理論領域と実用領域を結ぶ
新しいプログラミング単位

「機能と構成」領域　　河野　真治

概　　要

実用領域では、JavaやC++などの複雑なプログラミング言語や、高度なGUIが導入され、ハードウェアを含む組込みシステムなどの複雑な開発を迅速に行なうことが要求されている。一方で、理論領域でも、モデル検査やCPS変換によるプログラム理論の詳細化などが進んでいる。しかし、その双方を接続する方法は少ない。我々は、これを継続を基本とした新しいプログラミング単位を導入することで解決したいと考えている。

1．研究のねらい

これまで、主に情報を隠すため、プログラムを分割するために関数呼び出しや制御構造、オブジェクト指向プログラムなどが導入されて来た。一方で理論の方もさまざまな進歩があり、特に、最近のソフトウェアの信頼性に対する要求から、さまざまな手法が開発されている。特に、定理証明や、充足可能性を検査するモデル検査の手法が導入され、実際のプロトコルやプログラムなどに応用されている。

これらの両方を結びつけるプログラミング単位を従来のプログラム言語に対応した形で、提供できれば、理論領域の成果を、既に書かれたソフトを大量に持つ実用領域に結びつけることが容易になると考えられる。

2．研究方法と成果

2．1　継続を基本とした新しいプログラム単位（code segment）

ここで導入する単位はステートメントよりも大きくサブルーチンよりも小さい単位である。ループ構造を持たないステートメントの集合であり、コンパイラでは基本ブロックと呼ばれるような部分に相当する。これをプログラム単位としてプログラム言語的に明示的に使用するのが、ここでの新しい提案である。この単位をcode segmentと呼ぶ。codesegmentは継続（continuation）で接続される。継続への移動は引数付きのgoto文（parameterized goto statement）で表現される。code segment自体は制限されたCのステートメントにより表現さ

れる。この言語をCbC（Continuation based C）と呼ぶ。継続を持つCに近い言語としては、C--が知られているが、CbCは、継続を基本とするところが異なる。

code segmentは入力interfaceから条件文によって分岐する複数の出力interfaceを接続する単位となっている。状態遷移系を直接に表現する単位となっている。（図1）

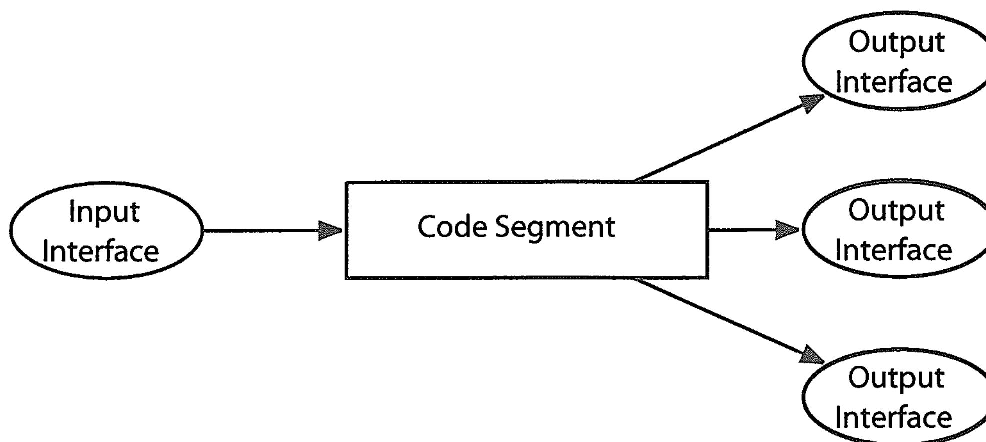

図1：code segment

以下の例は、CbCによる階乗の計算である。

```
code fact (int n, int result,
      code (*print) ()) {
    if (n>0) {
        result *= n;
        n--;
        goto fact (n, result, print) ;
    } else
        goto (*print) (result) ;
}
```

goto fact (n, result, print) ; は直接の継続であり、goto（*print) (result) ; は間接の継続である。その引数は、interfaceと呼ばれる。属するcodeと同じinterfaceを持つgoto文は、一つのjump命令にコンパイルされる。

CbCは、Cの下位言語であるが、Cのサブルーチンへ戻るための環境付き継続を導入するとCの上位言語となる。この言語をCwC（C with Continuation）と呼ぶ。CwCではCbCから通常のCの関数を呼び出すことができる。 CwCは現在、IA32用、PowerPC用のコンパイラが実装されている。

2.2　CbCの使い方

CbCはプログラム変換の基本単位として使うことを想定している。CbCのinterfaceを物理的なレジスタなどとの対応を定義することにより最適化そのものをCbCレベルで記述することも可能である。また、CPUの動作そのものを記述することも可能である。また、既存の言語からCbCに変換することで、既存のソフトウェア資産の解析ツールとして使うことが出来る。

Cからの変換はコンパイラによって行なわれるが、ループの変換は自明である。難しい部分は、関数呼び出し時のスタックの明示的な記述である。

```
j = g (i+3);
```

のようなCの関数呼び出しは、struct f_g0_saveなどの明示的なスタックの中身を表す構造体を用いて、

```
struct f_g0_interface *c =
   (struct f_g0_save *) (sp -=
   sizeof (struct f_g0_save)) ;
c = sp;
c->ret = f_g1;
goto g(i+3, sp);
```

のような形で、明示的なスタック操作に変換される。これは変換の一例であり、他の方法、例えばリンクトリストなどを用いても良い。f_g1は、関数呼び出しの後の継続であり、gでは、

```
code g (int i,stack sp) {
   goto (* ((struct
      f_g0_save *) sp)->ret)
         (i+4, sp);
}
```

のように間接的に呼び出される。スタックの中は、継続と中間変数などを格納する構造体である。スタックそのものは、これらの構造体を格納するメモリである。

これらの変換は、機械的に行なうことが可能だが、変換の詳細は若干複雑であり変数の種類や寿命などを含めた構文解析する必要がある。変換の結果は、callなどのアセンブラによる関数呼び出しのサポートを使えないために、元のCよりも早くなることはない。しかし、スタック操作が不要になる場合を検出し、それを取り除くことができれば、高速になる場合

がある。このような変形を行なった場合（Cbc-op）と、gccの最適化を行なった場合の実行時間の比較を行なってみた。以下のような三段の関数呼び出しについて10^7程度の繰り返しを例題として用いた（表１）。

```
f0(int i) {
    int k,j;
    k = 3+i;
    j = g0(i+3) ;
    return k+4+j;
}
g0(int i) { return h0 (i+4) +i; }
h0(int i) { return i+4; }
```

CwCコンパイラでの同程度の最適化を見るとCからCbCへの変換後のソースはPentium Ⅲで12.5%、PowerPCで42.9%程度遅くなることがわかる。スタック操作を削除する変換により、元のCの関数呼び出しよりも50%程度高速になる。これは、gccで最適化を行なわない場合よりも高速である。gccは、-O6による最適化の効果が大きく、スタック操作を削除したCwCコンパイラによるものよりも高速な実行となる。特にPowerPCでのgccの最適化の効果が大きい。これは、h0 (i) のような関数呼び出しはリンクレジスタのみを使うスタック操作を含まない形にコンパイルされるためである。

1GHz Pentium III Lunix

ソース	コンパイラ	実行時間sec
C	gcc 2.95.3	0.430
C	gcc 2.95.3 -O	0.370
C	gcc 2.95.3 -O6	0.240
C	CwC	0.640
CbC	CwC	0.720
CbC-op	CwC	0.330

1GHz PowerPC Mac OS X

C	gcc 3.1	0.960
C	gcc 3.1 -O	0.470
C	gcc 3.1 -O6	0.260
C	CwC	1.190
CbC	CwC	1.700
CbC-op	CwC	0.630

表１：CbCに変換した結果のベンチマーク

2.3　仕様記述としての使用法

仕様記述としては一般的には時相論理などが使われることが多いが、$\Box(p \rightarrow \Diamond q)$ などの意味は決定的なオートマトンとして定義することが可能である。このオートマトンをCbCによって記述することにより、仕様記述として用いることが出来る。

仕様の検証手法としては、実装記述とともに同時に仕様記述である決定的オートマトンを走らせるassert的な使い方が考えられる。また、CbCのinterfaceの状態を有限な状態に抽象化することができれば、実装記述とともにタブロー展開などの手法で直接的に実装の正しさを証明することも可能である。

タブロー展開は、仕様とプログラムの大域的な状態をすべて生成する手法である。反例を探す場合は反例が見つかった場合に停止して良いが、証明を行なう場合はすべての大域的な状態を生成する必要がある。大域的な状態の生成は、初期状態から非決定的に生成されるすべての次の状態を生成すること（状態の展開）により行なわれる。証明にはプログラムの抽象化された状態の数に比例し、また、プログラムが含む変数の数の指数乗の計算量がかかる。

2.4　CbCと並列検証系

実用的なプログラムではプログラムの持つ抽象化された状態は有限であり、状態遷移に影響を与える変数の数もそれほど多くはないと予想される。このような場合には、並列計算機による力技的な検証系が有効であると考えられる。この研究では、以下のようなリダクション・マシン的な並列タブロー法を採用している。

状態は時相論理式の部分項を変数として持つ部分項BDD（Ordered Binary Decision Diagram決定二分木）を用いて表現され、部分項BDDの変数の順序を並列計算機全体で維持する単一の部分項サーバを持つ。状態の展開はワーカーと呼ばれる各計算機ノードで行なわれる。生成された状態は部分項BDDに変換される。変換された部分項BDDは、ハッシュ関数により各計算機ノードにランダムに分配される。分配はランダムなので十分な状態数があれば適切な台数効果が得られるはずである。（図２）

2.5　並列検証系のための通信ライブラリ

このような並列計算手法は、データは計算機ノードに分散し共有されないので、PCクラスタなどに向いた実装となる。ただし、この場合の通信は完全にランダムに行なわれる。PC クラスタ用の通信ライブラリはMPIなど、いくつかあるが完全結合的な通信を行なう場合を想定したライブラリではなく、部分的な通信を想定した実装が多いので、このような分

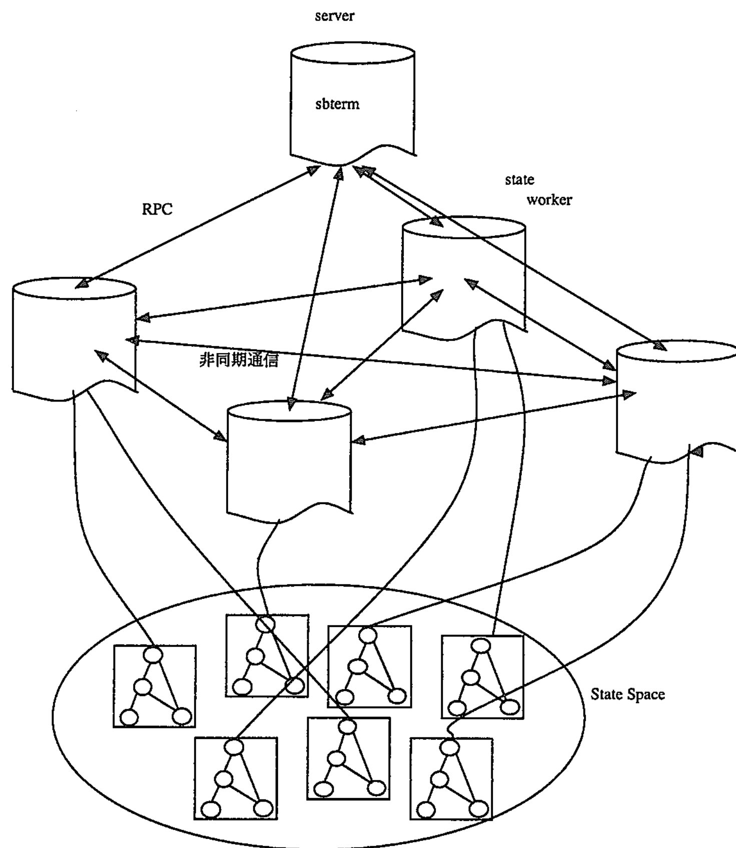

図２：並列検証例

散プログラムには不向きである。

そこで、完全結合向きのPCクラスタ通信ライブラリSuciを作成した。このライブラリは、Unixのデータグラム・ソケットを用いており、ユーザレベルでTCPの信頼性制御、フロー制御に相当する部分を行なう。これにより、スループット優先の通信とレスポンス優先の通信を持つこの並列検証系でも有効な通信を行なうことが可能である。いくつかのベンチマークでTCPベースのMPIの実装であるMPICHよりもより良好な結果を得ている（図３）。

並列検証系自体はPrologで記述されており、ITLに関する検証を行なうことができる。PrologはCで記述されたSuciライブラリを呼出して通信を行なう。

50台程度のPCクラスタで良好な台数効果が得られることを確認している。ここでは例題はDining Philosopher（６人）とUnixのマウスドライバを用いている（図４、図５）。Interleavingを多く含み状態数の多いDining Philosopherでは台数効果が得られやすい。一方で、変数の数の多いマウスドライバでは30台程度で十分な高速化が得られていることがわかる。実時間では43台で30秒、マウスドライバの場合で６分程度で終了している。

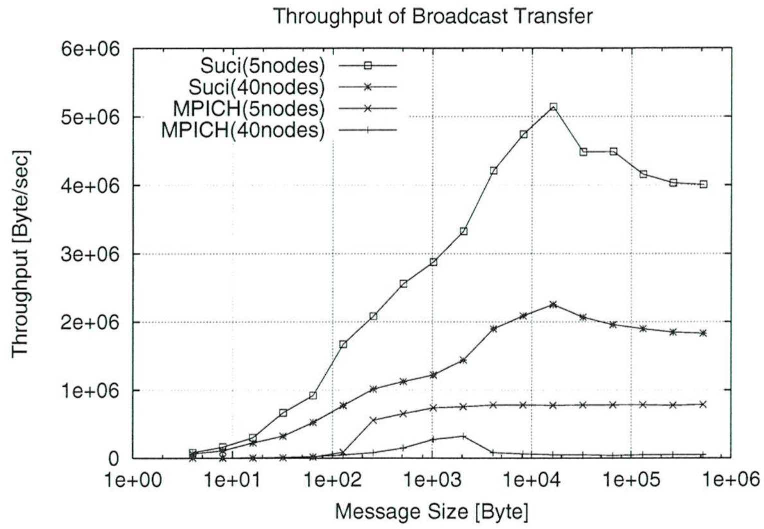

図３：Suciベンチマーク

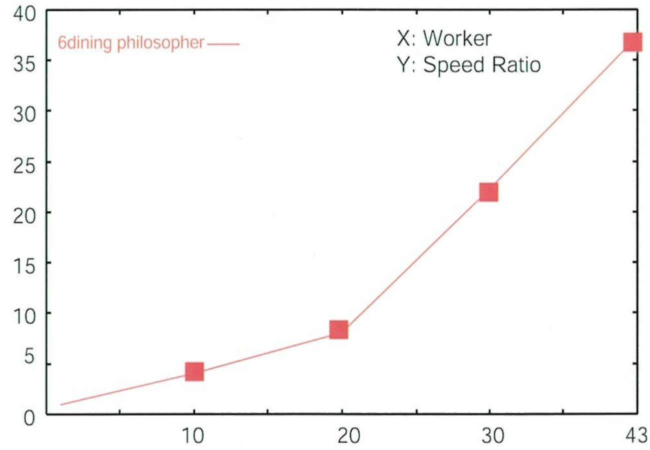

図４：並列検証系 Dining Philosopher 6

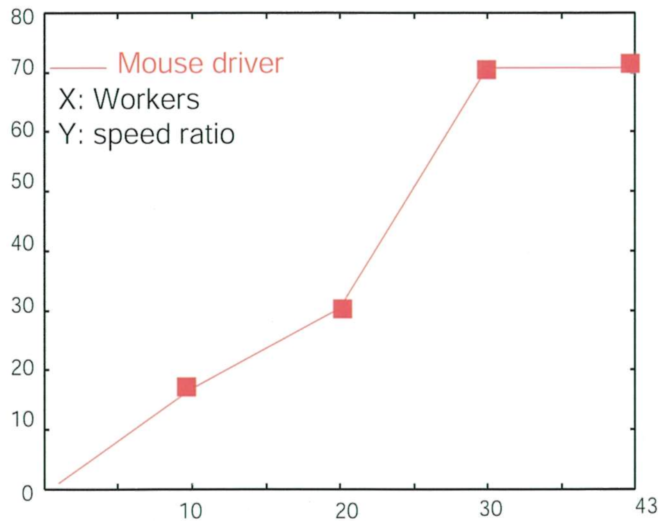

図５：並列検証系 Mouse Driver

３．まとめと今後の展開

この研究は、継続を基本とした新しいプログラム単位を提供することにより、プログラム変換やプログラム検証という理論的な成果を、既存のプログラム資産を多量に含む実用領域に適用することを目的としたものである。

新しいプログラム単位は、CbC/CwCコンパイラという形で提供されるが、その上でプログラムを行なうだけでなく、明確な実行意味を持つ仕様記述としても用いることが出来る。

また、その単位を用いた並列検証系を実装するための並列検証系のアルゴリズムを提案し、それを効率良く実現するための汎用の並列通信ライブラリを作成した。

今後は、CbC/CwCコンパイラを実用的なレベルにまで質を上げるとともに、並列検証系と組み合わせた、新しいプログラム単位を直接的に使った検証システムを提供することを考えている。

４．成果リスト

[1] 河野真治and 神里健司．UDPを使った分散計算環境とその応用．日本ソフトウェア科学会第16回大会論文集, September 1999.

[2] Shiji Kono, "Parallelization of Temporal Logic Verication by Dividing State Set", 1st International Work-shop on Specication and Verication of Timed Systems., March 1999.

[3] 河野真治．継続を持つＣの下位言語によるシステム記述．日本ソフトウェア科学会第17回大会論文集．Sep 2000.

[4] 河野真治（琉球大/科学技術振興事業団）神里健司（琉球大）．User Level Flow Control APIをもつ並列ライブラリの実装．SwoPP 2001. July 2001.

[5] 河野真治（琉球大/科学技術振興事業団），揚挺（琉球大）．Ｃ言語のContinuation based Ｃへの変換．SwoPP 2001. July 2001.

[6] 佐渡山陽，河野真治（琉球大）．Continuation Based ＣによるPS2 Vector unitのシミュレーション．情報処理学会システムソフトウェアとオペレーティング・システム研究会, June 2002.

[7] 河野真治，佐渡山陽．Continuation Based ＣによるTchnology Mappingのサポート．FIT 2002. Aug 2002.

[8] 屋比久友秀，河野真治（琉球大）．ユーザレベル通信ライブラリSuciのスナップショット・アルゴリズムへの応用．FIT 2002. Aug 2002

[9] 河野真治．継続を基本とした言語CbCのgcc上の実装．日本ソフトウェア科学会第19回大会論文集．Sep 2002.

[10] 屋比久友秀，河野真治，山城　潤．トランスポート層を考慮したスナップショット・アルゴリズムの考察．日本ソフトウェア科学会第19 回大会論文集．Sep 2002

[11] 山城　潤（琉球大），河野真治（琉球大，科学技術振興事業団さきがけ研究21「機能と構成」）．Java によるユーザトランスポート層の実現と評価．情報処理学会システムソフトウェアとオペレーティング・システム研究会．May 2003

[12] 上里献一 and 河野真治．Suci ライブラリのスナップショット API を利用した並列デバッグツールの設計．日本ソフトウェア科学会第20回大会論文集．Sep 2003

[13] 河野真治．継続を基本とするプログラム単位を用いた組込みシステム開発．組み込みソフトウェア工学シンポジウム2003. Oct 2003