ICOT Technical Memorandum: TM-0781

TM-0781

GHCプログラムの最適化

森田正雄(三菱総研), 上田和紀

July, 1989

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

ＧＨＣプログラムの最適化
Optimization of GHC Programs

森田正雄
Masao Morita

上田和紀
Kazunori Ueda

（株）三菱総合研究所
Mitsubishi Research Institute

（財）新世代コンピュータ技術開発機構
ICOT Research Center

Flat ＧＨＣプログラムの最適化技法を、二つの異なる角度から提案する。第１の技法は、中断（suspension）の少ないプログラムの時間効率を改善するものであり、コンパイル単位ごとの大域的解析によって、データの検査やプロセスの中断・再開始などに伴う動的な検査をできるだけ省略したコードを生成する。これによって、既存のネイティブ・コードの逐次処理系と比較して、naive reverse のプログラムで約６０％効率が向上した。第２の技法は、従来の処理方式では頻繁に中断を起こすプログラムを対象にしたものである。ストリームを用いたメッセージ通信に関する大域的解析を行って、メッセージ通信を、データ構造の生成でなく、手続き呼出しに近い形にコンパイルする。これによって、特に要求駆動型のプログラムの時間効率と空間効率が大幅に改善される。これらの技法は、逐次処理系の最適化のために開発したものであるが、並列処理系の効率向上にも役立てることができると期待される。

１．　はじめに

我々は、汎用計算機上にＧＨＣ[Ueda 86] の処理系（ＧＨＣ／Ｖ処理系）[森田 87] を開発してきた。ＧＨＣ／Ｖ処理系はＷＡＭ流の中間コード[Warr 83] やガード部の決定木への展開[森田 87] などの最適化技法などを取り入れたことにより、述語単位でみた目的コードはかなり効率的になってきている。しかし、ある程度まとまった処理を手続き型言語で記述する場合と比較すると、効率上かなりの隔たりがある。これは、手続き（同じ頭部を持つ節の集合）という小さな単位（手続き型言語のサブルーチンや関数より、はるかに小さな単位）で独立にコンパイルしているため、本来の処理の部分と比べて、その手続きに渡ってくるデータの動的な検査部分の比重が非常に高くなってしまっているからである。このような非効率的な処理を、大域的な最適化により、軽減する処理方式を検討し、評価した。

以上の最適化技法は、プロセスの中断（suspension）があまり起きないプログラムを対象にしたものであるが、第２の技法は、頻繁に中断が起きるプログラムを対象にしたものである。

ＧＨＣ／Ｖ処理系のようにＷＡＭ流の中間コードをベースにした実現方法では、プロセスによって動的データ構造を実現したプログラムのように、散発的なメッセージ通信が行われるプログラムを実行すると、効率の良くない動作を行う。そのようなプログラムのメッセージの送受信を解析して、効率的に処理する方法を検討した。この方式では、ストリームを用いたメッセージ送信を、データ構造の生成ではなく、手続き呼出しに近い目的コードにコンパイルする。ストリームを作らず、また中断のタイミングをあらかじめ解析するので、時間効率、空間効率の双方が改善される。

２．　中断の少ないプログラムの大域的最適化

２．１　ＧＨＣ／Ｖ処理系の問題点

ＧＨＣ／Ｖ処理系は、ＷＡＭ流の中間コードやガード部の決定木への展開などの最適化技法を取り入れたことにより、述語単位でみた目的コードは、かなり効率的になってきている。しかし、ＧＨＣ／Ｖ処理系が出力するネイティブ・コードは、手続き型言語のプログラムから得られるネイティブ・コードとはかなり違ったものである。つまり、比較・条件分岐命令が非常に多い機械コード列になっている。

これは、強い型の概念のない言語のプログラムを、手続きという小さな単位で独立にコンパイルしているため、本来の処理の部分と比べて、その手続きに渡ってくるデータの全ての可能性を考慮した動的な検査（比較・条件分岐命令）を行う部分の比重が高くなってしまっているからである。そしてこれは、プログラムの実行上かなりのオーバヘッドになっている。

第２節では、このような非効率的な処理を大域的な視点で最適化する方法を考える。なお、ここで考える方法は、コンパイル時の入出力モード解析を前提にしているため、プログラムによりこの最適化が行えないものがある。

２．２　動的な検査部分の分析

比較・条件分岐命令を多く出している主因は、単一化である。単一化時、次のような検査を動的に行っている。

① プロセスの処理の中断・再開始検査
② ポインタのたぐり操作（dereferencing）
③ ボディでの単一化時のモード検査

④　ヒープ領域あふれの検査

①の中断・再開始検査には、ガード部での単一化時、呼出し側を具体化するか否かを検査する中断検査と、ボディでの単一化時、その単一化によって具体化される変数の値を待って中断しているプロセスがあるか否かを検査する再開始検査とがある。

②のたぐり操作とは、変数同士の単一化の際に生成したポインタをたぐる操作である。変数同士の単一化があると、処理系は、どちらか片方の変数を、他方へのポインタにする。このポインタは値として意味を持たない、単に参照関係を示すポインタ（不可視ポインタ）である。そして、このポインタが存在するため、データを取り出すときに常にこれを意識しなければならない。すなわち、不可視ポインタをたぐり寄せる操作を、不可視ポインタ以外の値に行き着くまで繰り返さなければならない。

③はボディでの単一化時、ソーステキスト上で変数記号で示されている引数が未定義変数か、あるいは具体化されているか、を動的に判定する検査である。

④は単一化時動的にデータが生成される領域（ヒープ領域）のあふれ検査であるが、静的なプログラム解析による最適化は難しく、本論文では触れない。

2．3　最適化の概要

実行時の動的な検査を最適化するため、次のような方法をとる。なお、本論文では、ガード部に組込みのテスト述語しか現れない Flat GHC[Ueda 88] のプログラムを考察の対象とする。また、本論文では、プロセス（ボディ・ゴール）の実行はプロセス・スタックを用いて深さ優先で行うものとする。つまり、新たに生成されたプロセスや、中断のあと再スケジュールされるプロセスは、すでに実行待ちになっているプロセスに優先してスケジュールすることとする。

まず、コンパイルの単位をある程度大きくし、その単位（モジュールと呼ぶ）中のトップ・レベルの手続きだけ外から見えるようにし、それ以外の手続きを外部から参照できない局所的な手続きとする。このようにすると、モジュールのトップ・レベルから渡ってくる情報を直接参照する単一化は、依然としてあらゆるケースを想定した検査が必要であるが、それ以外の単一化は静的な解析により、多くの動的な検査を省略することができる。

まずガードでの単一化について考えてみよう。図1のようなプログラムを考える。

```
(1)  nreverse([]   ,O) :- true | O = [].
(2)  nreverse([H|T],O) :- true |
         nreverse(T,OI), append(OI,[H],O).
(3)  append([]   ,I,J) :- true | I = J.
(4)  append([I|J],K,L) :- true |
         L = [I|M], append(J,K,M).
```

図　1　リストの反転

nreverse/2 をトップ・レベルの手続きとし、append/3 を局所的な手続きとすると、nreverse/2 の第1引数はどのようなものか特定できないが、append/3 の第1引数はこのコンパイル単位の内部で生成されるものが渡ってくるため、動的な検査の一部が不要となる。

さらに、ボディでの単一化についても、静的解析によって動的な検査を減らすことができる。図1の例では、nreverse/2 の第2引数（O）の値は、その nreverse/2 の呼出しの実行中はいずれの節のガードでも検査されない。そこでこの引数に対して外部から与えられる値と nreverse/2 が計算する値との単一化は、nreverse/2 の実行が終了または中断するまで遅延させることができる。そうすることにより、(1),(3),(4)節のボディの単一化を、単なる代入操作にコンパイルすることができる。

このような最適化の基本となるのは、各述語の引数のモード解析（2．4．1節）である。モード解析の目的は次の二つである：

① 各述語の引数を出力引数（計算結果を返すための引数）と入力引数（それ以外の引数）に分類すること。
② トップレベルの述語を呼び出したゴールの出力引数が外部（(擬似)並列に走っている他ゴール）から具体化されない限り、ボディの単一化が失敗しないことを保証しようとすること。

モード解析は、述語の引数だけでなくデータ構造（たとえばストリーム中のメッセージ）の引数に対して行うことも可能であるが、本論文では話を簡単にするため、述語の引数に限定して考える。

以下では、解析によって可能になる種々の最適化について述べる。

2．3．1　ボディの単一化の最適化

GHCでは、ボディの単一化は出力引数に値をバインドするために用いるのが普通であり、単一化が失敗した場合はプログラムもしくは最初に与えたゴール節に誤りがあると考えてよい。ボディの単一化の最適化の要点は、単一化の失敗の可能性と、単一化の結果生じる結合を待っているプロセスの存在の可能性とを排除して、単一化を単なる代入操作にコンパイルすることにある。

この最適化を行うには、まずモジュール内の各述語の引数をモード解析によって入力と出力に分類する。そして、あるゴールの実行に対する他ゴールからの影響をできるだけ取り去るため、次のようなことを行う。まず、あるゴール（pとする）の実行を開始するときは、その各出力引数を別々の新しい未定義変数に置き換え、それらの未定義変数と出力引数との単一化を、p自身またはpから生成されたサブゴールがすべて終了または中断した後実行されるようにプロセス・スタックに登録する。また、中断したゴール（pまたはそのサブゴール）が再開始するときにも、その各出力引数を別々の新しい未定義変数に置き換え、それ

らの未定義変数と出力引数との単一化をプロセス・スタックに登録する。ただし、あるゴールの出力引数が、そのゴールの実行直前に生成された変数であるとき（たとえば図1で、節(2) のボディ・ゴールのうち nreverse(T,O1) を先に実行した場合の変数 O1 ）は、ゴールの実行開始時の出力引数の置き換えを省略することができる。

このようにすることにより、出力引数に対する単一化は、動的な検査の不要な代入となり、手続き呼出し時に出力引数が未定義であるという通常の場合の処理効率が向上する。

2．3．2　ボディ・ゴールの実行順の最適化

単一化における動的な検査の最適化とともに、モジュール内での不要な中断・再開始処理をなくすことも重要な最適化の一つである。

中断・再開始処理を最適化する方法として、第3節に述べるように処理方式を根本的に変えてしまう方法もあるが、ここでは単に中断・再開始処理の回数を減らす工夫を考える。その一つの方法は、データを生成するプロセスの処理をデータを消費するプロセスの処理に先行させることである。つまり、ボディ・ゴール中の共有変数の入出力関係を調べ、共有変数を出力引数としているものをなるべく先に呼ぶようにする。例えば図1の例では、(2) のボディ・ゴールの呼出しは nreverse/2 を先に処理する。

このように中断・再開始処理の最適化を行った後、中断・再開始処理が起こり得るところでのみ中断・再開始検査を行い、それ以外の単一化の処理から、中断・再開始検査部分を取り去る。

2．3．3　ポインタのたぐり操作

計算機上で論理変数とその値を表現する場合、ポインタの助けを借りる。そのため通常のインプリメンテーションでは、単一化時、毎回引数の値をたぐり寄せる操作を行う。しかし、結合がどのように作られたかが明らかであれば、多くの動的なたぐり操作を取り除くことができる。

例えば図2のプログラムを考えてみる。

```
(1)  a :- true | b(X), c(X).
(2)  b(X) :- true | X = foo.
(3)  c(X) :- X = foo | true.
```

図　2

ここではWAMのように、ゴールの引数はレジスタを用いて受渡しすると仮定する。(1) のボディ・ゴールを実行する場合、まず二つのボディ・ゴールの共有変数Xの変数セルが生成される。次に、ゴール b(X) を先に実行することにすれば、その引数レジスタの値は変数セルへのポインタとなる。未定義変数の値を、その変数のアドレスと考え、変数への書込みはレジスタ間接命令で行うとすると、(2) のボディでの陽なたぐり操作は不要である。そして X = foo の実行により変数セルに値がセットされる。次にゴール c(X) を実行するとき、引数レジスタの値は、値のすでにセットされた変数セルへのポインタである。従って、引数値は必ずレジスタから直接読出すことにすれば、(3) のガード部での単一化時1回のたぐり操作が必要である。

このように共有変数が引数として渡る場合、出力側のボディの単一化ではたぐり操作は不要で、入力側のガードの単一化では1回のたぐり操作を必要とする。また変数以外が渡る場合は入力する側のたぐり操作は不要である。

図1のプログラムを例にとると、nreverse/2 の第1引数の検査は外部からのストリームを参照しているため、最適化できない。しかし、append/3 はコンパイル単位内で生成したデータしか参照していないため、最適化が可能である。append/3 の入口では第1引数の1回のたぐり操作が必要だが、それ以降の再帰呼出しでは、ガードの単一化でもボディの単一化でもたぐり操作はいらない。ガードの単一化でたぐり操作がいらないのは、このコンパイル単位内の単一化では不可視ポインタは作られないことが静的に判るからである。ボディの単一化でたぐり操作がいらないのは、トップレベルの nreverse/2 の出力引数が未定義変数におきかえられている（2．3．1節）からである。

2．4　コンパイル技法

本節では、2．3節で述べた最適化技法を具体的に述べる。

この最適化では、述語レベルではなく、モジュール・レベルのモード解析がベースになる。モード解析は、各述語の引数を出力引数（計算結果を返すための引数）と入力引数（それ以外の引数）に分類し、モジュール内の各節の頭部とゴールに一貫したモードづけを行うことを試みる。

ここでは解析を単純化するため、プログラマは次の約束に従ってプログラムを書いたつもりであると仮定する：

① すべての述語の引数は、入力引数または出力引数に分類できる。
② どの述語も入力引数を具体化することはない。
③ ボディの単一化は、すべて未定義変数である出力引数に対する代入となる。
④ 出力引数の値をガードで検査することはない。
⑤ 生成されたプロセス間に共有変数があるとき、それを出力引数として持つプロセスは高々ひとつである。

これらの仮定により、モード解析を試みる。これらの仮定からわかるように、本論文で用いるモード解析は、返信を伴うストリーム通信のような、複雑なデータの流れの解析はできない。しかし、型推論と組み合わせれば、述語の引数だけでなくデータ構造の引数のモード解析も可能である。本質的に解析ができないのは、ある変数のトップレベルの値（主関数記号）を決める可能性のあるプロセスが複数個あるようなプログラムである。

なお、モードづけに成功したモジュールについては、上の五つの性質を満たすことが保証される。

2．4．1　モード解析

モード解析は、以下に述べるように、正規化したプログラムについて、モードに関する制約不等式を立てて解くことによって行う。

```
queen(N, Answer_list) :- true |
      generator(1,N,Seq),
      queen(Seq,[],[],Answer_list,[]).

generator(M,N,O) :- M=<N |
      M1:=M+1, O=[M|O1], generator(M1,N,O1).
generator(M,N,O) :-  M>N | O=[].

queen([P|U],C,L,I,O) :- true |
      append(U,C,NN), c1(P,1,NN,L,L,I,X),
      queen(U,[P|C],L,X,O).
queen([],[_|_],_,I,O) :- true | I=O.
queen([],[],   L,I,O) :- true | I=[L|O].

c1(T,D,N,[P|R],B,I,O) :- T=\=P+D, T=\=P-D |
      D1:=D+1, c1(T,D1,N,R,B,I,O).
c1(T,D,_,[P|_],_,I,O) :- T=:=P+D | I=O.
c1(T,D,_,[P|_],_,I,O) :- T=:=P-D | I=O.
c1(T,D,N,[],   B,I,O) :- true |
      queen(N,[],[T|B],I,O).

append([A|X],Y,Z) :- true |
       Z=[A|Z1], append(X,Y,Z1).
append([]   ,Y,Z) :- true | Z=Y.
```

図３　８－ｑｕｅｅｎｓ

ステップ１　与えられた Flat GHCプログラムの各節を、[Ueda 88] の方法にしたがって正規化する。正規化とは、ガード部とボディ部の単一化ゴールを実行、変形して次の条件を満たすようにすることである：

①ガード部には単一化ゴールは存在しない。
②ボディ部の単一化ゴールは
$v_1 = t_1, \ldots, v_n = t_n$
の形をしている。ここで

- $v_1 \cdots v_n$は、頭部に現れる相異なる変数。
- $v_1 \cdots v_n$は、$t_1 \cdots t_n$中やボディ部の他のゴールに現れない。
- あるt_iが変数の場合、その変数は頭部に現れる。

たとえば、図３のプログラムは正規化された形になっている。

ステップ２　各述語の引数のモードを、下記の制約から決定する：

(1)　ある述語の第ｋ引数が、その述語を定義している節Ｃのガード部で検査される場合、その引数は入力である。ここである述語の第ｋ引数がＣのガード部で検査されるとは、

- Ｃの頭部の第ｋ引数が非変数である。
- Ｃの頭部の第ｋ引数が、頭部に２回以上現れる変数である。
- Ｃの頭部の第ｋ引数が、ガードゴールに現れる変数である。

のうちの一つ以上が成り立つことである。

(2)　ボディの単一化ゴールのふたつの引数は、逆のモードを持つ（通常の述語のモードづけは一意的であるが、述語‘＝’は例外的に、入力＝出力、出力＝入力の二通りのモードを持つものとする）。

(3)　あるボディ・ゴールのある引数が非変数の場合、そのゴールのその引数は入力である。

(4)　ある述語ｐを定義している節Ｃ中の変数ｖが、あるボディ・ゴールの出力引数として現れるとき

- ｖが他のボディ・ゴールの引数として現れる場合、それらの引数は入力である。
- ｖが頭部に現れるときは、ｐの出力引数の位置に現れる。

(5)　単一化以外の述語の各引数のモードと、その述語を呼び出すゴールの各引数のモードは一致する。

これらの制約から、全述語の入出力モードを決定する。すべての入出力モードが決定できなかった場合には、コンパイル不能とする。

これらの制約は、具体的なプログラムが与えられると、二値領域 {in,out} 上の連立不等式として書き下すことができる。図３のプログラムについてそれを解くと、次のように入出力モードが決定する。

```
queen(in,out)
generator(in,in,out)
queen(in,in,in,out,in)
c1(in,in,in,in,in,out,in)
append(in,in,out)
```

2．4．2　ゴール呼出順決定

次に、中断・再開始処理を最適化し、逐次処理し易くするため、ボディ部のゴール間に現れる共有変数を検査し、ゴール呼出し順序を並び替える操作を行う。具体的には、ボディ部のゴール間に共有変数がある場合、それを出力引

数としてもつゴールを先に呼び出すようにゴール呼出しの順序替えを行う。図３では、ボディ部に単一化以外の複数のゴールがあるのは、 queen/2, generator/3, queen/5 および c1/7 のそれぞれの第１節であり、それらの節のボディ・ゴールを次のように並び替える。

```
queen(N,Answer_list) :- true |
        generator(1,N,Seq),
        queen(Seq,[],[],Answer_list,[]).
generator(M,N,O) :- M=<N |
        M1:=M+1, O=[M|O1], generator(M1,N,O1).
queen([P|U],C,L,I,O) :- true |
        append(U,C,NN),
        queen(U,[P|C],L,X,O),
        c1(P,1,NN,L,L,I,X).
c1(T,D,N,[P|R],B,I,O) :- T=\=P+D, T=\=P-D |
        D1:=D+1, c1(T,D1,N,R,B,I,O).
```

このような順序替えは必ずできるとは限らないが、もしできれば、多くの場合中断・再開始処理が最適化される。

実は、queen/5 の第１節の最後の二つのゴールのように、２本の差分リストを接続するために共有変数（Ｘ）が使われている場合については、ゴールの並び替えは中断・再開始処理の最適化にはならない。なぜなら、２本の差分リストは独立に生成できるからである。しかし、このような場合でも、上記のような並び替えをしておくと、特別な工夫をすることなしに、余分なたぐり操作のいらない（不可視ポインタのない）差分リストが生成されるという利点がある。ゴールの呼出し順序に関係なく、常に余分なたぐり操作のいらない差分リストが生成されるように工夫することも不可能ではないが、差分リストの生成コストが高くなる。

２．４．３　中間コード生成

現段階でコンパイラは未完成であるため、２．３節で述べた点について、コード生成の要点を述べる。以下に述べる要点は、従来提案されているガード部の決定木化[森田87]、レジスタ割当ての最適化 [Kimura 87]等の手法とは独立であり、両者を併用することができる。

① ポインタのたぐり操作

ＷＡＭをベースにした中間言語体系では、ガード部で引数値を調べる命令や、ボディ部で引数の値を具体化する命令の内部でそれらの引数のたぐり操作を行っていた。だが、本論文の方法では、たぐり操作は必要な箇所でのみ行うようにし、また必要な回数がわかっている場合はその回数（多くは１回）だけ行うようにする。さらにモジュール外部から渡ってくるリストのように解析ができないものについても、引数値の検査にあたって、不可視ポインタであるかどうかの検査を後回しにすることによって、余分なたぐり操作のいらないリストがきた場合の処理効率を上げることができる。（２．４．２節に述べた最適化技法によって、不可視ポインタの出現頻度は少ないと仮定してよい。）

② 出力引数への代入

２．４．１節のモード解析に成功したプログラムについては、ボディの単一化が、たぐり操作の不要な代入になることが保証されている。しかも２．３．１節に述べた技法によって、この代入により再開始できるプロセスのないことが保証されている。

③ データ型の検査

たとえば図３のプログラムの generator/3 を考える。queen/2 の節の中にあるゴール generator(1,N,Seq) の起動、及び再開始時には引数Ｎのデータ型の検査が必要であるが、generator/3 の第２節の再帰呼出し generator(M1,N,O1) の起動時には引数Ｍ１やＮは整数値に具体化していることが保証されている（ゴール M1:=M+1 が中断することはない）ので、これらの引数に対するデータ型の検査が省略できる。これによって第二節を処理するループを小さくできる。

２．５　簡単なプログラムでの評価

リスト反転（図１）と自然数の生成及び８クイーン（図３）で最適化の効果を実測したところ、表１のような結果となった。

表　１　　ＶＡＸ１１／７８０での評価

プログラム	最適化	性能	命令実行回数
n-reverse 300 要素	無し 有り	33.2 52.9	27 18
generator 1～40000	無し 有り	21.1 42.5	40 22
8 queens	無し 有り	11.5 16.3	－ －

注：　性能の単位はＫｒｐｓ（単一化及び算術演算以外のボディ・ゴールの１秒当りのリダクション数）、また命令実行回数は頻度高く実行されるループを１回まわったときの命令実行回数（ n_reverse であれば appendのループの命令実行回数）。なお８queen は複数の実行頻度の高いループがあるため、命令実行回数を表示していない。

各局所述語のガードでのたぐり操作、ボディでのたぐり操作、モード判定などが最適化されている。最適化の効果を n-reverse の appendループについて言えば、ガードでの単一化で３命令減り、ボディでの単一化で６命令減った。最適化無しの場合の速度が、ネイティブ・コードを生成する処理系で得られたかなり高いものであることを考えると、大域的最適化の効果は大きいと言える。

2．6　中断の少ないプログラムの最適化のまとめ

汎用機上のＧＨＣのプログラムの最適化は、動的な検査をできるだけ排し、言語のセマンティクスを満たす範囲内で、いかに効率のよい逐次処理コードを生成するかというところに行き着くと考えられる。ここに提示したコンパイル単位毎の最適化技法により、動的な検査の比重の高かった小さなループ中の処理が効率よく処理できるようになった。

しかし、すべてのゴールを逐次処理できる8-queens のようなプログラムは、さらに効率を改善することができる。それは、2．5節の評価に用いた目的コードでは、小さなゴールの呼出しに対しても、重いプロセス・レコード生成処理を行っているからである。中断する可能性のない小さなゴール（例えば　append/3）に対してはプロセス・レコードを生成せず、ボディの処理時サブルーチンとして実行するのが望ましいだろう。ただし、ゴールの実行が中断しないかどうかの解析は、算術演算のような単純なゴールについては容易であるが、一般には抽象解釈のような技法を必要とする。

本節で提案した最適化は、ネイティブ・コードの大きさを減らすのにも効率的であり、ネイティブ・コードの利用範囲を広げるのに役立つと期待できる。

3．　メッセージ指向の最適化

ＧＨＣで書かれたプログラムをみると、プロセスの発生の仕方、プロセス同士の結合の仕方やプロセス間通信の特性は多種多様である。すなわち、プログラムにより、最適化の方針は異なってくる。言語処理系を設計する場合、最大公約数的な設計法のみによらず、具体的なアプリケーション・プログラムを例にとって考えるのも一つの方法だろう。

本節では、ＧＨＣ／Ｖ処理系の枠組みから離れて、不規則で散発的なメッセージ通信が行われるようなプログラムを効率よく処理する処理系を考える。

3．1　データベース

検討するアプリケーションプログラムとして、数値をキーとした2進木データベースを考える（図5）。木構造のデータベースをＧＨＣで記述する場合、各ノードをデータを引数に持ったプロセスによって実現し、プロセス間のアークを、検索要求メッセージのストリームとして実現するのが自然である。

データベースへの要求は、一般に散発的に発生する。そのノードを実現する各プロセスは通常、メッセージが1個来るとリダクションできて、高々1個のメッセージを出して再び待ち状態にはいる。このようなアプリケーション・プログラムをＧＨＣ／Ｖ処理系で動作させると、要求が一つのノードに到達する度に、再開始・中断処理が行われ、非効率的に動作する。そこで、散発的にくるメッセージを一般的に効率よく処理する枠組みを考えてみる。

3．2　プロセスの処理とデータの流れ

1台のプロセッサが複数のプロセスを実行する擬似並列処理では、一つのプロセスの処理が始まったら、できるだけの仕事をしてから別のプロセスの処理に移る方法と、複数の仕事ができる状態でも一つだけ仕事をやり、すぐにその結果により動作できるようになったプロセスの処理に移る方法とがある。図4のプログラムを例にとって考えてみよう。

```
(1) append([]    , L, M) :- true | L = M.
(2) append([I|J], L, M) :- true |
                        M = [I|N], append(J,L,N).
```

図　4

ゴール append(A,B,C) は、Ａが非空リストに具体化すると、第2節を選択して、ボディ部の二つのゴールを実行することとなる。この際に第3引数Ｃに対する単一化と、再帰呼出しのどちらを優先させるかという問題があるが、ここではとりあえず中断することのない単一化を先に実行することとする。次にその単一化により、実行を再開始できるプロセスが存在した場合、それと再帰呼出しのどちらを先に実行するかという二つの選択肢がある。ここで、再帰呼び出しを優先させる方をプロセス指向と呼び、単一化によるプロセスの駆動を優先させる方をメッセージ指向と呼ぶことにする。

プロセス指向が入力データの処理のスループットを重視した方式であるのに対し、メッセージ指向はレスポンスを重視した方式であるといえる。処理する側から考えると、前者はループをいかに効率よく処理するかが問題であり、後者はメッセージ授受の際のプロセス切り替えをいかに効率よく処理するかが問題である。ＧＨＣ／Ｖ処理系では前者に主眼をおき、ＴＲＯ（ tail recursion optimization）や引数・変数をハードウェアレジスタに乗せる等の最適化を行っている。が、このような処理系には、散発的なメッセージ通信の効率が非常に悪くなるという欠点がある。他方メッセージの流れを主眼においた処理系は、連続してくるデータに対する処理効率は若干劣るかもしれないが、メッセージが散発的にきても効率低下がない。メッセージが散発的に発生するようなプログラムには、オブジェクト指向プログラム[Shapiro 83]やデータベース・プログラムなどがあり、メッセージの流れを優先させる処理方式の適用範囲は広いと考えられる。

3．3　処理方式

メッセージの流れを中心に考えると、各プロセスは定常的に待ち合わせの状態にし、ストリーム中を流れるメッセージの送受信に伴って処理プロセスを切り替えてゆくのが自然である。これはプロセス指向の処理において、最適化の方策として中断・再開始処理をできるだけ避けていたのと非常に対照的である。

この観点から図４のプログラムを見直してみよう。中断状態を各プロセスの定常状態と見なすのであるから、以下でも中断状態にあるプロセス append(A,B,C)を考える。またここでは、引数のストリームの具体化は１要素ずつおきるものとする。

Aが非空リストに具体化すると、第２節の二つのボディ・ゴールを実行することとなるが、中断状態をプロセスの定常状態と見なすならば、まず再帰呼出しを実行して、この append プロセスの中断状態を回復しておくのが自然である。その後、単一化を実行して、Cの値を待ち合わせているプロセスの処理に制御を移せばよい。append プロセスの中断状態を先に回復しておくと、単一化の実行が、第２節の最後の仕事となるので、制御の移動のあと、append プロセスに制御を戻す必要がなくなる。

但し、単一化の実行にともなう制御の移動には、次のような問題がある。

① 受信プロセスは、一般に必ず存在するかどうかわからない。
② 受信プロセスが何であるかは静的にはわからない。
③ 受信プロセスを直ちに実行するといっても、受信側プロセスはすでに生成され、メッセージを待ち合わせ中のプロセスであり、ＴＲＯのようにこれから生まれるプロセスを直接実行させる場合と本質的に違う。

まず①の問題は、(A) 受信プロセスがそもそも存在しない（未来永劫作られない）場合と、(B) 存在はするが送信しようとする情報を待つ状態になっていない場合の二つに分けられる。だが、大域的なプログラム解析によってプロセス間通信のための引数や通信の向きが判れば、３．４節に示すような技法によって、送信した情報が必ず受信されるようにすることができる。

次に②の問題は、ＧＨＣのプロセスが動的に生成されるものであることの帰結であり、単一化時にどのプロセスを駆動するかという情報は、動的に決まる情報として管理しなければならない。

最後に③の問題は、効率上の大きな問題である。プロセス指向の処理系におけるＴＲＯでは、プロセス・スタックを経由せずに直接起動するプロセスは新しく生まれるプロセスであって、引数情報を親プロセスの引数情報から直接引き継いで、容易に実行が開始できる。他方メッセージ指向の最適化の場合、受信側プロセスは待ち合わせ中のプロセスであり、プロセス切り替えは避けられない。そこでメッセージ指向の処理系では、プロセス切り替えをいかに効率的に処理するかが、最も重要である。プロセス指向の処理系のように、プロセス・スタックを介することなく、ストリームのみを見て、中断・再開始処理が行えるような枠組みを持つことが必要である。

これらを総合して考えると、ストリーム通信のための変数（以下通信変数という）は他の変数とは全く別の枠組みで処理するのが自然である。そこで、通信変数のための特別なタイプのセル、通信セルを考える。このセルは、受信側プロセスの環境（引数値等を記録したプロセス・レコード）へのポインタ及び処理再開始点（機械コードのアドレス）の情報を持つ。通信変数を具体化する単一化時は、このセルのみの参照により、プロセス・スタックを経由せず、直接受信側プロセスに制御を移す。また、メッセージは高速化のため、メモリでなく、ハードウェア・レジスタ（通信レジスタ）により、授受する。

一般にストリームは非有界バッファの役割を果たすが、メッセージ指向の処理系ではプロセッサは一つのメッセージの流れを追って処理を進めるので、本質的にバッファリングを必要とするプログラム以外については、ストリームがバッファとして機能することはない。あるプロセスが複数個のメッセージを同じストリームから受け取らないとリダクションできない場合は、バッファリングがいるようにも思えるが、この場合はメッセージを１個送るたびにプロセスの内部状態を更新する（受け取ったメッセージを記録し、再開始点を更新する）ことで対処でき、非有界バッファを用意する必要はない。本質的に非有界のバッファリングが必要となるのは、

- プロセスがストリームによって環状に接続されたプログラムや、
- 一つのストリームからメッセージがきてもリダクションできるとは限らないプロセス（前述の append プロセスは、第２引数が原因でこれに該当する）を持ったプログラム

に限られ、図５のようなデータベース的プログラムを含む多くのプログラムではバッファリングは不要である。そこで、バッファリングは受信プロセス側で必要に応じて行うこととし、メッセージの授受自身は上記のような機構にしておくことにより、プロセス間通信の最適化をはかることができる。

プロセス指向の処理系では、プロセスの引数や局所変数をハードウェア・レジスタに乗せるような最適化をすることがあるが、メッセージ指向の処理系では、この最適化はプロセス切り替わり時のレジスタの退避・復元のオーバーヘッドを招くために逆効果となる。メッセージ指向の処理系では、引数・変数の参照はメモリ参照によって行うべきである。

３．４　プログラムの解析

前節の処理方式を与えられたプログラムに適用するためには、コンパイル時のプログラム解析が重要となる。本節では、どの引数がストリームで、データがどちら向きに流れるかはプログラマの宣言とプログラム解析によってコンパイル時にわかるものと仮定する。ひとつの宣言方法は、ストリームのコンストラクタをリストのそれと区別する方法であり、本節のプログラムでは‘{_|_}’（非空ストリーム）と‘{}’（空ストリーム）を用いる。この宣言方法の利点は、プログラマの負担が少ないことと、Flat ＧＨＣの枠組を全く変更・拡張していないことである。なお、プログラム解析は、２．４節で示唆したような、型推論とモード解析を組み合わせた方法がそのまま適用できる。

3．3節で述べた①の問題を解決するためには、次の二つのことを保証すればよい：

(A) メッセージを受信するプロセスの存在を保証する。
(B) メッセージを受信するプロセスの実行を、生成するプロセスより先行させ、送ったメッセージが直ちに処理できるようにする。

まず (A) の要請は、プロセス間通信に種々の形態があるので、基本となる５つのタイプについて処理方法を説明する。一般のm対nの通信については、下の①または②または③と、④または⑤とを組み合わせて考えればよい。

① 1対1のプロセス間通信
② 受信プロセスがない場合（1対0の通信）
③ 同一通信変数を複数のプロセスで参照する場合（1対nの通信）
④ 複数のストリームからのメッセージの選択的待ち合わせがある場合（n対1の選択的通信）
⑤ 複数のストリームからのメッセージの非選択的待ち合わせがある場合（n対1の非選択的通信）

①はプロセス間通信の基本であり、②や③は、コンパイル時のプログラム変換によって①に帰着させることができる。

②は、ボディにのみ1回しか現れない出力通信変数があったり、受信プロセスが下のプログラムのように、入力ストリームが閉じる前に消滅するようになっている場合におきる。

```
p({N|T},O) :- N >= 1 | O = {N|OT}, p(T,OT).
p({N|T},O) :- N <  1 | true.
```

このようなプログラムのひとつの変換方法は、通信変数を空読みするダミーのプロセスを発生させる方法である。

```
p({N|T},O) :- N >= 1 | O = {N|OT}, p(T,OT).
p({N|T},O) :- N <  1 | dummy(T).
dummy({H|T}) :- true | dummy(T).
dummy({})    :- true | true.
```

ただし、受信プロセスの消滅が頻繁におきるプログラム（たとえば投機的(speculative)な探索をするプログラム）ではもっと高度な方法を考えなければならない。

③の場合の典型は、次のようなプログラムで、変数Sが通信変数であった場合である。

```
p :- true | producer(S),consumer1(S),consumer2(S).
```

このようなプログラムは、次のようなストリームの分配器つきのプログラムとみなしてコンパイルする：

```
p :- true | producer(S), tee(S,O1,O2),
            consumer1(O1), consumer2(O2).
tee( {} ,O1,O2) :- true | O1 = {}, O2 = {}.
tee({H|T},O1,O2) :- true |
            O1 = {H|O1}, O2 = {H|O2},
            tee(T,O1,O2).
```

④は、ストリームの併合器のようなプロセスを受信プロセスとしている場合を指す。この場合各入力ストリームからのメッセージは独立に処理できるので、通信変数毎の処理は、1対1通信の場合と同様となって、変換が不要である。

⑤は、ストリームの順序付き併合器（要素が整列してあるストリームを、順序を保って併合するプロセス）やappend プロセスのように、一方のストリームにメッセージを送っただけではリダクションできるとは限らないプロセスを受信プロセスとしている場合を指す。この場合は、一般に送ったメッセージが直ちに受信プロセスのリダクションを引き起こす保証がないばかりでなく、送ったメッセージが受信プロセスの次のリダクションに貢献するという保証もない(append の第2引数にメッセージを送り続けた場合を考えてみるとよい)。次のリダクションに無関係なメッセージがきたら、受信側でバッファリングをしなければならない。

次に(B)の要請は、モード解析の結果に基づいてゴールをスケジューリングすることにより、①～④の場合については問題なく満たすことができる。⑤の場合は、上述のように一般的にバッファリングが必要になるが、メッセージをバッファリングするかどうかは受信側が決めることなので、受信プロセスを中断するまで実行しておけば、メッセージをともかく受信側に流すことができる。したがって、いずれの場合もメッセージ送信は、常に送ったメッセージの面倒をみてくれる受信プロセスがあるものとして実行することができる。

3．5 メッセージ指向の処理系の試作

この処理系の試作は、ＧＨＣ／Ｖ処理系の基本部分を流用し、それをベースに拡張することにより行った。以下に主な拡張部分について記述する。なお、本試作は、提案する処理方式の効率評価の第一段階として行ったものであり、まだ図５のプログラムを処理するのに必要な程度の機能しか実現していない。特に、バッファリングを必要とするプログラムにはまだきちんと対処していないことを注意しておく。バッファリングが必要な場合の方策は、本節の終わりで考える。

不規則で散発的なメッセージの発生を効率よく処理するために、プロセス間の共有変数を通信セル（3．3節）によって実現し、またメッセージ授受のための通信レジスタを設けることにした。また、次のような５つの中間コードを設けた。以下の中間コードでは、ＤとＰはレジスタ番号、Ｇは入口点の名札を表す。

・send_call（D，P）

出力ストリームPにデータDを渡す。この命令は、節のボディで行う最後以外のメッセージ送信に用いる。

処理：① 通信セルが現れるまでPの値をたぐり寄せる。

② 通信レジスタにデータDをセットする。

③ 処理中のプロセスのプロセス・レコードの継続点エントリに次の中間コードのアドレスをセットする。

④ プロセス・スタックの先頭に処理中のプロセスをスタックする。

⑤ 通信セルが示す受信プロセスの処理に移る。

・send_jmp（D，P）

出力ストリームPにデータDを渡す。この命令は、節のボディで行う最後のメッセージ送信に用いる。

処理：① 通信セルが現れるまでPの値をたぐり寄せる。

② 通信レジスタにデータDをセットする。

③ 通信セルが示す受信プロセスの処理に移る。

・set_proc（P，G）

入力ストリームPにデータが来たときの処理の再開始点を名札Gの表す入口点にする。

処理：① 通信セルが現れるまでPの値をたぐり寄せる。

② 通信セルの処理再開始点エントリにGの表す入口点をセットする。

③ プロセス・スタックの先頭のプロセスの処理に移る。

・return

復帰する。

処理：① プロセス・スタックの先頭のプロセスの処理に移る。

・rcv_value（D，P）

入力ストリームPからデータを受取り、Dにセットする。

処理：① 通信セルが現れるまでPの値をたぐり寄せる。

② その内容が未定義変数ならば、

a. 通信セルを生成する。

b. 処理中のプロセス・レコードのアドレスを通信セルの環境エントリに登録する。

c. 下記のe. の命令のアドレスを通信セルの処理再開始点エントリにセットする。

d. プロセス・スタックの先頭のプロセスの処理に移る。

通信セルならば、

e. 通信レジスタの値をDにセットする。

節のボディでn（＞1）個のメッセージを送信する場合、単に（n－1）個の send_call 命令と1個の send_jmp 命令を逐次的に実行するのは、一般に正しくない。なぜなら、この送信プロセスが環状プロセス構造の中にあると、最初の send_call 命令が連鎖的にメッセージ送信を引き起こし、2番目の send_call 命令の実行前に、当プロセスにメッセージが届くことがあるからである。この問題に対する一つの解決法は、最後の send_jmp 命令を実行する直前までは、自分自身に対するメッセージ送信を rcv_value 命令で受け付けず、バッファリングしておくことである。send_jmp 命令の実行時にバッファリングされたメッセージ（下記に述べるn対1の非選択的通信によるものを除く）があったら、send_jmp 命令の実行後再び当プロセスに制御を戻し、それらのメッセージの受信処理を繰り返すようにする。

節の頭部で複数の入力ストリームからのメッセージを非選択的に待ち合わせる場合（n対1の非選択的通信）には、n対1の選択的通信の場合と同様どのストリームから先にメッセージが来ても良いようにし、さらに rcv_value 命令でメッセージを受け取る用意のないストリームからのメッセージは、バッファリングする必要がある。

以上の2点を考慮した中間言語命令は現在設計中であり、本論文の改訂版で紹介する予定である。

3．6 性能測定と評価

試作処理系とGHC／V処理系との性能比較ができるように、同一プログラムを双方の処理系で性能測定した。

測定対象のGHCプログラムは、図5のような2進木データベースプログラムである。なお、'{''}'は通信ストリームのコンストラクタであり、GHC／V処理系で実行させる場合、'{''}'を'['']'に置き換えてコンパイルする。またこのプログラムの中間コードイメージは、図6のようになる。

試験は、あらかじめ乱数を第1引数にもつ update メッセージの列を t_node プロセスに送信して2進木データベースを作成しておき、つぎに同じ乱数列から作られる、次のような search メッセージの列を送信することにより行った。

```
search(445,_), search(258,_), search(155,_), ....
```

まず、GHC／V処理系で search メッセージを一挙に与え、処理時間等を測定した。次に search メッセージを1件流し、応答が返ってきたら、次のメッセージを流すと言うように間欠的にメッセージを与え、それに要した時間等を計測した。次に試作処理系により、同様に処理時間等を測定した（この場合、メッセージを連続的に与えても、1件毎に与えても、処理は同じように行われるので、同じ結果となる）。測定結果を表2に示す。これらの測定結果

は、update メッセージによってデータベースを作成する手間を含んでいない。

表2　2進木データベースの測定結果（VAX11/780上）

測定対象（検索データ８００個）	処理時間（秒）	中断回数
GHC／V　DBの連続検索	1．04	0
DBの間欠的検索	2．09	9817
試作処理系　DBの検索	0．79	0

表2からわかるように、試作処理系の効率は、search メッセージの与え方とは無関係に、GHC／V処理系上での連続検索の効率を上回った。またGHC／V処理系での1回の中断・再開始のオーバヘッドは約１００μｓｅｃであることもわかった。

連続検索の時間効率については、GHC／Vによるデータには第2節に示したような改善の余地があることから、優劣比較の結論は下せないが、間欠的検索については、メッセージ指向の処理方式が有効であることが検証できたといってよい。また空間効率については、メッセージ指向の処理系は、メッセージの送信時にストリーム構造を作らないので、検索時にメモリが消費されないという利点がある。さらにメッセージ指向の処理方式の通信形態は1対1を基本としており、通信の終了時に通信のための領域（通信セル及びバッファ）を解放することができ、即時回収が不可能なゴミは一切発生しない。

また、図1のリストの反転プログラムについても性能測定を行った。このプログラムでは、append の第2引数に送られるメッセージをバッファリングする機構が必要であるが、暫定的な中間コードを作成し、処理した。なおこのプログラムでは、append の第2節の処理効率が全体の効率を支配しており、暫定部分の効率はさほど重要ではない。測定結果を表3に示す。

表3　リストの反転プログラムの測定結果（VAX11/780上）

プログラム	性能	命令実行回数
n-reverse 300 要素	60.4 Krps	9 (append呼ﾏﾏ)

メッセージ指向の処理系で n-reverse プログラムを実行すると、頻度高く実行される append プロセスは、送信されたメッセージを受信側へ渡すだけのプロセスとなる。このため最適化したプロセス指向の処理系と比較しても、優れた処理効率を示した（表1参照）。

ここに提示したメッセージ指向の試作処理系は、GHC／V処理系を改造したもので、まだ細かな最適化の余地が多く残っている。また、未だ動的に判定する処理部分が非常に多くあり、コンパイル時の静的な解析により、さらに最適化することも可能である。今後、第2節に提示したような最適化技法を援用することにより、さらに効率のよい処理方式にしてゆきたいと考えている。

4．　まとめ

汎用機上のGHCのプログラムの最適化は、動的な検査をできるだけ排し、言語のセマンティクスを満たす範囲内で、いかに効率のよい逐次処理コードを生成するかというところに行き着くと考えられる。このような視点に立ち、二つの具体的な最適化方法を提示した。第2節に示した技法の意義は、外部データに対する動的な検査の比重の高かった小さなループの処理効率が、プログラムをよく解析することによって大きく改善されることを示したことである。並列処理系でも、ここに示した解析技法や最適化技法は有効である。特に、入出力モードの解析は、プロセッサをまたがって実行される単一化の最適化に非常に有効だと考えられる。この技法はすべての Flat GHCプログラムを対象にしているわけではなく、またプログラム解析を詳細化すればするほどコンパイルや再コンパイルに要する手間が増える。だが、このような研究を通じて、言語仕様とコンパイルの手間と目的コードの効率の三者間のトレード・オフを研究してゆくことは重要なことだと考える。

第3節に示した技法の意義は、GHC／Vをはじめとする従来の処理方式が不得意としていた散発的なメッセージ通信を効率よく処理する枠組みを示したことにある。この処理方式がうまく実用化できれば、GHCによるオブジェクト指向計算、データベース、要求駆動計算などの記述の実用性が高まると期待できる。また、この最適化においても、プログラム解析が重要な役割を果たしていることを強調しておきたい。並列処理系でも、プロセッサ内のメッセージ通信にはこの技法はそのまま適用できる。プロセッサ間のメッセージ通信は、プロセッサ間の通信の面倒をみるプロセスを両プロセッサにおき、それらを経由して行えばよい。この場合、メッセージを他プロセスに送信したら、その処理が終わるのを待たずに次のメッセージの処理を始めることができる。

本論文で述べた2つの最適化手法は、一方がプロセスに着目した逐次処理に適したものに有効であり、他方がメッセージの流れに着目した逐次処理に適したものに有効である。すなわち、2つの最適化手法は対極に位置するものである。実際のプログラムは、それぞれの性格を持った部分が混在しているかもしれない。理想的には、コンパイル時にそれらを判断しながら、最適な処理を選択するのが望ましいが、それは非常に難しい。しかし、第3節で行ったように、プログラマが多少の情報を与えることにすれば、それを判断材料にして両手法を融合することが可能になるであろう。

コンパイル技術等の実用化レベルの問題が多くあるが、

それらを解決し、２つの最適化手法を融合することが今後の課題である。

《参考文献》

[Ueda 86] Ueda,K.: " Guarded Horn Clauses " , in Logic Programming '85, LNCS 221,Springer-Verlag, 1986, pp.168-179.

[Warr 83] Warren,D.H.D.: " AN ABSTRACT PROLOG INSTRUCTION SET " , SRI International,Technical Note 309, 1983.

[森田 87] 森田、吉光、太細、上田： 汎用計算機上のＧＨＣコンパイラ,第３５回情処全国大会 5Q-4, 1987.

[Ueda 88] Ueda,K. and Furukawa,K.: " Transformation Rules for GHC programs" ,in Proc. Int. Conf. on FGCS'88, ICOT, 1988, pp.582-591.

[Kimura 87] Kimura,Y. and Chikayama,T.: "An Abstract KL1 Machine and Its Instruction Set" , in Proc. 1987 Symp. on Logic Programming, IEEE Computer Society, 1987, pp.468-477.

[Shapiro 83] Shapiro,E. and Takeuchi,A.: "Object Oriented Programming in Concurrent Prolog" , New Generation Computing, Vol.1, No.1(1983),pp.25-48.


```
%  Binary Tree Database

nt_node({},                          _,   _,     L,R) :- true |
    L={}, R={}.
nt_node({search(Key,Value)|Cs},Key, Value1,L,R) :- true |
    Value=Value1,
    nt_node(Cs,Key,Value1,L,R).
nt_node({search(Key,Value)|Cs},Key1,Value1,L,R) :- Key<Key1 |
    L={search(Key,Value)|L1},
    nt_node(Cs,Key1,Value1,L1,R).
nt_node({search(Key,Value)|Cs},Key1,Value1,L,R) :- Key>Key1 |
    R={search(Key,Value)|R1},
    nt_node(Cs,Key1,Value1,L,R1).
nt_node({update(Key,Value)|Cs},Key, _,     L,R) :- true |
    nt_node(Cs,Key,Value,L,R).
nt_node({update(Key,Value)|Cs},Key1,Value1,L,R) :- Key<Key1 |
    L={update(Key,Value)|L1},
    nt_node(Cs,Key1,Value1,L1,R).
nt_node({update(Key,Value)|Cs},Key1,Value1,L,R) :- Key>Key1 |
    R={update(Key,Value)|R1},
    nt_node(Cs,Key1,Value1,L,R1).

t_node({}                      ) :- true | true.
t_node({search(_,  Value)|Cs}) :- true |
    Value=undefined,
    t_node(Cs).
t_node({update(Key,Value)|Cs}) :- true |
    nt_node(Cs,Key,Value,L,R),
    t_node(L), t_node(R).
```

図　５　　２進木データベースプログラム

```
 1  entry('nt_node'/5).
 2  rcv_value(6,1).
 3  switch(6,[[structure('update'/2)|l(103)],
              [structure('search'/2)|l(104)],
              [atomof($eos)|l(102)]]).
 4  label(103).
 5  unify_variable(8).
 6  unify_variable(9).
 7  switch_else(8,[],[l(105),l(106)]).
 8  label(105).
 9  c_integer(8).
10  tree([l(107),l(108)]).
11  label(107).
12  c_integer(2).
13  arith_lss(8,2).
14  commit.
15  send_jmp(6,4).
16  label(108).
17  c_integer(2).
18  arith_gtr(8,2).
19  commit.
20  send_jmp(6,5).
21  label(106).
22  c_get_value(8,2).
23  commit.
24  put_value(9,3).
25  return.
26  label(104).
27  unify_variable(8).
28  unify_variable(9).
29  switch_else(8,[],[l(109),l(110)]).
30  label(109).
31  c_integer(8).
32  tree([l(111),l(112)]).
33  label(111).
34  c_integer(2).
35  arith_lss(8,2).
36  commit.
37  send_jmp(6,4).
38  label(112).
39  c_integer(2).
40  arith_gtr(8,2).
41  commit.
42  send_jmp(6,5).
43  label(110).
44  c_get_value(8,2).
45  commit.
46  get_value(3,9).
47  return.
48  label(102).
49  commit.
50  put_const(atomof($eos),6).
51  send_call(6,4).
52  send_jmp(6,5).
```

```
54  entry('t_node'/1).
55  rcv_value(2,1).
56  switch(2,[[structure('update'/2)|l(103)],
              [structure('search'/2)|l(104)],
              [atomof($eos)|l(102)]]).
57  label(103).
58  unify_variable(4).
59  unify_variable(5).
60  commit.
61  spawn(procedure('t_node'/1)).
62  put_variable(6,sarg(1)).
63  spawn(procedure('t_node'/1)).
64  put_variable(7,sarg(1)).
65  put_value(4,2).
66  put_value(5,3).
67  put_value(6,4).
68  put_value(7,5).
69  set_proc(1,procedure('nt_node'/5)).
70  label(104).
71  unify_variable(4).
72  unify_variable(4).
73  commit.
74  get_const(atomof('undefined'),4).
75  return.
76  label(102).
77  commit.
78  proceed.
```

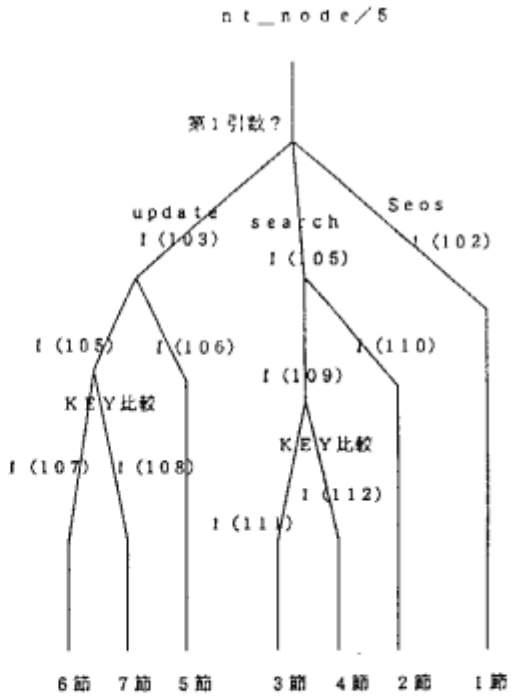

図　6　　2進木データベースプログラムの中間コードイメージ