アセンブリ言語上でのプログラム特化

徳生吉孝[†] 脇田建[††] 佐々政孝[††]

本稿では，RISC 型アセンブリ言語上でのプログラム特化技法を定式化する．アセンブリ言語上でのプログラム特化には，幾つかの問題がある．

一つ目は，データフローの複雑さの問題である．一般に，アセンブリプログラムは，コンパイラのレジスタ割付によって，ソースプログラムよりも複雑なデータフローを持つ．この問題を解決するためには，精度の高い束縛時解析を行わなくてはならない．そこで本稿では，最も精度の高い束縛時解析である，program-point sensitive な束縛時解析を採用した．

二つ目は，非構造的な制御フローの処理についての問題である．既存の C 言語上での特化器は，非構造的な制御フローを直接処理することが出来なかった．その為，特化の前に，非構造的な制御フローを取り除くプログラム変換を行う必要があった．本稿では，制御フロー解析を行うことによって，非構造的な制御フローを直接，処理することを可能にした．

我々の知る限り，アセンブリ言語上でのプログラム特化に関する研究は，これまで知られていない．

Program Specialization for the Assembly Language

Yoshitaka Tokusyou,[†] Ken Wakita[††] and Masataka Sassa[††]

In this paper, we formulate a program specialization technique for the assembly language. There are several problems in specializing assembly programs.

The first problem is due to the complexity of the data flow. Because of the register allocation, assembly programs have more complex data flow than source programs. To solve this problem, we must analyze accurately the binding time values of the programs. In this paper, we introduce the most accurate binding time analysis, called program-point sensitive binding time analysis.

The second problem is due to the existence of the unstructured control flow. Existing program specializers for the C language can not address directly the unstructured program. Therefore they need to transform the unstructured program into the structured program before the specialization phase. In this paper, we make it possible to specialize directly the unstructured program by analyzing the control flow of program.

As far as we know, research on the program specialization for assembly language has not been done until now.

1. はじめに

プログラム特化（*program specialization*）は**部分評価**（*partial evaluation*）とも呼ばれており，プログラムとその入力の一部を受け取ってプログラムの計算を部分的に進め，効率の良いプログラムを生成するためのプログラム変換技法である[12]．

既存のプログラム特化の研究の多くは，C[1),7)]，ML[4)] や Scheme[3)] 等の高水準言語で書かれたプログラムを対象として行われてきた．しかし，高水準言語上でのプログラム特化は，もはやソースコードが存在しないソフトウェアを特化の対象とすることはできない．日常的に使用している多くのソフトウェアは，ユーザーの手元にソースコードが存在しない．このようなソフトウェアを特化対象とするためには，ソフトウェアをバイナリコードのままで，またはそれに近いレベルで扱う必要がある．

そこで，本稿では，ソフトウェアをバイナリレベルでプログラム特化すること[8)]を目標に，そのための準備として，RISC 型アセンブリ言語上でのプログラム特化の技法を定式化した[18)]．我々の知る限りでは，アセンブリ言語上でのプログラム特化技法に関する研究は今までにない．

アセンブリ言語上でプログラム特化を行うことの利点は，もはやソースコードの存在しないソフトウェア

† （株）日立製作所 ビジネスソリューション事業部
Hitachi,Ltd. Business Solution Systems Division
†† 東京工業大学 大学院情報理工学研究科 数理・計算科学専攻
Dept. of Mathmatical and Computing Sciences, Tokyo Institute of Technology

を逆アセンブルしてから特化を行うことによって，元のソフトウェアよりも効率的に実行できることや，プログラムのソース言語に依存せずにソフトウェアの特化が行えることが挙げられる．

アセンブリ言語上でプログラム特化を行う場合，以下のような点で困難がある．

(1) 高水準言語の変数に相当するレジスタに加えて，メモリアドレスを扱う必要がある点
(2) アセンブリコードの持つ非構造的な制御フローを扱う必要がある点
(3) コンパイラによるレジスタ割付の結果，ソースコード中で異なる変数が同一のレジスタに割り付けられる場合がある等，アセンブリコードが複雑なデータフローを持つ点
(4) メモリアクセスに対するエイリアス解析が，高水準言語における一般的なエイリアス解析に比べて難しい点

本稿では，主に上記の（1）から（3）に対する解決策を提示する．(4) については，本稿では扱わない．

本稿では，複雑なデータフローを処理するために，*program-point sensitive* な束縛時解析を導入した．**束縛時解析**（*binding time analysis*[12]）とは，プログラムの入力の一部が与えられた時に，プログラム中の特化時に計算できる（**静的**：*static*）部分と，実行時まで計算できない（**動的**：*dynamic*）部分とを決定することである．静的と動的とを**束縛時値**（*binding time values*）と呼ぶ．program-point sensitive な束縛時解析とは，変数の束縛時値がプログラムポイント毎に計算されることである．これは，文献 10) によって初めて導入された束縛時解析で，変数の使用による出現に対する束縛時値と定義による出現に対する束縛時値とを，別々に扱うことで実現される．本稿では，program-point sensitive な束縛時解析を**制約解消**（*constraints solving*）[9),12),16)] を用いて，文献 10) のアルゴリズムよりも簡潔な表現で定式化した．制約解消による束縛時解析は，束縛時値に対するプログラム中の変数の不等式及び等式（**制約**，*constraint*）を解くことによって行われる．

また，本稿では，アセンブリ言語上の非構造的な制御フローを直接扱うために，制御フロー解析を行う．本稿の制御フロー解析は，フローグラフに基づいており，プログラム中に存在する制御フローの分岐点や合流点等に関して，様々な情報を収集する．

本稿の手法は一般の RISC アーキテクチャに適用できると考えるが，ここでは SPARC アーキテクチャ [19] のサブセットを仮定した．本稿では，アルゴリズムの簡単化のため，次のようにアーキテクチャを制限する．各項目の詳細は，3 節で述べる．

(1) %**sp** と %**fp** を含む 32 個の整数型レジスタのみを扱う．
(2) 制御転送命令の遅延スロットは全て **nop** 命令とする．
(3) 手続き間にまたがるプログラム特化は行わない．
(4) メモリ参照は，%**sp** と %**fp** からの相対アクセスのみを許す．
(5) SPARC アーキテクチャのレジスタ・ウインドウは扱わない．

以降では，それぞれ，2 節で program-point sensitivity について，3 節で本稿で扱うアセンブリ言語の仕様とその制限について述べ，4 節で本稿のアルゴリズムで用いる数学的な定義を行う．また，それぞれ，5 節で本稿のプログラム特化アルゴリズムで必要な制御フロー解析について，6 節で制約解消による program-point sensitive な束縛時解析のアルゴリズムについて，7 節では二段階の簡単な手順で**生成拡張**[1),12),13)]（プログラムの静的な入力値から，特化されたプログラムを生成するプログラム）を生成するアルゴリズムについて説明する．さらに，8 節で本稿のアルゴリズムを用いて行った評価実験，9 節で関連研究，10 節でまとめを述べる．

2. Program-point sensitivity

束縛時解析の目的は，予め決められた制約を満たす束縛時値の割当てのうち，動的となるものが最も少ない割当てを求めることである．[12]

program-point sensitive な束縛時解析 [10] は，*flow sensitive* な束縛時解析 [11] や，*polyvariant* な束縛時解析 [12] と同義であり，変数の束縛時値をプログラムポイント毎に計算することである．従って，一つの変数があるプログラムポイントでは静的になり，別のプログラムポイントでは動的になる．一方，program-point insensitive な束縛時解析は，*monovariant* な束縛時解析 [12] とも呼ばれており，各変数の束縛時値をプログラム全体で一つ計算することである．従って，ある変数がプログラム中で一箇所でも動的ならば，全ての出現で動的になる．

アセンブリ言語上でプログラム特化を行う場合，次のような点から program-point sensitive な解析を行う必要がある．

- コンパイラのレジスタ割当ては，ソースプログラム中の多数の変数を，アセンブリプログラムの少数のレジスタに割り当てる．そのため，ソースプ

ログラム中で動的である変数と，静的である変数とが，同じレジスタに割り当てられた場合，program-point insensitive な解析では，そのレジスタの全ての出現を動的にしてしまう．一方，program-point sensitive な解析では，同じレジスタであっても，プログラムポイント毎に束縛時値を割り当てるので，望ましい結果が得られることが多い．

- 高水準言語上でのプログラム特化では，生成する**残余プログラム**（*residualized program*：特化されたプログラム）の性能に関わらず，バックエンド・コンパイラの最適化によって，プログラムを効率的に実行できることが多い．しかし，アセンブリ言語上でプログラム特化を行う場合，そのような最適化を期待できない．そのため，特化時に効率の良い残余プログラムを生成するか，特化後に改めて残余プログラムを最適化する必要がある．program-point sensitive な解析を行った場合，program-point insensitive な解析を行った場合よりも，効率の良い残余プログラムを生成することが出来る．

例として，自然数のべき乗を計算する power 関数のアセンブリプログラムに対して，program-point sensitive な束縛時解析を行った結果を**図 1** に，program-point insensitive な解析を行った結果を**図 2** に示す．power 関数の入力値 x と n は，それぞれ %**i0** と %**i1** に渡され，返り値 x^n は %**i0** に渡されるとする．解析の入力は，%**i0** が動的，%**i1** が静的であるとする．図の下線は動的であることを表している．詳しくは，6 節と 7 節で述べる．

このプログラムではレジスタ %**o1** にソースプログラム中の変数 x と n の両方の値が割り当てられている．そのため，program-point insensitive な解析では，プログラムのほとんど全てが動的とされてしまい（図 2），特化が全く行われない．一方，program-point sensitive な解析では望ましい結果が得られ（図 1），この結果を用いることで，power 関数のループを n 個の乗算命令の列に置き換えた残余プログラムが生成される．

3. 言語とその制限

本節では，本稿で扱うアセンブリ言語の構文とその制限について述べる．本稿では，SPARC アーキテクチャ [19] のサブセットを仮定する．

レジスタは，スタック・ポインタ（%**sp**）とフレーム・ポインタ（%**fp**）を含む 32 個の**整数型レジスタ**，**プロセッサ・ステータス・レジスタ**（*PSR*）と**プログラムカウンタ**（*PC*）のみを扱う．整数型レジスタの仕様の詳細は，**図 3** の通りである．PSR は，条件判定のための **icc**（*integer condition code*）フィールドを含む．

本稿で扱う命令は，ロード・ストア命令，四則演算命令，比較命令，分岐命令と，リターン命令である．比較命令は，比較結果を **icc** フィールドに書き込み，分岐命令は，**icc** フィールドの状態によって分岐先を決定する．リターン命令は，レジスタ %**i0** の値を呼び出し側へ戻す．分岐命令とリターン命令を合わせて，制御転送命令 [19] と呼ぶ．

アセンブリ言語に対応するアーキテクチャについて，本稿で設ける制限に関する説明を行う．

(1) レジスタは，整数型レジスタのみを扱う．浮動小数点レジスタへの拡張は容易であるため，ここでは扱わない．

(2) 制御転送命令の遅延スロットは，スロット内の命令を制御転送命令の前に移動し，遅延スロットはないものとして扱う．処理後に，制御転送命令の直前の命令を遅延スロットに移動することは容易なので，以下では遅延スロットを考慮しない．本文中に例示したアセンブリプログラムでは，読み易さのために，制御転送命令の遅延スロットを **nop** 命令で埋めてある．

(3) 手続き呼出しを行わない．これは，アルゴリズムを手続き内解析にしたためである．手続き間解析に拡張した場合でも，同様の解析アルゴリズムを用いることが出来る．

(4) エイリアス解析を行っていないため，メモリ参照に関してエイリアスが存在しないプログラムのみを扱う．その為，**ld** と **st** 命令は，%**sp** 相対と %**fp** 相対でのメモリ参照（すなわち，%**sp** $\pm$ *imm* と %**fp** $\pm$ *imm* の形でのメモリ参照）のみに限る．エイリアス解析を行うことで，メモリ参照に関してエイリアスが存在するプログラムを扱うようにアルゴリズムを拡張することが可能である．また，命令によるメモリ参照は，%**sp**$+64$ や %**fp**-20 等，4 バイト境界での参照のみ許すことにする．

(5) SPARC アーキテクチャのレジスタ・ウインドウは扱わない．レジスタ・ウインドウを扱うように拡張した場合でも，本稿の特化アルゴリズムは同様に用いることが出来る．

図 4 に本稿で扱うアセンブリ言語の構文を表す．

```
0  power :                  10       cmp %o1,-1              20       b .LL2
1        st %i0,[%fp+68]    11       bne .LL4                21       nop
2        st %i1,[%fp+72]    12       nop                     22  .LL3 :
3        mov 1,%o0          13       b .LL3                  23       ld [%fp-20],%o0
4        st %o0,[%fp-20]    14       nop                     24       mov %o0,%i0
5  .LL2 :                   15  .LL4 :                       25       b .LL1
6        ld [%fp+72],%o1    16       ld [%fp-20],%o0         26       nop
7        add %o1,-1,%o0     17       ld [%fp+68],%o1         27  .LL1 :
8        mov %o0,%o1        18       mul %o0,%o1,%o0         28       ret
9        st %o1,[%fp+72]    19       st %o0,[%fp-20]         29       nop
```

図 1 program-point sensitive な解析の結果
Fig. 1 Result of the program-point sensitive analysis

```
0  power :                  10       cmp %o1,-1              20       b .LL2
1        st %i0,[%fp+68]    11       bne .LL4                21       nop
2        st %i1,[%fp+72]    12       nop                     22  .LL3 :
3        mov 1,%o0          13       b .LL3                  23       ld [%fp-20],%o0
4        st %o0,[%fp-20]    14       nop                     24       mov %o0,%i0
5  .LL2 :                   15  .LL4 :                       25       b .LL1
6        ld [%fp+72],%o1    16       ld [%fp-20],%o0         26       nop
7        add %o1,-1,%o0     17       ld [%fp+68],%o1         27  .LL1 :
8        mov %o0,%o1        18       mul %o0,%o1,%o0         28       ret
9        st %o1,[%fp+72]    19       st %o0,[%fp-20]         29       nop
```

図 2 program-point insensitive な解析の結果
Fig. 2 Result of the program-point insensitive analysis

global	: %**g0** — %**g7**(%**r0** — %**r7**)	%**g0**(%**r0**) は常に値 0 を保持する
out	: %**o0** — %**o7**(%**r8** — %**r15**)	%**o6**(%**r14**) はスタック・ポインタ %**sp** を兼ねる
local	: %**l0** — %**l7**(%**r16** — %**r23**)	
in	: %**i0** — %**i7**(%**r24** — %**r31**)	入力パラメータ %**i6**(%**r30**) はフレーム・ポインタ %**fp** を兼ねる %**i0**(%**r24**) は呼び出し側への戻値を保持する

図 3 整数型レジスタの仕様
Fig. 3 Specification of integer registers

```
program     ::=  label ':' sequence
sequence    ::=  command*
command     ::=  instruction | label ':' instruction
instruction ::=  ld '[' address ],'reg_d | st reg_s ',[' address ']' |
                 mov reg_s ','reg_d | mov imm','reg_d |
                 add reg_s1 ','reg_s2 ','reg_d | add reg_s ','imm','reg_d |
                 sub reg_s1 ','reg_s2 ','reg_d | sub reg_s ','imm','reg_d |
                 mul reg_s1 ','reg_s2 ','reg_d | mul reg_s ','imm','reg_d |
                 div reg_s1 ','reg_s2 ','reg_d | div reg_s ','imm','reg_d |
                 cmp reg_s1 ','reg_s2 | cmp reg_s1 ','imm |
                 b label | be label | bne label | bg label | bge label |
                 bl label | ble label | ret | nop
address     ::=  reg | reg_1 '+' reg_2 | reg '+' imm | reg '-' imm | imm
reg         ::=  %g0(%r0) | ... | %g7(%r7) | %o0(%r8) | ... | %sp(%r14) | %o7(%r15) |
                 %l0(%r16) | ... | %l7(%r23) | %i0(%r24) | ... | %fp(%r30) | %i7(%r31)
label ∈ Identifier
imm ∈ Z
```

図 4 アセンブリ言語の構文
Fig. 4 Syntax of the assembly language

4. 数学的な定義

本節では，本稿に必要な数学的定義を行う．X と Y を集合とする．$x_0, \ldots, x_n \in X$，$y_0, \ldots, y_n \in Y$ として，X と Y 上の部分関数 $f : X \longrightarrow Y$ を次のように表す．

$$\begin{aligned} f &= [x_0 \mapsto y_0, \ldots, x_n \mapsto y_n] \\ &= \bigcup_{i \in \{0,\cdots,n\}} (x_i \mapsto y_i). \end{aligned}$$

部分関数 f の定義域を，$\mathbf{dom}(f)$ と表す．$x \in X$ に対して，f の像が未定義の場合，$f(x) = \varepsilon$ と表す．**図 5** に，f と g を X と Y 上の部分関数として，本

$$(f \circ g)(x) = \begin{cases} g(x) & x \in \mathbf{dom}(g) \\ f(x) & x \notin \mathbf{dom}(g) \end{cases}$$

$$(f \setminus \{x_0, \ldots, x_n\})(x) = \begin{cases} \varepsilon & x \in \{x_0, \ldots, x_n\} \\ f(x) & x \notin \{x_0, \ldots, x_n\} \end{cases}$$

図 5 部分関数上の演算
Fig. 5 Operations on partial functions

稿で必要な部分関数上の演算 $f \circ g$（**更新**，*updating*）と，$f \setminus \{\_\}$（**制約**，*restriction*）を定義する．

5. 制御フロー解析

本節では，本稿の束縛時解析に必要なフローグラフに基づいた制御フロー解析について説明する．非構造的な制御フローを許す言語上での program-point sensitive な束縛時解析に必要な制御フロー解析を示したのは，本稿が初めてである．ただし，制御フロー解析のアルゴリズムは，通常のものを用いれば十分なので，本稿では特に述べない [2),15)]．

メモリ空間中のテキスト領域のアドレスの集合を $InstAddr$ とし，$InstAddr$ の要素を**命令アドレス**（*instruction address*）と呼ぶことにする．プログラム P の実行アドレス全体の集合を $Dom(P) \subset InstAddr$ と表す．pc をプログラムカウンタとし，$pc \in Dom(P)$ とする．pc 番地の次番地を $pc{+}1$ とする．便宜的に，プログラムは，常に 0 番地から実行されるものとする．また，プログラムの制御は，ret 命令の実行後，$\perp$ 番地に移動するものとする．プログラム P の pc 番地の命令を $P(pc)$ によって表す．

icc フィールドを含むレジスタ全体の集合を Reg とし，$reg \in Reg$ とする．メモリ空間中のデータ領域とスタック領域のアドレスの集合を $MemAddr$ とし，$address \in MemAddr$ とする．アセンブリコード中に出現する [%**sp**+64] や [%**fp**−20] 等は，$MemAddr$ の要素である．以降，集合 Reg と集合 $MemAddr$ の和集合の要素を**変数**と呼び，$x, y, \cdots \in (Reg \cup MemAddr)$ などと表す．

プログラム P 中の pc 番地の後続番地（直後に実行され得る命令のアドレス）の集合を $succ_P(pc)$，先行番地（直前に実行され得る命令のアドレス）の集合を $pred_P(pc)$ と定義する．**表 1** に，pc 番地の命令が使用する変数の集合 $Use(pc)$ と，定義する変数 $def(pc)$ を命令毎に定義する．**b** と **nop** は変数を使用しないので，$Use(pc)$ は空集合（$\emptyset$）であり，**b***cc*，**b**，**ret** と **nop** は変数を定義しないので，$def(pc)$ は定義されない（空文字（ε）で表す）．

次に，プログラムを**基本ブロック**（*basic block*）に分割し，**フローグラフ**（*flow graph*）を作成する．フローグラフは，基本ブロックをノード（頂点）とし，基本ブロック間の制御フロー関係を辺で表した有向グラフである．本稿では，プログラム P のフローグラフを，$G(P) = \langle N(P), E(P) \rangle$ と表す．$N(P)$ は基本ブロックの集合であり，$E(P)$ は基本ブロック間の制御フロー関係を表した辺の集合である．頂点 A から頂点 B への辺を，$A \to B \in E(P)$ と表す．

ノード $A \in N(P)$ の後続ノードの集合 $SUCC(A) \subseteq N(P)$ を定義する．

$$SUCC(A) = \{N \in N(P) \mid A \to N \in E(P)\}.$$

$A_0, \cdots, A_n \in N(P)$ かつ $A_0 \to A_1, \cdots, A_{n-1} \to A_n \in E(P)$ の時，$A_0 \to \cdots \to A_n$ は，頂点 A_0 から頂点 A_n への**路**（**パス**，*path*）と言う．この時，頂点 A_0 から頂点 A_n へ**到達可能**（*reachable*）であると言い，$A_0 \overset{G(P)}{\to} A_n$ と書く．$\overset{G(P)}{\to}$ を形式的に定義すると，$\forall A, B, C \in N(P)$ に対して，以下のようになる．

(1) $A \to B \in E(P) \Rightarrow A \overset{G(P)}{\to} B$.
(2) $A \overset{G(P)}{\to} B \wedge B \overset{G(P)}{\to} C \Rightarrow A \overset{G(P)}{\to} C$.

フローグラフの頂点 A から頂点 B に至る全ての路が，必ず頂点 C を通り，かつ $B \neq C$（$A = C$ はよい）の時，頂点 A を通る路において，頂点 C は頂点 B を**厳密に支配する**（*strictly dominate*）と言い，「$C\ sdom_A\ B$」と書く．

さらに，フローグラフ $G(P)$ から**強連結性**（*strongly connected*）を満たす部分グラフを，全て抽出する．強連結性を満たす部分グラフとは，部分グラフ内の任意のノードから部分グラフ内の全てのノードに到達可能であるような部分グラフである [15)]．例えば，辺を持たず唯一の頂点からなる部分グラフや，自然ループ以外のループを含めた全てのループを構成する部分グラフは，強連結性を満たす部分グラフである．

強連結性は，自然ループ以外のループを持つプログラムを制御フロー解析の対象とする場合に良く用いられる性質である [15)]．本稿では，自然ループ以外のループを持ち得るプログラムを特化の対象とするため，強連結性を考慮する．ただし，本稿では辺を持たず唯一の頂点からなる部分グラフは，強連結性を満たす部分グラフと見なさない．以降，フローグラフ内の強連結性を満たす部分グラフを，**ループ**（*loop*）と呼ぶこと

表 1 $Use(pc)$ と $def(pc)$ との定義
Table 1 Definitions of $Use(pc)$ and $def(pc)$

$P(pc)$	$Use(pc)$	$P(pc)$	$def(pc)$
st, mov, add 等（四則演算）, **cmp**	$\{ reg_s \}$ or $\{ reg_{s1}, reg_{s2} \}$	**ld, mov, add** 等	reg_d
ld	$\{ address \}$	**st**	$address$
ret	{%**i0**}	**cmp**	**icc**
bcc	{**icc**}	**b**, **b**cc, **ret**, **nop**	ε
b, **nop**	$\emptyset$		

にする．

図 6 に，本稿で必要な制御フロー解析によって得られる集合を定義する．本稿では，基本ブロックは命令アドレスの集合であるとする．すなわち，pc 番地にある命令が基本ブロック A に含まれる場合，$pc \in A$ と表す．

関数 $tail(A)$ は，基本ブロック A の末尾の命令アドレスを返す関数である．関数 $head(A)$ は，基本ブロック A の先頭の命令アドレスを返す関数である．図 6 で，A，B，C と X は基本ブロックを表し，pc は命令アドレスを表す．

以下，図 6 の式 (1) から式 (9) について，それぞれ説明する．

（式 1） $Reachable_P(A)$ は，フローグラフ $G(P)$ において，ノード A とノード A から到達可能な全てのノードの集合である．

（式 2） $Intersect_P(A, B)$ は，ノード A から到達可能なノードの集合と，ノード B から到達可能なノードの集合の共通部分である．

（式 3） $Join_P(A, B)$ は，ノード A とノード B とを通るパスが合流するノードの集合である．合流するノードとは，異なる二つのパスが出会うノードのことであり，出会った以降のノードは含まれない．以降，合流するノードを**合流ノード**と呼ぶ．

式 3 は，ノード X が $Join_P(A, B)$ に含まれるならば，$X \in Intersect_P(A, B)$ であり，かつ，A と B から X に至るパス上で X を厳密に支配する $Intersect_P(A, B)$ 内のノードがないことを表す．

（式 4） $Route_P(A, B)$ は，ノード A からノード B に至り得るパス上にある全てのノードの集合である．

（式 5） $CondBranch_P$ は，条件付き分岐命令を含む基本ブロックの集合である．以降，$CondBranch_P$ に含まれるノードを**分岐ノード**と呼ぶ．

（式 6） $Merge_P(X)$ は，分岐ノード X から出た 2 つのパスが合流し得る全てのノードの集合である．

（式 7） $BtoM_P$ は，プログラム P の全ての分岐ノード A と，分岐ノード A に対応する全ての合流ノード B に対する，以下の 3 組全体の集合である．

(1) $tail(A)$.

(2) $head(B)$.

(3) $\bigcup_{C \in Route_P(A,B)} (\bigcup_{pc \in C} \{def(pc)\})$.

(1) は分岐ノード A の末尾の命令アドレスである．以降，このような命令アドレスを**制御の分岐点**と呼ぶ．(2) は分岐ノード A に対応する合流ノード B の先頭アドレスである．以降，このような命令アドレスを**制御の合流点**と呼ぶ．(3) は分岐ノード A の直後から，対応する合流ノード B の直前に至る間に定義され得る変数全体の集合である．

（式 8） SCS_P は，フローグラフ $G(P)$ 内の全てのループ（すなわち，強連結性を満たす部分グラフ）の集合である．辺を持たず一つの頂点のみからなる部分グラフは，SCS_P に含まれない．

以降，ノード $X \in N(P)$ とループ $\Sigma \in SCS_P$ に対して，$X \in \Sigma$ によって，ノード X がループ Σ に含まれることを表すとする．

（式 9） $ExitScs_P$ は，あるループ $\Sigma \in SCS_P$ に含まれる分岐ノード $X \in N(P)$ のうち，以下の条件を満たすものの末尾の命令アドレスの集合である．

- 分岐ノード $X \in N(P)$ に対応する合流ノード $Y \in N(P)$ が，X を含むループ $\Sigma \in SCS_P$ に含まれない．

以降，このような条件を満たす分岐ノード X を，**ループΣ の出口ノード**と呼ぶ．さらに，ループ Σ の出口ノードの末尾の命令アドレス（すなわち，$ExitScs_P$ の要素である命令アドレス）を，**ループΣ の出口点**と呼ぶ．. 一般に，ループは複数の出口ノードを持ち，ループの出口点には条件付分岐命令が存在する．

6. 束縛時解析

本節では，5 節で行った制御フロー解析の結果を用いて，program-point sensitive な束縛時解析のアルゴリズムを，制約解消によって定式化する．

program-point sensitive な束縛時解析は，Hornof らによって，文献 10) で初めて導入された束縛時解析である．Hornof らは，データフロー方程式を用いて，program-point sensitive な束縛時解析を定式化した．制約解消を用いて，program-point sensitive な束縛

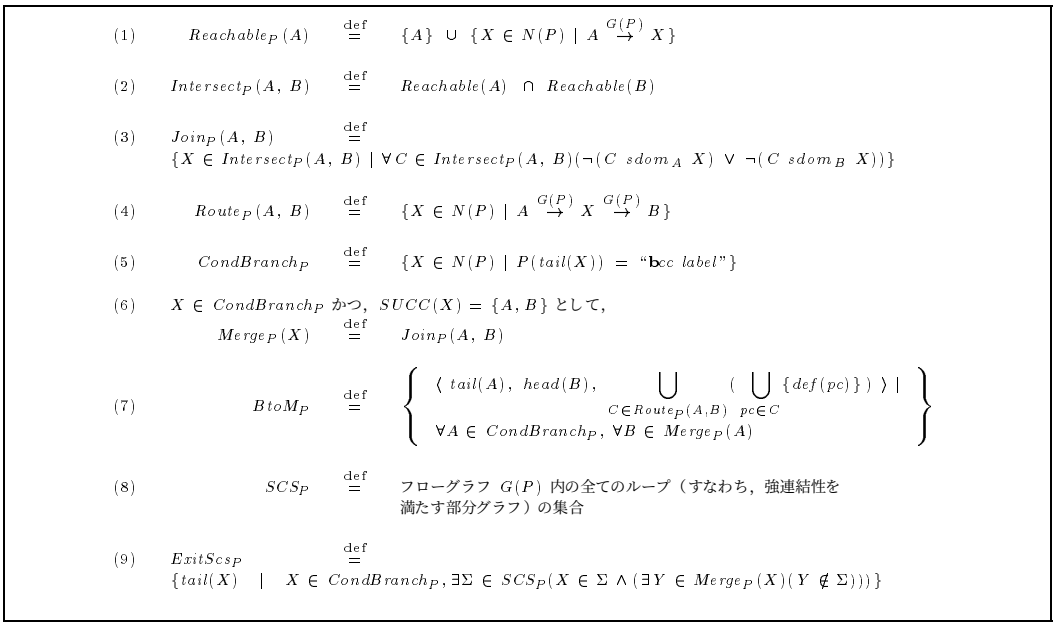


図 6 制御フロー解析によって得られる集合の定義
Fig. 6 Definitions of the sets calculated by the flow analysis

時解析を定式化したのは，本稿が初めてである．

制約解消による束縛時解析は，以下の三つの順番で行われる [9),12),16)]．

(1) **制約生成**（*constraints generation*）
(2) **正規化**（*normalization*）
(3) **制約解消**（*constraints solving*）

本稿では制約生成の規則のみを扱う．正規化と制約解消については，文献 9) のアルゴリズムと同等のものを用いるので本稿では省略する．本稿の定式化における正規化と制約解消についての詳しい説明は，文献 18) を参照されたい．

また，本節の束縛時解析のアルゴリズムは，既存の C 言語上での束縛時解析のアルゴリズム [1),13)] と異なり，変数に対する束縛時解析と，命令に対する束縛時解析を別々に処理する．命令に対する束縛時解析は，**行動解析**（*action analysis*）[6),7)] と呼ばれ，命令が特化器によってどのように処理されるかを決定するものである．

以降，本節では，6.1 節で束縛時解析の準備として，使用と定義の束縛時値について定義し，6.2 節で制約生成の規則，6.3 節で命令に対する行動値の割当てについて説明し，6.4 節で束縛時解析の例を示す．

6.1 使用と定義の束縛時値

S と D は，それぞれ静的と動的の束縛時値を表すものとし，束縛時値の集合を $Bt = \{S, D\}$ とする．集合 Bt 上に，順序関係 $S < D$ を定義し，$\sqcup$ を関係 $<$ 上の上限とする．$t_0, \cdots, t_n \in Bt$ の上限を，以下のように表す。

$t_0 \sqcup \cdots \sqcup t_n$，又は $\bigsqcup_{i \in \{0,\cdots,n\}} t_i$.

program-point sensitivity を実現するために，二つの部分関数 UB と DB を定義する．

$UB, DB : InstAddr \rightarrow (Reg \cup MemAddr \rightarrow Bt)$

UB と DB は，命令アドレスの集合 $InstAddr$ から，変数の集合 $(Reg \cup MemAddr)$ から Bt への部分関数 $(Reg \cup MemAddr \rightarrow Bt)$ への部分関数である．

プログラムの命令アドレス pc と，プログラム中の変数 x に対して，$UB(pc)(x)$ は，pc 番地の命令で**使用される**変数 x の束縛時値を表し，$DB(pc)(x)$ は，pc 番地の命令で**定義される**変数 x の束縛時値を表す．部分関数 UB を変数の**使用の束縛時**と呼び☆，部分関数 DB を変数の**定義の束縛時**と呼ぶ☆☆．これ以降では，$UB(pc)(x)$ と $DB(pc)(x)$ を，それぞれ $UB_{pc}(x)$ と $DB_{pc}(x)$ と略記する．

使用の束縛時 UB と定義の束縛時 DB について，もう少し説明する．

$UB_{pc}(x)$ と $DB_{pc}(x)$ に関する上記の説明を言い換えると，$UB_{pc}(x)$ は，pc 番地の命令のソースオペランドにある変数 x の束縛時値を表し，$DB_{pc}(x)$ は，pc

☆ UB：**B**inding-time for variable **U**sage
☆☆ DB：**B**inding-time for variable **D**efinition

番地の命令のデスティネーションオペランドにある変数 x の束縛時値を表すと言える．すなわち，次のコード片に対して，$UB_{pc}(\%\mathbf{i0})$ は，第一オペランドにある $\%\mathbf{i0}$ の束縛時値であり，$DB_{pc}(\%\mathbf{i0})$ は，第三オペランドにある $\%\mathbf{i0}$ の束縛時値である．

pc : **add** $\%\mathbf{i0}, \%\mathbf{i1}, \%\mathbf{i0}$

次に，$UB_{pc}(x)$ と $DB_{pc}(x)$ が，それぞれ静的と動的であることの意味を説明する．

- pc 番地の命令で変数 x が使用されているとし，変数 x の使用に対応する全ての変数 x の定義が，それぞれ $a_0, \ldots, a_n$ 番地の命令で行われているとする．$UB_{pc}(x)$ が静的であるとは，$DB_{a_0}(x), \ldots, DB_{a_n}(x)$ が全て静的であることを意味する．$UB_{pc}(x)$ が動的であるとは，$DB_{a_0}(x), \ldots, DB_{a_n}(x)$ の内，少なくとも一つが動的であることを意味する．
- pc 番地の命令で変数 x が定義されているとし，変数 x の定義が到達する範囲内にある全ての変数 x の使用が，それぞれ $a_0, \ldots, a_n$ 番地の命令で行われているとする．$DB_{pc}(x)$ が静的であるとは，$UB_{a_0}(x), \ldots, UB_{a_n}(x)$ が全て静的であることを意味する．$DB_{pc}(x)$ が動的であるとは，$UB_{a_0}(x), \ldots, UB_{a_n}(x)$ の内，少なくとも一つが動的であることを意味する．

集合 Bt 上の順序関係 $\leq$ を，UB と DB 上に拡張する．$\forall pc \in InstAddr$ に対して，$f, g \in UB(pc)$ として，以下の関係を定義する．$DB(pc)$ についても同様である．

$$
\begin{aligned}
& f \leq g \stackrel{\text{def}}{\Longleftrightarrow} \\
& (\mathbf{dom}(f) \subseteq \mathbf{dom}(g)) \\
& \wedge (\forall x \in \mathbf{dom}(f)(f(x) \leq g(x)))
\end{aligned}
$$

$f \leq g$ の時，f は g と等しいかより小さい（又は，g は f と等しいかより大きい）と言う．

6.2 制約生成

ここでは，program-point sensitive な束縛時解析のための制約生成規則を定式化する．制約は等号制約（$=$）と不等号制約（$\geq$）からなる．本稿では，制約の生成規則を $\mathcal{C}_0 \triangleq C_1$ の形で表す．**図 7** に制約の生成規則を表す．

以下，図 7 の式 (1) から式 (6) について，それぞれ説明する．

（式 1） $\mathcal{C}_{\mathcal{UB}}$ は，自番地とその全ての後続番地とにおける使用の束縛時 UB の関係を次のように規定した制約である．

- pc 番地の後続番地 a における使用の束縛時 UB_a は，pc 番地における使用の束縛時 UB_{pc} に対して，UB_{pc} における pc 番地で定義される変数 $def(pc)$ の束縛時値を，pc 番地で使用する全ての変数の束縛時値の上限（$\bigsqcup UB_{pc}(x), \forall x \in Use(pc)$）で更新したものと等しいか，より大きい．

例えば，$P(pc)$ =“**ld** $[address], reg_d$” ならば，$succ(pc) = \{pc + 1\}$ なので，以下のようになる．

$$
\begin{aligned}
& \mathcal{C}_{\mathcal{UB}}(pc) \triangleq \\
& UB_{pc+1} \geq UB_{pc} \circ [reg_d \mapsto UB_{pc}(address)]
\end{aligned}
$$

pc 番地の後続番地が複数個あるならば，それぞれに対する制約の論理積を取る．

（式 2） $\mathcal{C}_{Side\text{-}effect}$ は，動的な条件付分岐命令による制御の分岐点から，対応する合流点までの間に存在する静的な代入命令（*static assignment under dynamic control*[12]）を扱うための制約である．条件付分岐命令が動的であるとは，条件付分岐命令のあるアドレスにおける **icc** の使用の束縛時値が動的であることを表す．以降，動的な条件付分岐命令による制御の分岐点から対応する合流点に至る制御フローを，**動的な制御**（*dynamic control*）と呼ぶ．動的な制御の範囲内では，制御フローパスが特化時に一通りに決定できず，実行可能な制御フローパスが複数通り存在する．

$\mathcal{C}_{Side\text{-}effect}$ は，次の事柄を表した制約である．

- $\forall \langle a, b, \{x_0, \cdots, x_n\} \rangle \in BtoM_P$ に対して，制御の分岐点 a に対応する制御の合流点 b における使用の束縛時 UB_b は，a 番地から b 番地に至る間に定義され得る全ての変数 $x_0, \cdots, x_n$ を，a 番地における **icc** の使用の束縛時値 $UB_a(\mathbf{icc})$ に対応付ける部分関数と等しいか，より大きい．

すなわち，a 番地にある条件付分岐命令が動的ならば，b 番地における変数 $x_0, \cdots, x_n$ の使用の束縛時値（$UB_b(x_i), i \in \{0, \cdots, n\}$）は，動的でなければならないことを表している．

このような制約が必要な理由は，動的な制御中で定義された変数の合流点における値は，特化時には決定できず，実行時まで明らかでないからである．従って，動的な制御中で定義される変数は，静的な代入命令によって定義された変数であっても，合流点における束縛時値は動的でなくてはならない．

（式 3） $\mathcal{C}_{SCS}$ は，**動的なループ**（*dynamic loop*）内に存在する静的な代入命令を扱うための制約である．動的なループとは，出口ノードに動的な条件付分岐命令を持つループである．

動的なループは，繰り返し回数が実行時まで明らかでないので，ループ内で静的な代入命令によって定義さ

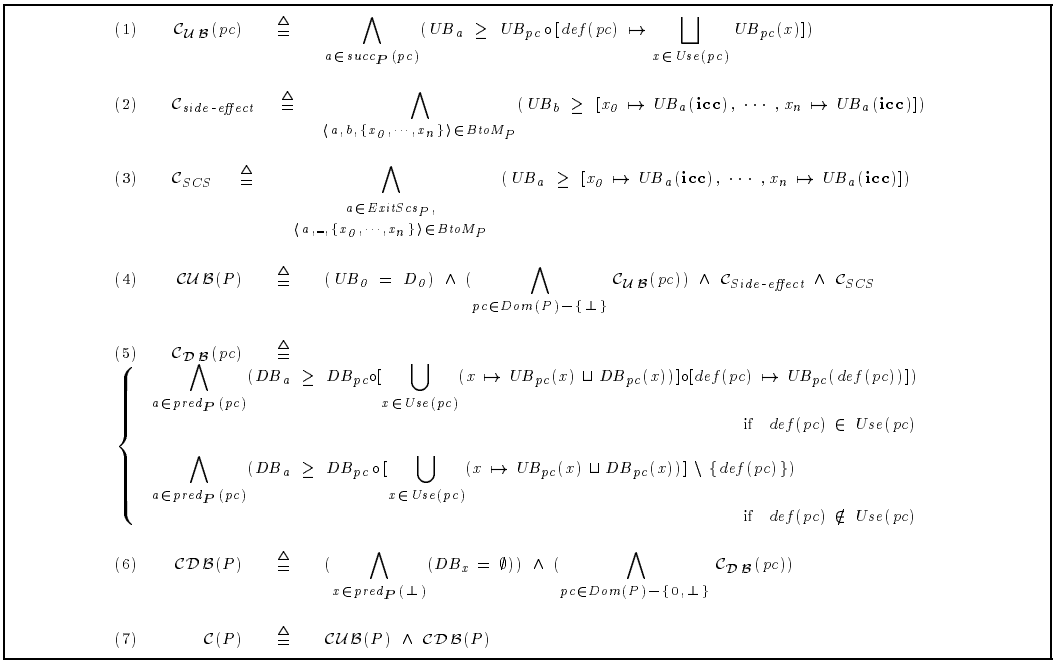


図 7 プログラム P の制約生成規則
Fig. 7 Generating rules of constraints for program P

れる変数の値であっても，特化時には明らかでない．従って，動的なループ内で静的な代入命令によって定義される変数のループ内での束縛時値は，動的でなくてはならない．

$\mathcal{C}_{SCS}$ は，次の事柄を表した制約である．

- $\forall a \in ExitScs_P$ と $\forall \langle a, \_, \{x_0, \cdots, x_n\}\rangle \in BtoM_P$ に対して，ループの出口点 a における使用の束縛時 UB_a は，ループ内で定義され得る全ての変数 $x_0, \cdots, x_n$ を，a 番地における **icc** の使用の束縛時値 $UB_a(\mathbf{icc})$ に対応付ける部分関数と等しいか，より大きい．

すなわち，ループの出口点 a にある条件付分岐命令が動的ならば，a 番地における変数 $x_0, \cdots, x_n$ の使用の束縛時値（$UB_a(x_i), i \in \{0, \cdots, n\}$）は，動的でなければならないことを表している．

制約 $\mathcal{C}_{SCS}$ と制約 $\mathcal{C}_{Side\text{-}effect}$ は，形は似ているが，意味は異なる．$\mathcal{C}_{Side\text{-}effect}$ は，動的な制御の合流点以降において，静的な代入を動的にするための制約であるのに対して，$\mathcal{C}_{SCS}$ は，動的なループ内を静的な代入を動的にするための制約である点に注意されたい．

（式 4） $\mathcal{CUB}(P)$ は，上記の式 1，式 2 と，式 3 の制約を論理積で結合した制約であり，プログラム P 全体の使用に関する制約である．$\mathcal{CUB}(P)$ を**使用の制約**（*use constraint*）と呼ぶ．使用の制約 $\mathcal{CUB}(P)$ は，以下の要素の論理積からなる．

(1) 等号制約 $(UB_0 = D_0)$．ただし，D_0 はプログラム P の入力パラメータに対する束縛時値の割当て（**初期分割**，*initial division*[12]）を表す．
(2) $\bot$ 番地を除く $Dom(P)$ の全ての命令アドレスに対する制約 $\mathcal{C}_{\mathcal{UB}}$ の論理積．
(3) 動的な制御を扱うための制約 $\mathcal{C}_{Side\text{-}effect}$．
(4) 動的なループを扱うための制約 $\mathcal{C}_{SCS}$．

使用の制約 $\mathcal{CUB}(P)$ を正規化し，制約解消することによって，プログラム P 中の各命令のソースオペランドにある変数の束縛時値を決定することが出来る．

（式 5） $\mathcal{C}_{\mathcal{DB}}$ は，自番地とその全ての先行番地とにおける定義の束縛時 DB の関係を規定した制約である．

- pc 番地の先行番地 a における定義の束縛時 DB_a は，pc 番地における定義の束縛時 DB_{pc} において，以下の二つの操作を行って得られたものと等しいか，より大きい．
  (1) 一番目の操作では，pc 番地で使用される全ての変数 $x \in Use(pc)$ の束縛時値を，pc 番地における変数 x の使用の束縛時値 $UB_{pc}(x)$ と，定義の束縛時値 $DB_{pc}(x)$ の上限で更新する．
  (2) 二番目の操作は，pc 番地の命令が定義する変数と同じ変数を使用する場合（$def(pc) \in Use(pc)$）と，pc 番地の命令が定義する変数と同じ変数を使用しない場合（$def(pc) \notin Use(pc)$）に場合分けされる．

$def(pc) \in Use(pc)$ の場合は，pc 番地で定義される変数 $def(pc)$ の束縛時値を，pc 番地における $def(pc)$ の使用の束縛時値 $UB_{pc}(def(pc))$ で更新する．$def(pc) \notin Use(pc)$ の場合は，$def(pc)$ を DB_a の定義域から制限する．

例えば，$P(pc)$ =“**add** %**o0**,%**l0**,%**o0**” の場合は，pc 番地の命令が定義する変数と同じ変数を使用するので，以下のようになる．

$$\begin{aligned}&\mathcal{C}_{\mathcal{DB}}(pc) \triangleq\\ &DB_{pc-1} \geq\\ &(DB_{pc} \circ [\%\mathbf{o0} \mapsto UB_{pc}(\%\mathbf{o0}) \sqcup DB_{pc}(\%\mathbf{o0}),\\ &\%\mathbf{l0} \mapsto UB_{pc}(\%\mathbf{l0}) \sqcup DB_{pc}(\%\mathbf{l0})])\\ &\circ [\%\mathbf{o0} \mapsto UB_{pc}(\%\mathbf{o0})]\end{aligned}$$

これを簡約すると，以下のようになる．

$$\begin{aligned}&\mathcal{C}_{\mathcal{DB}}(pc) \triangleq\\ &DB_{pc-1} \geq DB_{pc} \circ [\%\mathbf{o0} \mapsto UB_{pc}(\%\mathbf{o0}),\\ &\%\mathbf{l0} \mapsto UB_{pc}(\%\mathbf{l0}) \sqcup DB_{pc}(\%\mathbf{l0})]\end{aligned}$$

例えば，$P(pc)$ =“**ld** [%**sp** + 64],%**o0**” の場合は，pc 番地の命令が定義する変数と同じ変数を使用しないので，以下のようになる．

$$\begin{aligned}&\mathcal{C}_{\mathcal{DB}}(pc) \triangleq\\ &DB_{pc-1} \geq (DB_{pc} \circ [[\%\mathbf{sp} + 64] \mapsto\\ &UB_{pc}([\%\mathbf{sp} + 64]) \sqcup DB_{pc}([\%\mathbf{sp} + 64])])\\ &\setminus \{\%\mathbf{o0}\}\end{aligned}$$

（式 6） $\mathcal{CDB}(P)$ は，プログラム P 全体の定義に関する制約であり，**定義の制約**（*definition constraint*）と呼ぶ．定義の制約 $\mathcal{CDB}(P)$ は，以下の要素を論理積で結合した制約である．

(1) ⊥ 番地の全ての先行番地 x に対する等号制約 ($DB_x = \emptyset$)．これは，**ret** 命令以降に使用される変数がないことを表している．

(2) 0 と ⊥ 番地を除く $Dom(P)$ の全ての命令アドレスに対する制約 $\mathcal{C}_{\mathcal{DB}}$ の論理積．

定義の制約 $\mathcal{CUB}(P)$ を正規化し，制約解消することによって，プログラム P 中の各命令のデスティネーションオペランドにある変数の束縛時値を決定することが出来る．

（式 7） $\mathcal{C}(P)$ は，使用の制約 $\mathcal{CUB}(P)$ と定義の制約 $\mathcal{CDB}(P)$ の論理積であり，プログラム P 全体の制約である．制約 $\mathcal{C}(P)$ を正規化し，制約解消することによって，プログラム P に存在する変数の全ての出現に対する束縛時値を決定することが出来る．

6.3 命令の行動値

本節までで，各命令で使用または定義される変数の束縛時値の割当てが決定された．本節では，命令そのものに対する**行動値**（*action value*）の割当てを決定する．命令の行動値とは，命令がプログラム特化器によって実際にどのように処理されるのかを表したものである．特化器は，各命令を特化時に完全に評価する（evaluate）か，別の命令に置き換える（rebuild）か，残余コードとして出力する（residualize）かの何れかの処理を行う．

特化器によって，各命令に対して行われる処理を**行動**（*action*）と呼ぶ [7]．Tempo[5] は，束縛時解析で得られた情報を基にして，プログラムの全ての要素に対する行動を決定する**行動解析**（*action analysis*）を行っている [6]．

ここでは，特化器の行動を表す値として，以下の三つを定義する．

- Ev　命令は特化時に完全に評価される．
- Res　命令はそのまま残余コードとして残される．
- Reb　命令は他の命令や命令列に置き換えられる．

表 2 に，pc 番地の命令の行動値を定義する．表 2 の集合 $DynCont_P$ は，プログラム P 中で動的な制御の範囲内にある全ての命令アドレスの集合である．$DynCont_P$ の定義を以下に表す．

$$DynCont_P \stackrel{\text{def}}{=} \left\{ x \in X \;\middle|\; \begin{array}{l} \forall A \in CondBranch_P,\\ \forall B \in Merge_P(A),\\ \forall X \in Route_P(A,B)\\ (UB_{tail(A)}(\mathbf{icc}) = D) \end{array} \right\}$$

ld，**st**，**mov**，**add** と，**cmp** 命令は，各命令が定義する変数の束縛時値が動的ならば Res 行動値を取り，静的ならば Ev 行動値を取る．無条件分岐命令 **b** は，命令が動的な制御の範囲内にあるならば Res 行動値を取り，それ以外の場合は Ev 行動値を取る．条件付分岐命令 **b***cc* は，**icc** の使用の束縛時値が動的ならば Res 行動値を取り，**icc** の使用の束縛時値は静的だが，動的な制御の範囲内にあるならば Reb 行動値を取り，それ以外の場合は Ev 行動値を取る．

ラベル（*label*:）は少し分かりにくいが，自身が動的な条件付き分岐命令の分岐先ラベルか，動的な制御の範囲内にある分岐命令の分岐先ラベルになっているならば Res 行動値を取り，それ以外の場合は Ev 行動値を取る．

6.4 束縛時解析の例

最後に，本節のアルゴリズムを用いて行った power

表 2 *pc* 番地の命令の行動値
Table 2 Action value of the instruction at address *pc*

pc 番地の命令	命令の行動値	
ld, st, mov, add, cmp	*Res*	if $DB_{pc}(def(pc)) = D$
	Ev	if $DB_{pc}(def(pc)) = S$
b	*Res*	if $pc \in DynCont_P$
	Ev	otherwise
b*cc*	*Res*	if $UB_{pc}(\mathbf{icc}) = D$
	Reb	if $UB_{pc}(\mathbf{icc}) = S \wedge pc \in DynCont_P$
	Ev	otherwise
label:	*Res*	if $\exists a \in Dom(P)((P(a) =$ "**b***cc label*" $\wedge\ UB_a(\mathbf{icc}) = D) \vee$ $((P(a) =$ "**b** *label*" $\vee\ P(a) =$ "**b***cc label*") $\wedge\ a \in DynCont_P))$
	Ev	otherwise

```
0   power :                  11          st %o1,[%fp+72]      22          st %o0,[%fp-20]
1           st %i0,[%fp+68]  12          cmp %o1,-1           23          b .L1
2           st %i1,[%fp+72]  13          bne .L3              24          nop
3           mov 1,%o0        14          nop                  25    .L4 :
4           st %o0,[%fp-20]  15    .L2 :                      26          ld [%fp-20],%o0
5           b .L1            16          b .L4                27          mov %o0,%i0
6           nop              17          nop                  28          b .L5
7     .L1 :                  18    .L3 :                      29          nop
8           ld [%fp+72],%o1  19          ld [%fp-20],%o0      30    .L5 :
9           add %o1,-1,%o0   20          ld [%fp+68],%o1      31          ret
10          mov %o0,%o1      21          mul %o0,%o1,%o0      32          nop
```

図 8 power 関数の束縛時解析の結果：%**i0** を動的，%**i1** を静的とした場合
Fig. 8 Result of binding time analysis of power function where %**i0** is dynamic and %**i1** is static

```
0   power :                  11          st %o1,[%fp+72]      22          st %o0,[%fp-20]
1           st %i0,[%fp+68]  12          cmp %o1,-1           23          b .L1
2           st %i1,[%fp+72]  13          bne .L3              24          nop
3           mov 1,%o0        14          nop                  25    .L4 :
4           st %o0,[%fp-20]  15    .L2 :                      26          ld [%fp-20],%o0
5           b .L1            16          b .L4                27          mov %o0,%i0
6           nop              17          nop                  28          b .L5
7     .L1 :                  18    .L3 :                      29          nop
8           ld [%fp+72],%o1  19          ld [%fp-20],%o0      30    .L5 :
9           add %o1,-1,%o0   20          ld [%fp+68],%o1      31          ret
10          mov %o0,%o1      21          mul %o0,%o1,%o0      32          nop
```

図 9 power 関数の束縛時解析の結果：%**i0** を静的，%**i1** を動的とした場合
Fig. 9 Result of binding time analysis of power function where %**i0** is static and %**i1** is dynamic

関数の束縛時解析の結果を示す．ただし，ここでは，7 節のアルゴリズムの簡単化のために，2 節で挙げた power 関数のプログラムを，全ての基本ブロックの先頭にラベルを付け，全ての基本ブロック間の制御移動を無条件又は条件付分岐命令によって行うように変形したものを用いる．詳しくは，7 節で述べる．

図 8 に %**i0** を動的，%**i1** を静的とした場合の power 関数の束縛時解析の結果を，**図 9** に %**i0** を静的，%**i1** を動的とした場合の power 関数の束縛時解析の結果を示す．図 8 と図 9 では，命令とラベルに付けられた下線は命令とラベルが *Res* 行動値を持つことを，各命令のソースオペランドに付けられた下線は使用の束縛時値が動的であることを，各命令のデスティネーションオペランドに付けられた下線は定義の束縛時値が動的であることを表している．下線の付けられていないオペランドや命令は，静的な値を持ち，特化時に完全に評価されることを表している．

7. 生成拡張

本節では，プログラムの生成拡張（*generating extention*）と，二段階の非常に簡単な手順で生成拡張を生成する方法について説明する．

生成拡張とは，特定のプログラムに特化された特化器であり，そのプログラムの静的な入力値を受け取って，残余プログラムを出力するプログラムである．本稿のように，解析フェーズと特化フェーズの二つのフェーズからなるプログラム特化器では，特化そのものに掛かる効率の面から，生成拡張を用いて残余プログラムの生成を行うのが一般的である．[1),12),13)]

本稿では，*pending loop*[1),12),13)] と呼ばれる手法を用いて，生成拡張を実現する．pending loop とは，生成拡張を構成する最外のループの名称である．pend-

ing loop を用いた生成拡張は，間単に実現することが出来る [1),13)]．生成拡張の生成方法については，7.2 節で詳しく説明する．

本稿では，生成拡張アルゴリズムの簡単化のため，生成拡張の生成前に，元のプログラムを以下の二つの条件を満たすように変形しておくとする．

(1) 全ての基本ブロックは，先頭に一つだけラベルを持つ．先頭に複数個のラベルを持ったり，先頭以外の部分にラベルを持ったりしない．

(2) 全ての基本ブロック間の制御移動は，無条件又は条件付分岐命令によって行う．すなわち，全ての基本ブロックは，末尾に無条件分岐命令か条件付分岐命令を持つ．

上の条件 (1) から，基本ブロックとラベルは一対一に対応するので，本節では，ラベルを基本ブロック名と見なす．

以降，本節では，7.1 節で pending loop を用いた生成拡張，7.2 節で生成拡張を生成する方法について説明する．

7.1 pending loop を用いた生成拡張

本節では，pending loop を用いた生成拡張の例を示すことによって，生成拡張の構造と，生成拡張で使用する疑似命令とデータについて説明する．**図 10** と **図 11** に，それぞれ，図 8 と図 9 の束縛時解析の結果に基ずく，power 関数の生成拡張を示す．

pending loop を用いた生成拡張は，元のプログラムを命令毎に生成拡張の命令列に変換することによって生成される．元のプログラムで静的な命令は，そのまま生成拡張の命令となり，動的な命令は，対応する命令を文字列として出力する疑似命令に変換される [1),14)]．詳しい説明は，7.2 節で行う．

pending loop を用いた生成拡張は，以下で説明する *pending list*[1),12),13)] と *marked set*[12)] と言う二つのデータを操作する．

pending list 基本ブロックに付けられたラベル $label$ と，プログラム中の全ての静的変数 $x_0, \cdots, x_n$ の値を集めたタプル（**部分状態**，*partial state*）$s = (s_0, \cdots, s_n)$ との組（*specialized program point*：**特化されたプログラムポイント** と呼ぶ [12)]）$\langle label, s \rangle$ を要素とするスタック型のリスト

marked set 既に特化処理を行って，残余プログラムにコードを出力済みの基本ブロックに対する特化されたプログラムポイント $\langle label, s \rangle$ の集合

本稿の生成拡張は，pending list と marked set を用いて，以下の手順に従って処理を行う．

(1) 元のプログラムの関数名を持つラベル $.func$ と静的変数の入力値からなる部分状態 s_0 から，特化されたプログラムポイント $\langle .func, s_0 \rangle$ を作成し，pending list にプッシュする．その後，pending loop に入る．

(2) pending list が空ならば，特化処理を終わる．

(3) pending list から特化されたプログラムポイント $\langle label, s \rangle$ をポップする．

(4) 部分状態 s に保存された値を取り出し，対応する静的変数に値を復元する．

(5) marked set に $\langle label, s \rangle$ を追加する．

(6) $label$ が示すアドレスへ無条件分岐を行い，そこから特化処理を継続する．

(7) 特化処理中に，元のプログラムの **ret** 命令に出会った場合は，pending loop の先頭に制御を戻す（(2) に戻る）．

(8) 特化処理中に，元のプログラムで動的な分岐命令に出会った場合は，分岐先ラベル $label'$ とその時点での静的変数の部分状態 s' からなる特化されたプログラムポイント $\langle label', s' \rangle$ が，pending list と marked set に含まれているかどうかを検査し，含まれていない場合に限り，pending list に $\langle label', s' \rangle$ をプッシュする．その後，pending loop の先頭に制御を戻す（(2) に戻る）．

ここで，生成拡張が上記のような処理を行うことの技術的な理由について，三つ説明を行う．

(1) **動的な制御の扱い** 動的な制御の範囲内では，実行時に選択可能な制御フローパスが複数通り存在し，そのぞれぞれについて別々に特化処理を行わねばならない．従って，特化処理中に動的な制御に出会った場合は，動的な制御の入口点におけるプログラム中の全ての静的変数の値を保存しておき，一つ目の制御フローパスに沿って特化処理を行った後，制御を入口点に戻し，全ての静的変数の値を復元してから，別の制御フローパスに沿って特化処理を行う．

(2) **同じ残余コードを重複して出力することの回避** 上記の生成拡張では，marked set を用いて，対象とする特化されたプログラムポイントが，既に特化処理済みかどうかの検査を行う．これは，一度特化処理を行って，残余コードを出力済みの基本ブロックを，再び特化処理の対象とすることから回避するためである．もし，このような検査を行わなかったならば，同じ残余コードが複数回出力されるため，残余プログラムのコー

```
0   power_gen :
1         PUSHPENDING .power
2   .pending_loop :
3         ANYPENDING
4         be .exit
5         nop
6         POPPENDING %g1
7         jmp %g1
8         nop
9   .exit :
10        ret
11        nop
12  .power :
13        ST %i0,[%fp+68]
14        st %i1,[%fp+72]
15        mov 1,%o0
16        MOV (%o0),%o0
17        ST %o0,[%fp-20]
18        b .L1
19        nop
20  .L1 :
21        ld [%fp+72],%o1
22        add %o1,-1,%o0
23        mov %o0,%o1
24        st %o1,[%fp+72]
25        cmp %o1,-1
26        bne .L3
27        nop
28  .L2 :
29        b .L4
30        nop
31  .L3 :
32        LD [%fp-20],%o0
33        LD [%fp+68],%o1
34        MUL %o0,%o1,%o0
35        ST %o0,[%fp-20]
36        b .L1
37        nop
38  .L4 :
39        LD [%fp-20],%o0
40        MOV %o0,%i0
41        b .L5
42        nop
43  .L5 :
44        RET
45        b .pending_loop
46        nop
```

図 10 %i0 を動的，%i1 を静的とした場合の power 関数の生成拡張
Fig. 10 Generating extension of power function where %i0 is dynamic and %i1 is static

```
0   power_gen :
1         PUSHPENDING .power
2   .pending_loop :
3         ANYPENDING
4         be .exit
5         nop
6         POPPENDING %g1
7         jmp %g1
8         nop
9   .exit :
10        ret
11        nop
12  .power :
13        st %i0,[%fp+68]
14        ST %i1,[%fp+72]
15        mov 1,%o0
16        MOV (%o0),%o0
17        ST %o0,[%fp-20]
18        b .L1
19        nop
20  .L1 :
21        MAKELABEL .L1
22        LD [%fp+72],%o1
23        ADD %o1,-1,%o0
24        MOV %o0,%o1
25        ST %o1,[%fp+72]
26        CMP %o1,-1
27        BNE .L3
28        NOP
29        EXISTPENDING .L2
30        be .Ldummy_L3
31        nop
32        B .L2
33        NOP
34        PUSHPENDING .L3
35        b .pending_loop
36        nop
37  .Ldummy_L3 :
38        PUSHPENDING .L2,.L3
39        b .pending_loop
40        nop
41  .L2 :
42        b .L4
43        nop
44  .L3 :
45        MAKELABEL .L3
46        LD [%fp-20],%o0
47        ld [%fp+68],%o1
48        MUL %o0,(%o1),%o0
49        ST %o0,[%fp-20]
50        B .L1
51        NOP
52        PUSHPENDING .L1
53        b .pending_loop
54        nop
55  .L4 :
56        LD [%fp-20],%o0
57        MOV %o0,%i0
58        b .L5
59        nop
60  .L5 :
61        RET
62        b .pending_loop
63        nop
```

図 11 %i0 を静的，%i1 を動的とした場合の power 関数の生成拡張
Fig. 11 Generating extension of power function where %i0 is static and %i1 is dynamic

ド量が必要以上に増大してしまう.[1),13)]

(3) **無限特化処理の回避** 上記 (2) で述べた検査を行わなかった場合，動的なループ内では，より深刻な**無限特化処理** (*infinite specialization*)[13)] の問題を引き起こす．無限特化処理の問題とは，ループ内の特化処理を無限回繰り返し行ってしまうことである．この問題も，上記 (2) で述べた処理によって解決できる．

さて，図 10 と図 11 には，pending list と marked set を操作する五つの疑似命令が存在する．これらについて，説明する．

- **PUSHPENDING** $label_1[, label_2]$
  PUSHPENDING は，次の手順に従って処理を行う．

(1) プログラム中の全ての静的変数 $x_0, \cdots, x_n$ に対するその時点での部分状態 $s = (s_0, \cdots, s_n)$ を作成する．
(2) ラベル *label* と部分状態 s から，特化されたプログラムポイント $\langle label, s \rangle$ を作成する．
(3) pending list と marked set に，$\langle label, s \rangle$ と同じもの含まれているかどうかをを検査し，含まれていない場合は，pending list に $\langle label, s \rangle$ をプッシュする．どちらかに同じものが含まれている場合は，プッシュしない．オペランドにラベルが二つある場合は，$label_2$ を先にプッシュし，次に $label_1$ をプッシュする．

- **POPPENDING** *reg*
  POPPENDING は，次の手順に従って処理を行う．
  (1) pending list から，特化されたプログラムポイント $\langle label, s \rangle$ をポップする．
  (2) プログラム中で使用されていないレジスタのうち，予め指定されたレジスタ *reg*（図 10 と図 11 では %**g1**）に，ラベル *label* のアドレスをセットする．
  (3) 部分状態 s から値を取り出して，プログラム中の対応する静的変数に値を復元する．
  (4) marked set に $\langle label, s \rangle$ を追加する．
- **ANYPENDING**
  ANYPENDING は，pending list が空かどうかの判定を行い，その結果を **icc** フィールドに書き込む．具体的には，pending list が空の場合は，$reg = 0$ として **cmp** *reg*, 0 を実行した時と同じ動作を行い，空でない場合は，$reg \neq 0$ として **cmp** *reg*, 0 を実行した時と同じ動作を行う．
- **EXISTPENDING** *label*
  EXISTPENDING は，次の手順に従って処理を行う．
  (1) プログラム中の全ての静的変数 $x_0, \cdots, x_n$ に対するその時点での部分状態 $s = (s_0, \cdots, s_n)$ を作成する．
  (2) ラベル *label* と部分状態 s から，特化されたプログラムポイント $\langle label, s \rangle$ を作成する．
  (3) pending list と marked set に，$\langle label, s \rangle$ と同じものがあるかどうかの判定を行い，結果を **icc** フィールドに書き込む．具体的には，同じものがない場合は，$reg = 0$ として **cmp** *reg*, 0 を実行した時と同じ動作を行い，同じものがある場合は，$reg \neq 0$ として **cmp** *reg*, 0 を実行した時と同じ動作を行う．
- **MAKELABEL** *label*
  MAKELABEL は，次の手順に従って処理を行う．
  (1) その時点でのプログラム中の全ての静的変数の値を $s_0, \cdots, s_n$ とした時，ラベル $label\_s_0\_\ldots\_s_n$ を出力する．
  (2) 特化されたプログラムポイント $\langle label, s \rangle$（ただし，$s = (s_0, \cdots, s_n)$ とする）を作成する．
  (3) marked set に $\langle label, s \rangle$ を追加する．

図 10 と図 11 には，**ld**，**st**，**add** などの通常のアセンブリ命令の他に，**LD**，**ST**，**ADD** などの大文字の命令が存在する．これら大文字の命令は，自分自身と同じ命令を文字列として出力する疑似命令である．図 10, 16 行目の **ST** 命令や，図 11，46 行目の **MUL** 命令などのオペランドにある小括弧 () に囲まれた部分は，文字列の出力時に数値（即値）に置き換えられる部分である．例えば，疑似命令 **MUL** %**o0**, (%**o1**), %**o0** の実行時に，レジスタ %**o1** の値が 2 であるならば，文字列 "**mul** %**o0**, 2, %**o0**" を出力する．

疑似命令 **B** *label* は，プログラム中の全ての静的変数の値を $s_0, \cdots, s_n$ として，文字列 "**b** $label\_s_0\_\ldots\_s_n$" を出力する．**B***CC* *label* についても同様である．

図 10 と図 11 の生成拡張を実行して得られる残余プログラムを，それぞれ**図 12** と**図 13** に示す．図 12 は，%**i1** = 3 として，図 10 を実行して得られる残余プログラムである．図 13 は，%**i0** = 2 として，図 11 を実行して得られる残余プログラムである．

7.2 生成拡張の生成

本節では，生成拡張を生成する方法について説明する．生成拡張の生成は，以下の二段階の手順を実行するだけで完了する．

手順 1 図 10 と図 11 の 0 行目から 11 行目までの固定されたコードを，元のプログラムの先頭に追加する．

手順 2 束縛時解析の結果を基に，**図 14** と**図 15** の変換規則に従って，元のプログラムを命令毎に生成拡張の命令列に変換する．

図 14 と図 15 では，$\stackrel{ge}{\Longrightarrow}$ の左側の命令が，変換元のプログラムの命令であり，$\stackrel{ge}{\Longrightarrow}$ の右側の命令列が，変換先の生成拡張の命令列である．**st** reg_d, [*address*]，**b**，**bcc** や，*label* は，複数個の命令からなる命令列に変換される．**bcc** と **bcc** の変換先命令列にある *next_label* は，変換元の命令が含まれる基本ブロックの直後の基

```
0   power_res :
1         st %i0,[%fp+68]
2         mov 1,%o0
3         st %o0,[%fp-20]
4         ld [%fp-20],%o0
5         ld [%fp+68],%o1
6         mul %o0,%o1,%o0
7         st %o0,[%fp-20]
8         ld [%fp-20],%o0
9         ld [%fp+68],%o1
10        mul %o0,%o1,%o0
11        st %o0,[%fp-20]
12        ld [%fp-20],%o0
13        ld [%fp+68],%o1
14        mul %o0,%o1,%o0
15        st %o0,[%fp-20]
16        ld [%fp-20],%o0
17        mov %o0,%i0
18        ret
19        nop
```

図 12 %i0 を動的，%i1 を静的とした場合の power 関数の残余コード
Fig. 12 Residualized code of power function where %i0 is dynamic and %i1 is static

```
0   power_res :
1         st %i1,[%fp+72]
2         mov 1,%o0
3         st %o0,[%fp-20]
4   .L1_2 :
5         ld [%fp+72],%o1
6         add %o1,-1,%o0
7         mov %o0,%o1
8         st %o1,[%fp+72]
9         cmp %o1,-1
10        bne .L3_2
11        nop
12        ld [%fp-20],%o0
13        mov %o0,%i0
14        ret
15        nop
16  .L3_2 :
17        ld [%fp-20],%o0
18        mul %o0,2,%o0
19        st %o0,[%fp-20]
20        b .L1_2
21        nop
```

図 13 %i0 を静的，%i1 を動的とした場合の power 関数の残余コード
Fig. 13 Residualized code of power function where %i0 is static and %i1 is dynamic

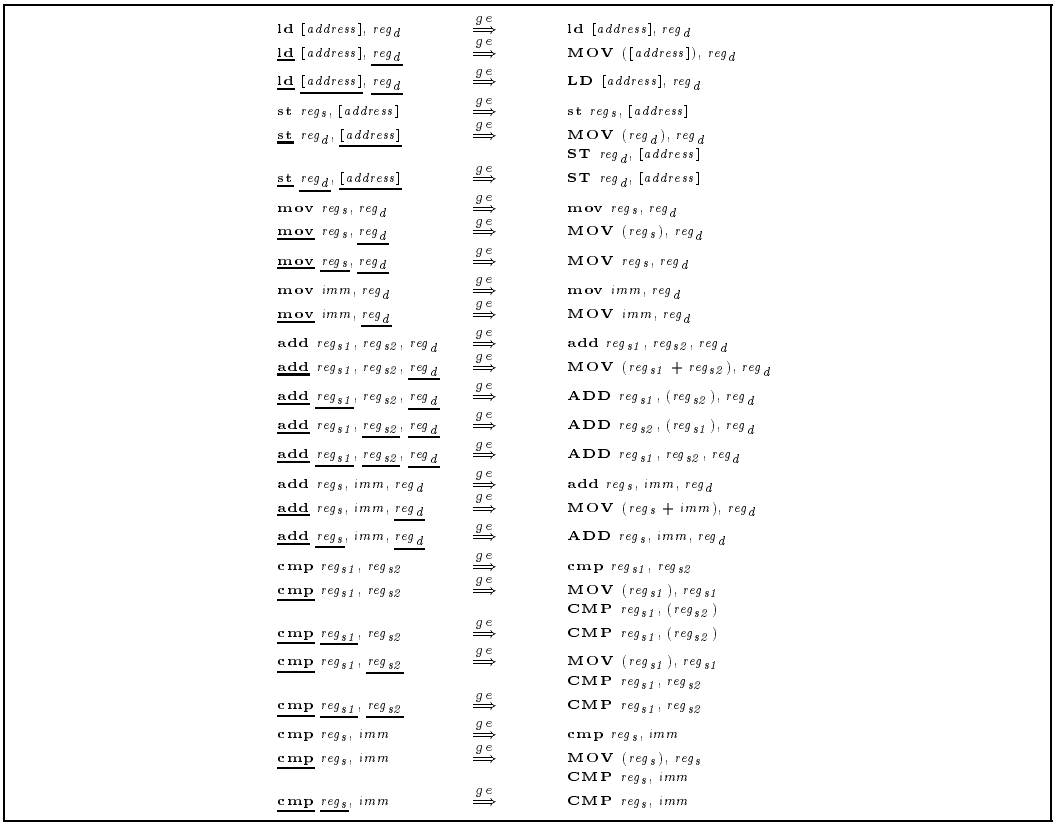


図 14 生成拡張への変換規則 (1)
Fig. 14 Generating extension transformation rule (1)

本ブロックのラベルを表す.

変換元の命令とラベルに付けられた下線は以下の意味を表す.

- 命令の下線が付けられたオペランドの変数は，動的な束縛時値を持つ.
- 一重の下線が付いた命令とラベルは，*Res* 行動値を，二重の下線が付いた命令は，*Reb* 行動値を持つ.
- 命令の下線の付いていないオペランドの変数は，静的な束縛時値を，下線の付いていない命令は，

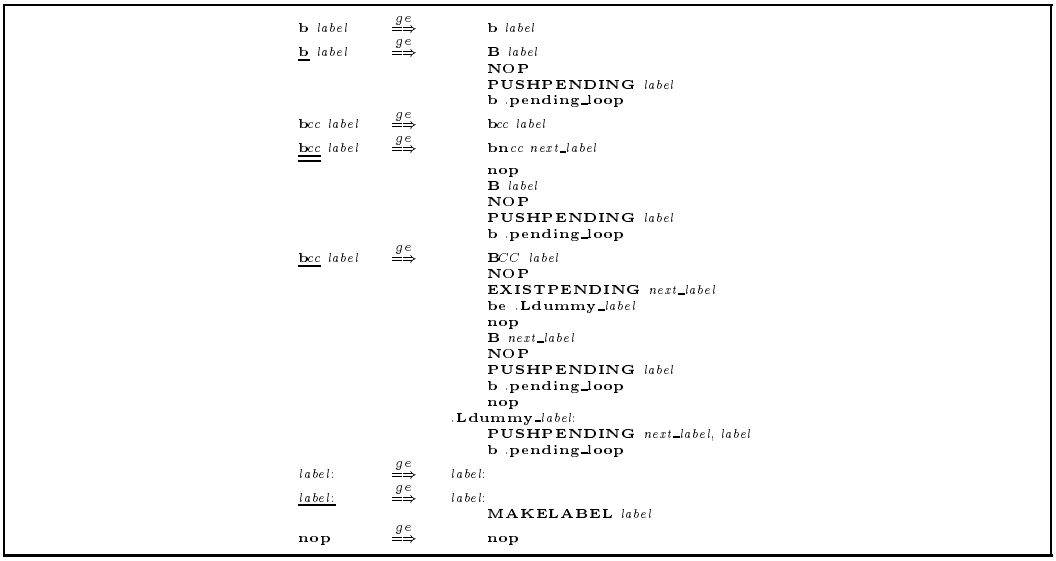


図 15 生成拡張への変換規則（2）
Fig. 15 Generating extension transformation rule (2)

Ev 行動値を持つ．

図 14 と図 15 の変換規則は，*Ev* 行動値を持つ命令とラベルは，そのまま生成拡張の命令とラベルとし，*Res* と *Reb* 行動値を持つ命令とラベルは，命令とラベルを文字列として出力する疑似命令を含む命令列に変換する．[1),14)]

b，bcc と，bcc の規則は，特に複雑な変換先命令列を持つので，これらについて説明する．

- b *label*
  b の変換先命令列は，自分自身の無条件分岐命令を文字列として出力し（**B** *label*）た後，**PUSHPENDING** 命令を呼び出して，静的変数の部分状態とラベル *label* を pending list に登録し，pending loop の先頭にプログラムの制御を戻す（**b .pending_loop**）．
- bcc *label*
  bcc は，条件付分岐命令自身は静的だが，動的な制御の範囲内にあることを表す．従って，bcc は特化時に分岐先を決定可能であり，元の条件付分岐命令を，実際に制御の移動を行うラベルに対する無条件分岐命令に置き換えることができる．
  bcc の変換先命令列は，元の分岐命令と逆の判定条件を持つ条件付分岐命令 **bn***cc next_label* の分岐が成立しない（元の条件付分岐命令 **b***cc label* の分岐が成立する）場合は，ラベル *label* への無条件分岐命令を出力し，その後，b と同様の処理を行う．**bn***cc next_label* の分岐が成立する（**b***cc label* の分岐が成立しない）場合は，単に直後の基本ブロックへ制御を移動するだけである．
- bcc *label*
  bcc は，自身が動的な条件付分岐命令であり，特化時に分岐先を決定できない．従って，bcc の変換先命令列は，分岐先ラベル *label* と直後の基本ブロックのラベル *next_label* について，pending list への登録を行わねばならばい．ただし，*next_label* に対する特化されたプログラムポイントが，既に特化対象となっている（**EXISTPENDING** *next_label* の判定が真）場合は，残余プログラム中の対応するラベル *next_label* への無条件分岐命令を出力し，**PUSHPENDING** は，*label* のみを引数に取る．特化対象となっていない場合は，*next_label* と *label* の両方を **PUSHPENDING** 命令の引数に取る．

8. 性能評価

本節では，本稿のアルゴリズムを用いて行った評価実験について報告する．実験は，CPU クロック周波数 143MHz の Sun UltraSPARC 上で行った．評価を行ったのは，power 関数と binary search 関数である．これらは，共に，C プログラムを gcc-2.8.1 -O3 -mv8 -S コマンドを用いて，SPARC アセンブリプログラムにコンパイルしたものである．測定は，C の “sys/times” ライブラリの times 関数を用いて行った．なお，特化したプログラムは，本稿のアルゴリズムに従って，gcc の出力を手で変換したものを用いた．

表 4 binary search 関数の実験結果
Table 4 Evaluation results of binary search function

静的な入力 (%i1)	%i0 = 0			%i0 = %i1 − 1		
	実行時間 (μ 秒)		向上比	実行時間 (μ 秒)		向上比
	特化せず	特化あり		特化せず	特化あり	
32	0.647	0.239	2.707	0.658	0.288	2.285
1024	1.055	0.411	2.567	1.068	0.453	2.358
1048576	3.420	1.959	1.746	3.669	2.519	1.457

表 3 power 関数の実験結果
Table 3 Evaluation results of power function

静的な入力 (%i1)	実行時間 (μ 秒)		向上比
	特化せず	特化あり	
10	0.626	0.478	1.310
20	1.565	1.285	1.218
30	2.865	2.444	1.172

power 関数は，x = %**i0** と n = %**i1** の二つの入力を持ち，x^n を返す関数である．ここでは，%**i0** を動的，%**i1** を静的とし，%**i0** の値を 2，%**i1** の値をそれぞれ 10，20 と，30 として計測した．実験結果を**表 3** に示す．

もう一つの実験に用いた binary search 関数は，C-Mix プロジェクト [17] によって配布されているものである☆．binray search 関数は，キー値 x = %**i0**，配列のサイズ size = %**i1** と，配列の先頭アドレス a = %**i2** の三つの入力を持ち，%**i0** の値を持つ要素が配列中にあれば，その要素の添字を返し，なければ，−1 を返す関数である．ここでは，%**i0** と %**i2** を動的，%**i1** を静的とした．配列は $\forall n \cdot a[n] = n$ を満たすように初期化し，%**i1** の値はそれぞれ 32，1024 と，1048576 (2^{20}) とする．%**i0** の値は，計算量が最善の場合 (0) と最悪の場合 (%**i1** − 1) の二通りについて実験を行った．実験結果を**表 4** に示す．binary search 関数では，特化によって，size を 2 の累乗に丸める計算が静的に実行されるので，1.4 倍以上の性能向上が得られた．

9. 関連研究

9.1 Use-sensitivity

Hornof らは，C 言語上で use sensitive な束縛時解析を定式化した [10),11)]．use sensitivity とは，変数の使用毎に束縛時値を計算することであり，アセンブリ言語上では， use sensitivity と program-point sensitivity とは同等である．

文献 10) では，二段階のデータフロー方程式によって，use sensitive な束縛時解析を実現している．一段階目の束縛時解析は，前向きのデータフロー方程式であり，本稿の使用の制約生成に相当する．二段階目の束縛時解析は，後ろ向きのデータフロー方程式であり，本稿の定義の制約生成に相当する．

Hornof らが，文献 10) で use sensitivity を導入した理由は，本稿で program-point sensitivity を導入した理由と本質的に異なる．本稿では，アセンブリプログラムが持つ複雑なデータフローを解決するために，program-point sensitivity を導入した．一方，文献 10) では，配列や構造体などの *lift* **操作** (*lift operation*) が出来ない要素を多く含んだ C 言語プログラムを，高精度な特化の対象とするために，use sensitivity を導入した．lift 操作とは，静的な変数や値を動的な文脈 (context) で用いるために，束縛時値を静的から動的に変更することである [12)]．

C 言語の配列や構造体に相当するアセンブリ言語のメモリアーキテクチャは，本稿の範囲外であるが，これらを含むように拡張した場合でも，本稿のアルゴリズムを使用できると考える．

9.2 C 言語上での特化

特化処理におけるプログラムの制御フローの扱いと，動的な制御の扱い，さらに，残余プログラムの最適化の三点から，既存の C 言語上での特化技術 [1),5),17)] と本稿での定式化との差異を述べる．

制御フロー 既存の C 言語上での特化手法は，非構造的な制御フローの扱いに関して，対照的な二つの手法を取っている．

Tempo[5)] は，構造化された C 言語プログラムのみを，特化対象として扱う．そのため，非構造的な制御フローを持つプログラムは，構造的な制御フローのみを持つプログラムに変換しなくてはならない [6)]．一方，C-Mix[17)] は，特化の効率を向上するため，C 言語プログラム中の全ての制御構造文を，無条件の goto 文と条件付きの goto 文のみからなる制御構造に展開してから特化処理を行う [13)]．

本稿の手法は，非構造的な制御フローを直接扱う点において，C-Mix の手法に近い．本稿では，5 節の制御フロー解析によって，特化処理で非構造的な制御フローを直接扱うことが可能であることを示した．

☆ ここで用いた binary search 関数は，プログラム特化に適した特別なアルゴリズムを用いたものであり，通常のアルゴリズムを用いたものではない．本プログラムは C-Mix 配布パッケージの “examples/binsearch/” ディレクトリ以下にある．

動的な制御 既存の C 言語上での特化手法は，動的な if 文の扱いに関しても，対照的な二つの手法を取っている．

C-Mix は，残余コード生成時に，プログラム中で動的な if 文に続く部分を，if 文の then 節と else 節内に複製したコードを出力する [1),13)]．すなわち，“if e { S_1 } else { S_2 }; S_3;” を，動的な if 文を含むコード片とすると，“if e { S_1; S_3 } else { S_2; S_3 };” の形で，残余コードが出力される．この手法では，プログラム中の特化時に計算可能な部分が増えるので，残余プログラムの実行効率が向上することが多い．しかし，binary search 関数のようなプログラムの場合は，静的な while ループ中の動的な if 文を全て展開した結果，残余プログラムのコード量が爆発的に増大してしまうと言う問題が起こる [17)]．Tempo と本稿の手法では，if 文に続く部分を，if 文の then 節と else 節内に複製しない．これは，Tempo と本稿の手法では，精度の高い束縛時解析を行っており，C-Mix の手法を行わなくても，残余プログラムの実行効率が問題にならないからである．また，Tempo と本稿の手法では，残余プログラムのコード量が爆発的に増大することは少ない．しかし，精度の高い束縛時解析を行うためには，様々な解析が必要であり，そのため，特化処理のコストが増大してしまうと言う欠点がある．

残余プログラムの最適化 C-Mix と Tempo は，高水準言語上でのプログラム特化器なので，生成した残余プログラムの効率に関わらず，コンパイラのバックエンド部での最適化によって，効率的に実行できることが多い．一方，アセンブリ言語上でプログラム特化を行う場合は，コンパイラでの最適化が期待できないので，特化時に効率の良い残余プログラムを生成するか，特化後に改めて残余プログラムを最適化する必要がある．

9.3 Java 仮想マシン言語上での特化

Masuhara らは，四則演算命令，オペランドスタックとローカル変数を操作する命令と，メソッド呼び出し命令からなる Java 仮想マシン（JavaVM）言語のサブセット上での実行時特化手法を定式化した [14)]．

JavaVM 言語は，本稿のアセンブリ言語と同様に低水準言語である．本稿のアセンブリ言語と JavaVM 言語との最大の違いは，本稿のアセンブリ言語がレジスタ型機械語であるのに対して，JavaVM 言語はスタック型機械語である点である．

文献 14) では，本稿と同様に，JavaVM 言語プログラムが持つデータフローの複雑さと，非構造的な制御フローの扱いについて言及している．また，文献 14) では，JavaVM 言語上での flow sensitive な束縛時解析を，型システムに基づいた制約解消によって実現している．

10. ま と め

本稿では，ソースコードが存在しないソフトウェアをバイナリレベルで特化する準備として，RISC 型アセンブリ言語上でのプログラム特化技法を定式化した．アセンブリ言語上でのプログラム特化には，二つの大きな課題があった．一つ目は，データフローの複雑さの問題であり，二つ目は，非構造的な制御フローの扱いの問題である．

本稿では，前者に対しては，program-point sensitive な束縛時解析を導入することによって問題を解決した．program-point sensitive な束縛時解析は，変数の使用による出現に対する束縛時値と定義による出現に対する束縛時値とを，別々に扱うことで実現できる．本稿では，制約解消を用いることによって，既存のデータフロー方程式を用いた解析よりも簡潔な表現で定式化した．

また，後者に対しては，高度な制御フロー解析を導入することによって，非構造的な制御フローを直接扱える手法を実現した．非構造的な制御フローを直接扱うことによって，非構造的な制御を避けてプログラム変換を行う既存の C 言語上でのプログラム特化器に比べて，より効率的な特化処理が可能となった．

今後の課題としては，本稿で扱ったアセンブリ言語に対する制限を排除し，より複雑なアセンブリ言語のアーキテクチャを特化対象として扱うために，次のような点で，本稿のアルゴリズムを拡張することが挙げられる．

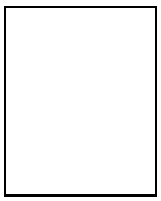
(1) メモリアクセスに対する制限を取り除くため，エイリアス解析を導入すること．
(2) アルゴリズムを手続き間解析に拡張し，手続き呼び出しを加えること．
(3) アセンブリプログラムに対して，関数インライニングなどの特化時最適化を導入すること．
(4) 残余プログラムに対する最適化処理を導入すること．

参 考 文 献

1) Andersen, L. O.: *Program analysis and specialization for C programming language*, PhD Thesis, Dept. of Computer Science, University of Copenhagen, Denmark (1994).

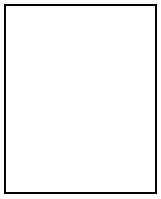
2) Appel, A. W.: *Modern compiler implementation in C*, Cambridge University Press (1998).
3) Bondorf, A. and Danvy, O.: Automatic autoprojection of recursive equations with global variables and abstract data types, *Science of Computer Programming*, vol. 16, pp. 151–195 (1991).
4) Birkedal, L. and Welinder, M.: Binding time analysis for Standard ML, *Lisp and Symbolic Computation*, Vol. 8, No. 3, pp. 191–208 (1995).
5) Consel, C. et al.: Tempo Specializer☆1, *Compose project*, IRISA, France (1998).
6) Consel, C., Hornof, L., Noël, F., Noyé, J. and Volanschi, N.: A uniform approach for compile-time and run-time specialization, *Partial Evaluation. Proceedings, volume 1110 of Lecture Notes in Computer Science* (Olivier Danvy, R. G. and Thiemann, P.(eds.)), Springer-Verlag, pp. 54–72 (1996).
7) Consel, C. and Noël, F.: A general approach for run-time specialization and its application to C, *Proceedings of the 23rd Symposium on Principles of Programming Languages (POPL' 96)*, ACM SIGPLAN-SIGACT, pp. 145–156 (1996).
8) Yarvin, C. and Sah, A.: *QuaC: binary optimization for fast runtime code generation in C*, Technical report UCB//CSD-94-792, Dept. of Computer Science, University of California Berkeley (1994).
9) Henglein, F.: Efficient type inference for higher-order binding-time analysis, *Functional Programming Languages and Computer Architecture, volume 523 of Lecture Notes in Computer Science*, Springer-Verlag, pp. 448–472 (1991).
10) Hornof, L., Consel, C. and Noyé, J.: Effective specialization of realistic programs via use sensitivity, *Proceedings of the the Fourth International Symposium on Static Analysis (SAS'97), volume 1302 of Lecture Notes in Computer Science*, Springer-Verlag, pp. 293–314 (1997).
11) Hornof, L. and Noyé, J.: Accurate binding time analysis for imperative language: flow, context and return sensitivity, *Theoretical Compter Science*, Vol. 248, pp. 3–27 (2000).
12) Jones, N. D., Gomard, C. K. and Sestoft, P.: *Partial evaluation and automatic program generation*☆2 , Out of print, Prentice Hall (1993).
13) Makholm, H.: Specializing C an introduction to principles behind C-Mix, Technical report, Dept. of Computer Science, University of Copenhagen, Denmark (1999).
14) Masuhara, H. and Yonezawa, A.: Run-time bytecode specialization: A portable approach to generating optimized specialized code, *Second Symposium on Programs as Data Object, volume 2053 of Lecture Notes in Computer Science*, Springer-Verlag, pp. 138–154 (2001).
15) Muchnick, S. S.: *Advanced compiler design and implementation*, Morgan Kaufmann Publishers (1997).
16) Nielson, F., Nielson, H. R. and Hankin, C.: *Principles of program analysis*, Springer-Verlag (1999).
17) Secher, J. P., Makholm, H. and Glenstrup, A. et al.: C-Mix☆3, *TOPPS group*, Dept. of Computer Science, University of Copenhagen, Denmark (1998).
18) 徳生吉孝: アセンブリ言語上でのプログラム特化☆4, 修士論文, 東京工業大学 情報理工学研究科数理・計算科学専攻 (2000).
19) Weaver, D.L., Germond, T. and SPARC International, Inc.: *The SPARC Architecture Manual Version 9*, Prentice Hall (1994).

(平成 12 年 8 月 28 日受付)
(平成 13 年 3 月 20 日採録)



徳生 吉孝

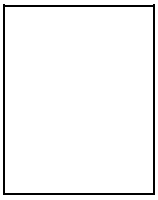
1976 年生. 1998 年千葉大学理学部数学・情報数理学科卒業. 2000 年東京工業大学大学院数理・計算科学専攻修士課程修了. 在学中は, プログラミング言語, 部分評価に興味を持つ. 同年 (株) 日立製作所入社. 現在, DBMS (データベース管理システム) の研究開発に従事.



脇田 建 (正会員)

1965 年生. 1989 年東京大学理学部情報科学科卒業. 1991 年同大学大学院理学系研究科情報科学専攻修了. 東京工業大学大学院情報理工学研究科数理・計算科学専攻講師. 博士 (理学). プログラミング言語, 分散処理等に興味を持つ. 日本ソフトウェア科学会, ACM, IEEE 各会員.

☆1 http://compose.labri.fr/prototypes/tempo/
☆2 Full text of this book is available at http://www.dina.kvl.dk/~sestoft/pebook/pebook.html in 1999.
☆3 http://www.diku.dk/research-groups/topps/activities/cmix/
☆4 This article is available at http://www.is.titech.ac.jp/~sassa/lab/ob/toku/publications.html.



佐々　政孝（正会員）

1948 年生．1970 年東京大学理学部物理学科卒業．1974 年同大学院博士課程中退，東京工業大学理学部情報科学科助手．1981 年筑波大学電子・情報工学系．1992 年東京工業大学理学部．現在同大学情報理工学研究科数理・計算科学専攻教授．理学博士．プログラミング言語，コンパイラ，プログラミング環境，属性文法に興味を持つ．著書「プログラミング言語処理系」（岩波書店）．日本ソフトウェア科学会，ACM，IEEE 各会員．