                  in [f' @ v|R]
                        else let R = Schedule state
                             in [f @ v|R]
```

図 5: 部分計算器本体

もし条件値が未知なら、両分岐をそれぞれ部分計算した後 if 式を再構成する。ここで *state* が二回 *pEval* に用いられているため、コードの複製が起こる可能性がある[3]。

fix は、再帰的なプロセスの定義である。定義されたプロセスに対して λ 式の記号値を作り、それをその λ 式自身の本体[4]と fix の本体の両方に代入を行う。@ は、fix で定義されたプロセスを生成するもので、逐次の関数適用と同様のことを行う。すなわち、f が既知、つまり λ 式の記号値の場合、まずその適用を unfold（インライニング）するか residualize（呼び出しを残す）するかを何らかの手段で決断する[5]。unfold なら通常の関数適用のように適用を行う。residualize なら、まず適用するプロセス定義を呼び出しの引数に関して特化したものを作り、それへの呼び出しを residual コードとして残す。一方、もし f が未知の場合は必ず residualize する。

プログラム P の部分計算は、初期状態を *initial_state* (=<ϕ,ϕ>) として、

pEval [P] *initial_state*

を実行することにより開始する。

部分計算中、同じ式が変数の複数の出現に代入されると計算の複製が起こり得るが、部分計算の結果はグラフ構造になっており、同じ計算は一つのノードで表

[3] このコードの複製では、計算の複製は起こらない。また、複製によって reduction が可能になることもあり、必ずしも悪いことではない。

[4] letrec を使っていることから分かるように、ここで cycle ができ得る。

[5] 現時点ではユーザの指示によって決めている。

されているので、後処理で hoisting して解決する。詳しい手法は逐次言語のための部分計算器 Fuse[11]を参考にされたい。

3.3 部分計算の例

上述の部分計算器が有効であることを、図 6の counter のプログラムを用いて例解する。一部見やすさのために、k-1 など式の中に式が含まれているが、本来は let がある。

counter で定義されたプロセスは、チャネル add にメッセージ(y,n)を受信する度に、状態値 x を y+x で更新していくプロセスである。それと同時に、チャネル n にその状態値を送信する。k_send は、counter の値を b でインクリメントするメッセージを並列に k 回チャネル add に送る。その結果は n にそれぞれ送ってもらう。k_recv は n に値が k 回送られるまで待ち、そのあとチャネル 1 にメッセージを送る。

一方これらを使用する側は、まずチャネル add、初期状態値 a で counter プロセスを生成する。そして add に並列に k 回インクリメントのメッセージを送り、同時に n に k 回値が送られるまでチャネル 1 で待つ。このとき counter における n への送信は k_recv ですべて消費されている。最終的な counter の値を得るために、もう一度 counter の値を b でインクリメントして、その答えを n で待ち、その値をこのプログラムの結果としてチャネル p に送る。

さてこのプログラムを、自由変数 a と b を未知、k を静的に 2 として、前述の部分計算器にかける。すると、図 7のようなプログラムが得られる。(冗長な式 (b+(b+(b+a)))は後処理で hoisting される。)

この例では、初め 2 回の add(b,n)はどちらが先に

```
fix counter(add,x)=
      add(y,n)=>let x'=y+x
               in counter @ (add,x')
                  |n(x')
    k_send(k,add,b,n)=
      if k=0 then _
      else add(b,n)
           |k_send @ (k-1,add,b,n)
    k_recv(k,add,n,l)=
      if k=0 then l()
      else n(y)=>k_recv@(k-1,add,n,l)
in $add.$n.$l.counter @ (add,a)
           |k_send(k,add,b,n)
           |k_recv(k,add,n,l)
           |l()=>(add(b,n)
                  |n(z)=>p(z))
```

図 6: counter の例

```
add(y,n)=>let x'=(y+(b+(b+(b+a))))
        in counter @ (add,x')
           |n(x')
| p(b+(b+(b+a)))
```

図 7: counter を部分計算した結果

メッセージを送るかは非決定的である。しかしながらどれを先に実行しても、counter の値が結局 3 回 b でインクリメントされることに違いはない。もし非決定的なところを部分計算時に決定することを放棄すれば、上記のようなプログラムは、通信を除去できないし、さらにそのためにほとんどの式が残ってしまい、部分計算の効果はほとんどなくなってしまう。

4 部分計算の正当性について

非決定性を持つ言語では、プログラムの結果が一意に決まるとは限らない。したがって、逐次言語の部分計算器の正当性「元のプログラムと residual プログラムが同じ入力に対して同じ結果を出す[3]」とは別の定義が明らかに必要である。

我々の並列言語では「結果」とは、外部とのメッセージのやり取りとして定式化される。すると、直感的には並列言語の部分計算の正当性を次のように説明できる。すなわち、任意のプログラム P に対し、P を部分計算器にかけて得られた residual プログラム Q について次の関係がなりたつこと：

1. P では見えない外部とのやり取りが、Q でも見えない
2. P で必ず見える外部とのやり取りが、Q でも必ず見える

本稿で示した部分計算器における非決定性の扱いが、これらに相当することは予想できるが、正当性の形式的定義と証明は今後の課題である。

ちなみに、非決定性を厳密に保存する部分計算を考える場合は、testing equivalence[9]や bisimulation[8]を用いることが必要である。

5 関連研究

我々の部分計算器と手続き型言語の部分計算器の類似点は、システムの状態を保持しながら部分計算を行うという点である。手続き型言語においてシステムの状態とは、変数のバインディングやメモリの状態である。一方我々の言語では、システムの状態はプロセスの集合である。

逐次型言語では、副作用の扱いが部分計算の構成を難しくしているといえる。我々の言語ではそのような

副作用がチャネルを用いて表されるため、このような言語を選ぶことにより、部分計算としては副作用をチャネルとして統一的に扱えるという利点がある。副作用がチャネルで表せる例としては、書き換え可能データは図 6の counter プロセスのようにチャネルと状態を保持するプロセスを用いて表せ、入出力は外部のチャネルへのメッセージとして表せる。

また、並列論理型言語の部分計算がすでにいくつか提案されている[2]。しかしこれらは非決定性を厳密に保存する立場を取っている。

並列言語と部分計算に関する研究にはリフレクティブ並列オブジェクト指向言語 ABCL/R3[7]があるが、これはメタレベルの言語を関数的なものに限定して、オブジェクト間の通信は入出力としてただ残している。このような通信も最適化できるような枠組みを作ることも、我々の目標の一つである。また部分計算の対象として、より一般的な枠組みを用いることにより、部分計算の手法の見通しがよくなると考えている。

6 まとめと今後の課題

本稿では、プロセスとチャネルに基づく並列言語の部分計算の手法を提案した。並列プログラムの非決定性を部分計算時に決定するアプローチをとることにより、より効果の高い部分計算ができることを示した。

今後は、この部分計算器が本稿で述べた正当性を満たすこと、またより大きなアプリケーションに適用し、性能評価を行って有効性を実証することが課題である。

謝辞

小林直樹氏、浅井健一氏、Rajeev Surati 氏、松岡聡氏、高橋俊行氏、その他米澤研究室の皆様には、数々の議論と貴重なアドバイスをしていただき深く感謝いたします。

References

[1] Appel, A. W., *Compiling with Continuations*, Cambridge University Press, 1992.

[2] Fujita, H., A. Okumura, and K. Furukawa, "Partial Evaluation of GHC Programs Based on the UR-set with Constraints," *in proceedings of Logic Programming: Fifth International Conference and Symposium*, pp. 924-941, 1988.

[3] Jones, N. D., C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice Hall, 1993.

[4] Kobayashi, N. and A. Yonezawa, "Higher-Order Concurrent Linear Logic Programming," *in proceedings of International Workshop TPPP '94, Sendai*, pp. 137-166, 1994.

[5] Kobayashi, N. and A. Yonezawa, "Type-Theoretic Foundations for Concurrent Object-Oriented Programming," *in proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '94)*, pp. 31-45, 1994.

[6] Kobayashi, N., M. Nakade, and A. Yonezawa, "Static Analysis of Communication for Asynchrnous Concurrent Programming Languages," *to appear in proceedings of International Static Analysis Symposium, Springer LNCS*, 1995.

[7] Masuhara, H., S. Matsuoka, K. Asai, and A. Yonezawa, "Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation," *to appear in proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1995.

[8] Milner, R., J. Parrow, and D. Walker, "A Calculus of Mobile Processes, I, II," *Information and Computation*, vol. 100, pp. 1-40, 1992, September.

[9] Nicola, R. D. and M. C. B. Hennessy, "Testing Equivalence for Processes," *Theoretical Computer Science*, vol. 34, pp. 83-133, 1984.

[10] Pierce, B. C. and D. N. Turner, "Concurrent objects in a process calculus," *in proceedings of Theory and Practice of Parallel Programming (TPPP), Sendai, Japan*, pp. 187-215, 1995.

[11] Ruf, E., *Topics in Online Partial Evaluation*, PhD thesis, Stanford University, 1993. (CSL-TR-93-563)