

# 双方向CTLによるJava最適化器の生成

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

方 玲

(04M54028)

平成18年度修士論文

指導教官 佐々 政孝 教授

平成18年8月25日

# 目次

第 1 章	序論	7
1.1	本研究の背景	7
1.2	本論文の貢献	7
1.3	本論文の構成	8
第 2 章	プログラムの最適化	9
2.1	プログラムの最適化	9
2.1.1	無用命令除去	9
2.1.2	コピー伝播 (定数伝播も含む)	9
2.1.3	共通部分式除去	10
2.1.4	ループ不変式の移動	10
2.1.5	部分冗長性除去 (PRE)	10
第 3 章	時相論理	12
3.1	Kripke 構造	12
3.2	CTL 分岐時相論理	13
3.2.1	CTL の構文規則	13
3.2.2	CTL の意味論	13
3.2.3	CTL の演繹体系	14
3.2.4	CTL の例題	14
3.3	双方向 CTL	14
3.3.1	双方向 CTL とは	15
3.3.2	構文規則	15
3.3.3	意味論	16
3.3.4	双方向 CTL の演繹体系	16
第 4 章	時相論理による最適化の記述	18
4.1	制御フローモデル	18
4.1.1	プログラムの構文	18
4.1.2	制御フローモデル	18
4.2	CTL-FV	20
4.2.1	自由変数の導入	20
4.2.2	自由変数の束縛	20
4.2.3	自由変数束縛の計算量	20
4.3	最適化の記述	21

<b>第 5 章</b>	<b>双方向 CTL モデル検査器</b>	<b>22</b>
5.1	双方向 CTL モデル検査器のアルゴリズム	22
5.1.1	CTL 式の解析	23
5.1.2	モデル検査	23
5.1.3	モデル検査の擬似コード	29
5.2	モデル検査の計算量	31
<b>第 6 章</b>	<b>本研究の最適化の記述</b>	<b>33</b>
6.1	最適化の記述	33
6.1.1	最適化記述の文法	33
6.2	最適化記述の文法の説明	35
6.2.1	MATCH 部	35
6.2.2	CONDITION 部	35
6.2.3	PROCESS 部	36
6.3	本記述の利点	36
6.3.1	複雑な定式化が容易に	36
6.3.2	自由変数の減少	36
6.4	本研究で実装した定式化	37
6.4.1	無用命令除去	38
6.4.2	コピー伝播 (定数伝播を含む)	38
6.4.3	部分冗長性除去 (共通部分式除去, ループ不変式の移動)	39
6.4.4	CTL による定式化のノーハウ	43
<b>第 7 章</b>	<b>双方向 CTL による最適化器</b>	<b>45</b>
7.1	前処理部	45
7.2	モデル検査	45
7.3	書換え部	45
7.4	最適化器の計算量	45
<b>第 8 章</b>	<b>実装</b>	<b>47</b>
8.1	Soot と Jimple	47
8.2	プログラム変換	49
8.3	システムの構造	49
8.4	処理の擬似コード	50
8.5	処理の詳細	51
8.5.1	入力	51
8.5.2	ルールファイル	51
8.5.3	危険辺除去	51
8.5.4	プログラム情報を収集する	52
8.5.5	自由変数の束縛	52
8.5.6	モデル検査	52
8.5.7	辺の計算	53

8.5.8	書換え部 . . . . .	53
8.5.9	その他実用上必要ないくつかの処理 . . . . .	55
8.5.10	最適化処理の例 . . . . .	56
<b>第 9 章</b>	<b>実験と考察</b>	<b>58</b>
9.1	実験環境 . . . . .	58
9.2	ベンチマークの説明 . . . . .	58
9.3	実験の結果 . . . . .	59
9.3.1	本研究の手法による最適化の処理時間 . . . . .	59
9.3.2	最適化前と最適化後の実行時間の比較 . . . . .	60
9.3.3	Lacey らの手法との比較 . . . . .	61
9.3.4	番らの手法との比較 . . . . .	61
9.3.5	同じ最適化を異なる CTL 式で記述したときの最適化時間の比較 . . . . .	61
9.4	実験データの分析 . . . . .	62
9.4.1	分析と考察 . . . . .	62
9.4.2	今後の課題 . . . . .	64
<b>第 10 章</b>	<b>関連研究</b>	<b>66</b>
10.1	Lacey らの研究 . . . . .	66
10.2	山岡らの研究 . . . . .	67
10.3	番らの研究 . . . . .	67
10.4	伊藤らの研究 . . . . .	67
<b>第 11 章</b>	<b>今後の課題</b>	<b>68</b>
11.1	最適化時間の短縮 . . . . .	68
11.2	最適化の効果の向上 . . . . .	69
<b>第 12 章</b>	<b>まとめ</b>	<b>70</b>

# 目次

2.1	無用命令除去	9
2.2	コピー伝播	9
2.3	共通部分式除去	10
2.4	ループ不変式の移動	10
2.5	部分冗長性除去	11
3.1	(有限の)Kripke 構造とその構造を展開した CTL 無限木	15
3.2	Kripke 構造とその構造を展開した双方向 CTL 無限木	17
4.1	コードと制御フローモデルの例 (L(n) は略した)	19
5.1	プログラムと CTL 式 (無用命令除去) の例	22
5.2	CTL 構文木	23
5.3	モデル検査の例	24
5.4	AU の例 (普通の場合)	26
5.5	AU の例 (loop の場合)	27
5.6	AU の例 (互いに行き来する loop の場合)	28
6.1	双方向 CTL 木による CTL 式の説明-無要命令除去	39
6.2	双方向 CTL 木による CTL 式の説明-コピー伝播	40
6.3	A Simple Algorithm for Partial Redundancy Elimination の例	41
6.4	同じ最適化を行う異なった CTL 式	43
8.1	システム略図	49
8.2	システム詳細図	49
8.3	コピー伝播	54
8.4	無用命令除去	55
8.5	モデル検査の結果の例	56
8.6	書換えの例	57
8.7	図 8.5 のプログラム書換え後の例	57
9.1	SPECjvm98 ベンチマークの最適化効果 (目的コードの実行時間, 最適化なしを 1 に正規化)	60
9.2	奥村らの Java コードの最適化効果 (目的コードの実行時間, 最適化なしを 1 に正規化)	61

9.3	番らの手法と本研究の最適化時間の比較 (コピー伝播の例、番らの方法を 1 に正規化) . . . . .	62
9.4	同じ最適化を異なる CTL 式で記述したときの最適化時間の比較 (縦軸：秒) . . . . .	63
9.5	例外による最適化の阻害 . . . . .	64
10.1	Lacey の式が扱えない例 . . . . .	66

# 表 目 次

5.1	CTL 式とその部分式の例 . . . . .	23
9.1	SPECjvm98 ベンチマークの説明 . . . . .	58
9.2	奥村らの Java コードの説明 . . . . .	59
9.3	SPECjvm98 ベンチマークの本研究の手法による最適化時間 (単位 : 秒) . . . . .	59
9.4	奥村らのコードの本研究の手法による最適化時間 (単位 : 秒) . . . . .	60
9.5	自由変数による解析時間の差の例 . . . . .	62

# 第1章 序論

## 1.1 本研究の背景

コンパイラの最適化器はプログラムを書きかきだして作成することがほとんどだが、近年 CTL という論理による最適化の研究も行われている。CTL による最適化は、次のような条件付き書換え規則を用いることにより、多くの古典的なコンパイラの最適化を簡潔に表現することができる。条件付き書換え規則は次のようである。

$$I \Rightarrow I' \text{ if}$$

ここで、 $I$  と  $I'$  は命令、 $\phi$  は時相論理式を表している。この規則は「プログラムのある点  $p$  に命令文  $I$  が存在するとき、条件  $\phi$  が成り立つならば、命令  $I$  を  $I'$  に置換える」と解釈する。 $\phi$  は CTL 論理式である。

この手法のメリットは、

- 論理式で一行か、多くても十数行で最適化を記述できる
- 証明できる

ということである。

CTL による最適化器の性能は、記述能力、最適化時間、最適化効果の三つの基準で評価できる。

従来の研究として、Lacey らの研究は、過去時制を含む CTL-FV を提唱し、証明した。しかし、証明した式は実際の最適化の一部しか扱えない例が多い。記述しきれないものは通常のプログラムを呼び出すことになる。実装についてもあまり触れておらず、最適化時間などのデータもない。

山岡らの研究は、既存のモデル検査器 SMV を使い、Lacey らの理論の一部を実装した。過去時制を用いることができず、無用命令除去しか扱えない。

番らの研究は過去時制を除去することによってコピー伝播を行えるが、除去処理の時間と、除去によって式が長くなるため、最適化時間がかなり長い。

## 1.2 本論文の貢献

本研究はこの分野における課題である「CTL 等の論理による手法で通常のアロリズムと同じように最適化できるか」という問題の解決に向けて研究を行った。

本研究の貢献は次のとおりである。

- 実用性の高い、CTL による Java 最適化器を実装した。

- 実験の結果，従来研究と比べて最適化時間が格段に短く，最適化効果も十分見込めることがわかった．
- 実現に当たり種々試行錯誤を行った結果，問題点が明らかになり，今後に役立つ多くの知見が得られた．

従来の研究と比べて，本研究の独創性は次のとおりである．

- モデル検査器は従来研究と異なり，何も変換せずに直接処理できる形で実装し，効率を大幅に向上した．
- 最適化記述について，従来研究は条件式が特定の番号の命令文しか対象とできないが，本研究の条件式は命令文の集合を対象とすることができる．
- 通常の最適化器に近い性能を持った，CTL による最適化器の実装は本研究がはじめてである．

今後は，問題点の解決，性能の改善をし，CTL 式に基づく実用的な Java 最適化器を伝統的な最適化器の一部として組み込めことを見込めるようになった．

### 1.3 本論文の構成

本論文では，以下の順に述べていく．第 2 章ではプログラム最適化について，第 3 章では本研究に関係する時相論理について説明する．第 4 章では時相論理による最適化の記述を説明する．そして，本研究の解説に入り，第 5 章では，本研究で実装したモデル検査器を説明する．第 6 章は本研究の最適化の記述を説明する．第 7 章は本研究の最適器を説明する．第 8 章は実装の内容を述べる．第 9 章は実験と考察について，第 10, 第 11 と第 12 章はそれぞれ関連研究，今後の課題とまとめを述べる．

## 第2章 プログラムの最適化

最適化はコンパイラの中で重要な存在である，いろいろな最適化の手法を用いて，実行速度を向上したり，目的コードを小さくするのが目的である [13][8]．最適化の手法はいろいろあるが，今回本研究で定式化したものを例として説明する．

### 2.1 プログラムの最適化

#### 2.1.1 無用命令除去

定義した後使われないものや，そこへ制御が移る事のないコードは，無用命令として除去できる．

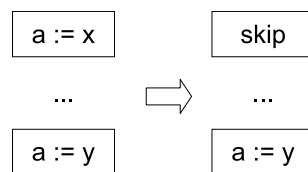


図 2.1: 無用命令除去

図 2.1 の左側の  $a := x$  は再定義  $a := y$  まで使われないので， $a := x$  は無用である．図 2.1 の右側はその無用命令を `skip` で書換え，無駄な計算をなくす．

#### 2.1.2 コピー伝播（定数伝播も含む）

$x := y$  というコピー文は  $x$  は  $y$  と同じ値を持つため，後に現れる  $x$  の使用は  $y$  の使用に置き換えられる．この文は後で無用命令除去できる可能性が高い．伝播先の命令文が  $y$  と定数の演算の場合， $y$  も定数だと，定数の畳み込みもできる．

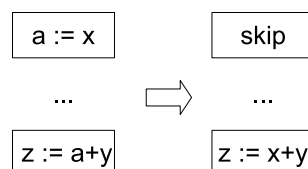


図 2.2: コピー伝播

### 2.1.3 共通部分式除去

同じ値を持つ部分式が2か所以上に現れた場合，それらを共通部分式という．計算した結果を利用する事によって，演算の回数を減らす事ができる．

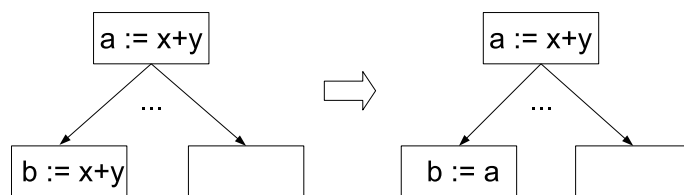


図 2.3: 共通部分式除去

### 2.1.4 ループ不変式の移動

ループの繰り返しによらず一定な結果を与える計算をループの外に追い出す事によって，計算の回数が減る．

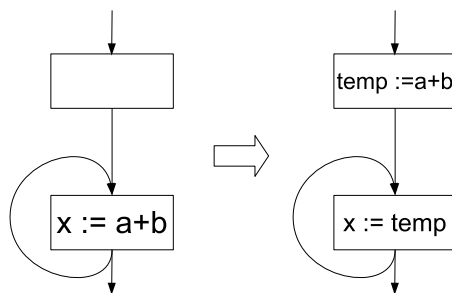


図 2.4: ループ不変式の移動

### 2.1.5 部分冗長性除去 (PRE)

部分冗長性除去 PRE (Partial Redundancy Elimination) は，共通部分式除去やループ不変コードのループ外移動を統合的に行う最適化手法で，非常に強力である [13][8]．図 2.5(左) の場合，左上のブロックからの経路上では「 $a + b$ 」が2回実行される．しかし，下側の式を除去してしまうと，右上からの経路上に「 $a + b$ 」の式がなくなってしまうので，下側の「 $a + b$ 」は冗長ではない．これは部分冗長 (partially redundant) であるという．この場合には，図 2.5(右) のように，右上のブロックに「 $a + b$ 」を挿入することによって，下側の「 $a + b$ 」は冗長になり，除去可能になる．

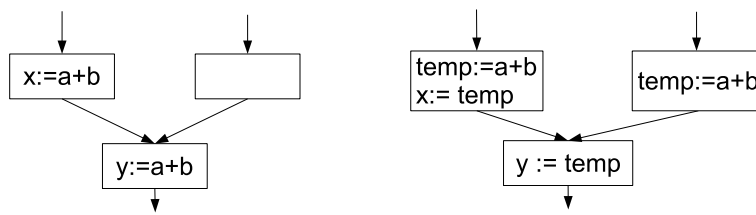


图 2.5: 部分冗長性除去

## 第3章 時相論理

時相論理 (temporal logic)[7] は特殊な様相論理であり, それは命題の真理値が時間とともにどのように変わるかを定性的に記述し論証するための形式的な体系を与える。「時刻0の世界」, 「時刻1の世界」... 「時刻nの世界」といったように, 時刻に依存した複数の世界が存在し, 論理式の真偽値が各世界によって異なるという論理体系である。時相論理は「時間」の概念の捉え方に応じて, いくつかに分類される。時相論理はさらに別の観点からも分類される。命題論理か一階の論理か。大域的か合成的か。分岐的か線形か。時点か時区間か。そして過去時制か未来時制か。その結果,  $LTTL, CTL, CTL^*, PCTL, NCTL...$  などの何種類もの時相論理がある。以下は本研究に関係がある理論について述べる。

以下は本研究と関係する時相論理及び関連理論を述べる。

### 3.1 Kripke 構造

定義 3.1 (Kripke 構造)

検査する対象のモデルは, 原子述語の集合  $Prop$  上で定義される Kripke 構造  $M = (S, s_{start}, R, L)$  であり,

- $S$  はモデルの空でない状態の集合
- $s_{start} \in S$  はモデルの始状態の集合
- $R \subset S \times S$  は  $S$  上の遷移関係
- $L$  は  $S \rightarrow 2^{Prop}$  は  $S$  上で成り立つ原子述語の集合

定義 3.2 (Kripke 構造上の経路)

Kripke 構造  $M$  上の経路 (path) とは状態の列  $s_0 s_1 \dots, (\forall i; s_i R s_{i+1})$  である。経路は無有限列あるいは有限列である。

定義 3.3 (経路のサフィックス)

経路  $\pi = s_0 s_1 \dots$  に対し,  $\pi^i$  は経路の状態列のうち  $s_i$  から始まる部分, すなわち  $s_i s_{i+1} \dots$  を表す。

定義 3.4 (論理式が成り立つ)

$M = (S, s_{start}, R, L)$  をモデルとしたとき, 状態  $s_{start} \in S$  で CTL 論理式  $\phi$  が成り立つことを次のように表す。

$$M = (S, s_{start}, R, L) \models \phi$$

## 3.2 CTL 分岐時相論理

CTL (computational tree logic) は分岐時相論理 (branching-time temporal logic) の一つである。経路限量子の直後に時相演算子が現れる形式のみが許される [7]。かつ経路限量子と時相演算子はペアになっていないといけない。経路限量子の直後に時相演算子が現れるとき、そのペアは本論文では結合子と呼ぶ。分岐的な時相論理においては、時間の構造は各時点がその直後の時点を複数個もつという、分岐した木構造を持つ。時間に沿った一つのパスにおいて線形時間構造の性質を持つ。

下記の説明で は時相論理式とする。

- 経路限量子

- $A$  : これからすべての経路で が満たされれば真
- $E$  : これからのある経路で が満たされれば真

- 時相演算子

- $_1U _2$  : 経路上  $_2$  が成り立つまで  $_1$  が満たされ続けるならば真
- $X$  : 次の状態で が満たされれば真
- $G$  : 経路上 が満たし続けると真
- $F$  : 経路上 が何処かで満たされれば真

経路限量子の綴りの意味は、 $A=(All)$ 、 $E=(Exist)$  である。

時相演算子の綴りの意味は、 $U=(Until)$ 、 $X=(neXt)$ 、 $G=(Globally)$ 、 $F=(Future)$  である。

CTL はペアとして見たとき  $AU( _1, _2)$ 、 $EX( )$ ... などの形をしている式しか書けない。

つまり、経路上で、ある状態から ( ) が満たされる、偶数の状態で ( ) が満たされる、ある時からある時まで ( ) が満たされる... というのが表現できない。

### 3.2.1 CTL の構文規則

CTL の構文規則は以下の通りである。

$$CTL \ni \quad ::= \quad | \neg \quad | \quad _1 \quad _2 \\ | EX \quad | E \quad _1U \quad _2$$

### 3.2.2 CTL の意味論

$$s \models true \text{ iff } \quad \in L(s)$$

$$s \models \quad \text{ iff } \quad \in L(s)$$

$$s \models \neg \quad \text{ iff } s \models \quad \text{ ではない}$$

$s \models \phi_1 \wedge \phi_2$  iff  $s \models \phi_1$  かつ  $s \models \phi_2$

$s \models EX \phi$  iff  $\exists s' ; sRs'$  かつ  $s' \models \phi$

$s \models E \phi_1 U \phi_2$  iff ある  $s$  から始まる経路  $s_0 s_1 \dots (s_0 = s)$  が存在して,  $\exists i > 0;$   
 $s_i \models \phi_2$  かつ  $0 \leq \forall j < i; s_j \models \phi_1$

### 3.2.3 CTLの演繹体系

構文定義中に現れないがよく用いられる結合子が下記の通りである．それらの結合子は，構文定義で定義した結合子しか用いずに，同じセマンティクスのものに変形できる．

$$\phi_1 \vee \phi_2 \equiv \neg (\neg \phi_1 \wedge \neg \phi_2)$$

$$EF \phi \equiv E \text{ true } U \phi$$

$$AF \phi \equiv A \text{ true } U \phi$$

$$EG \phi \equiv \neg (AF \neg \phi) \equiv \neg (A \text{ true } U (\neg \phi))$$

$$AG \phi \equiv \neg (EF \neg \phi) \equiv \neg (E \text{ true } U (\neg \phi))$$

$$AX \phi \equiv \neg (EX \neg \phi)$$

$$A \phi_1 U \phi_2 \equiv \neg (E \text{ true } U \neg \phi_2) \wedge \neg (E \neg \phi_2 U \neg \phi_1)$$

### 3.2.4 CTLの例題

図 3.1 は (有限の)Kripke 構造 (左) をほどいてつくった無限木 (右) である． $s_0$  は始状態．Kripke 構造  $M$  の各要素は

- $S = \{s_0, s_1, s_2\}$
- $s_{start} = s_0$
- $R = \{(s_0, s_1), (s_0, s_2), (s_2, s_1), (s_1, s_0)\}$
- $L = \{a, b, c\}$

## 3.3 双方向 CTL

双方向 CTL は時相論理の 1 つであり，提案は [9] に始まると思われる．以下，双方向 CTL について説明する．

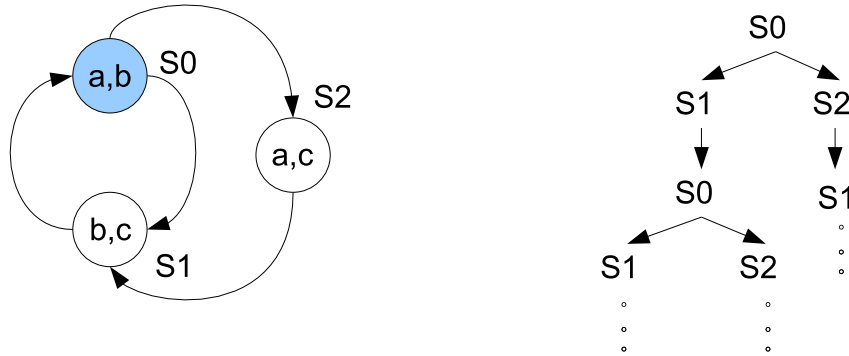


図 3.1: (有限の)Kripke 構造とその構造を展開した CTL 無限木

### 3.3.1 双方向 CTL とは

双方向 CTL の提案は [9] に始まると思われるが、ここでは [3] の CTL-FV を論理として採用した。自由変数を除けば、これは分岐的な時相論理で、時間の構造は各時点が分岐した木構造を対称的に二つ持ち、それぞれ直後と直前の時点を複数個もつものである。

双方向 CTL では CTL の未来時相演算子と対称的な過去時相演算子を持つ。

CTL と異なり経路限量子は  $A, E$  のほか  $\overleftarrow{A}, \overleftarrow{E}$  を持ち、それぞれ  $A, E$  に対して双対的な意味を与えたものである。

過去時相は将来時相とまったく同等、対称的な存在である。

過去時制を制限無しに記述、検証ができるため、簡潔性と表現力及び効率が優れている。

本論文には記述や実装にあたって、便利のため、双方向 CTL を CTL に略称する場合がある。たとえば、双方向 CTL 式を CTL 式、双方向 CTL 構文木を CTL 構文木... のように略す。

### 3.3.2 構文規則

双方向 CTL の構文規則は以下の通りである。

$$\begin{aligned} \text{双方向 CTL} \ni & ::= \mid \neg \mid \_1 \_2 \\ & \mid EX \mid E \_1 U \_2 \\ & \mid \overleftarrow{E} X \mid \overleftarrow{E} \_1 U \_2 \end{aligned}$$

他の結合子  $EF, EG, AF, \overleftarrow{A}X, \dots$  は変換によって上記の双方向 CTL の構文規則の結合子だけを用いた式に変換できる。

### 3.3.3 意味論

双方向 CTL の意味論は Kripke 構造によって与えられる。Kripke 構造  $K$  は三つ組  $(S, R, L)$  であり、 $S$  は状態の集合、 $R \subseteq S \times S$  は遷移関係、 $L: S \rightarrow 2^{Prop}$  は各状態にその状態において真となる述語の集合を割り当てる関数である。

$K$  における  $s_0$  からのパスとは、 $\forall i \geq 0: (s_i, s_{i+1}) \in R$  となるような状態の (有限もしくは無限の) 極大な列  $\dots = (s_0, s_1, \dots)$  である。ここで、有限のパス  $(s_0, s_1, \dots, s_m)$  が極大であるとは、 $\forall s: (s_m, s) \notin R$ 、つまり  $s_m$  が successor を持たないことである。 $s_0$  からの逆向きのパスとは、 $\forall i \geq 0: (s_{i+1}, s_i) \in R$  となるような極大な列である。

論理式  $\phi$  が Kripke 構造  $K$  の状態  $s$  で真であるという関係を  $K, s \models \phi$  で表す。また、 $K$  が明らかな場合には  $K$  を省略する。関係  $\models$  は以下のように定義される。

$$s \models \phi \quad \text{iff} \quad \phi \in L(s)$$

$$s \models \neg \phi \quad \text{iff} \quad s \not\models \phi \quad \text{ではない}$$

$$s \models \phi_1 \wedge \phi_2 \quad \text{iff} \quad s \models \phi_1 \quad \text{かつ} \quad s \models \phi_2$$

$$s \models EX \phi \quad \text{iff} \quad \exists s'; sRs' \quad \text{かつ} \quad s' \models \phi$$

$$s \models \overleftarrow{E}X \phi \quad \text{iff} \quad \exists s'; s'R s \quad \text{かつ} \quad s' \models \phi$$

$$s \models E \phi_1 U \phi_2 \quad \text{iff} \quad \text{ある } s \text{ から始まる経路 } s_0 s_1 s_2 \dots (s_0 = s) \text{ が存在して、} \\ \exists i > 0; s_i \models \phi_2 \quad \text{かつ} \quad 0 \leq \forall j < i; s_j \models \phi_1$$

$$s \models \overleftarrow{E} \phi_1 U \phi_2 \quad \text{iff} \quad \text{ある } s \text{ から始まる遡る経路 } s_0 s_{-1} s_{-2} \dots (s_0 = s) \text{ が存在して、} \\ \exists i < 0; s_i \models \phi_2 \quad \text{かつ} \quad i < \forall j \leq 0; s_j \models \phi_1$$

双方向 CTL は開始状態が Kripke 構造の順向き遷移関係から展開した CTL 木と Kripke 構造の逆向き遷移関係から展開した  $\overleftarrow{CTL}$  木の二つの木構造を持つ。

図 3.2 の左は有限 Kripke 構造、右はその構造を展開した双方向 CTL 無限木である。 $s_0$  を開始状態として、実線で書かれた木構造が CTL 木、点線で書かれた木構造が  $\overleftarrow{CTL}$  木である。

### 3.3.4 双方向 CTL の演繹体系

次は構文定義中に現れないが、構文に定義されたものに変換することで定義できる。

$$\phi_1 \vee \phi_2 \equiv \neg (\neg \phi_1 \wedge \neg \phi_2)$$

$$EF \phi \equiv E \text{ true } U \phi$$

$$EG \phi \equiv \neg (AF \neg \phi) \equiv \neg (A \text{ true } U (\neg \phi))$$

$$AF \phi \equiv A \text{ true } U \phi$$

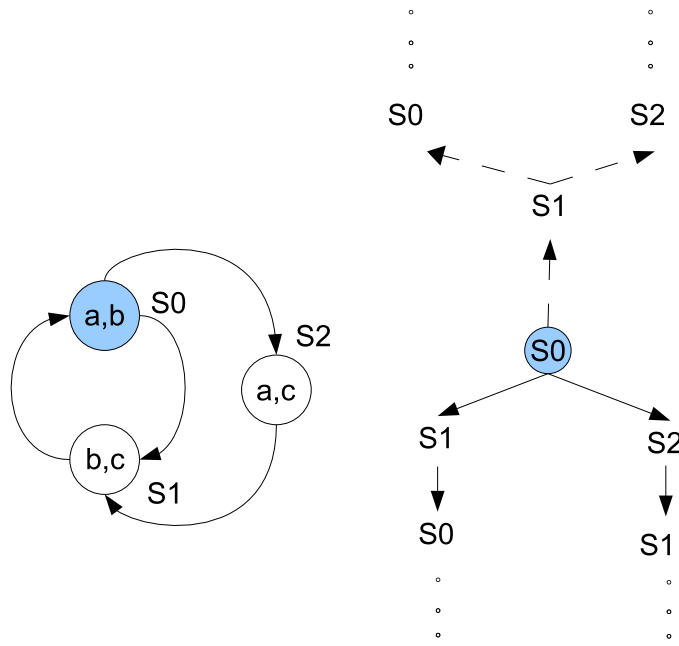


図 3.2: Kripke 構造とその構造を展開した双方向 CTL 無限木

$$AG \equiv \neg (EF \neg) \equiv \neg (E \text{ true } U (\neg))$$

$$AX \equiv \neg (EX \neg)$$

$$A_{-1}U_{-2} \equiv \neg (E \text{ true } U \neg_{-2}) \quad \neg (E \neg_{-2}U \neg_{-1})$$

$$\overleftarrow{EF} \equiv \overleftarrow{E} \text{ true } U$$

$$\overleftarrow{EG} \equiv \neg (\overleftarrow{AF} \neg) \equiv \neg (\overleftarrow{A} \text{ true } U (\neg))$$

$$\overleftarrow{AF} \equiv \overleftarrow{A} \text{ true } U$$

$$\overleftarrow{AG} \equiv \neg (\overleftarrow{EF} \neg) \equiv \neg (\overleftarrow{E} \text{ true } U (\neg))$$

$$\overleftarrow{AX} \equiv \neg (\overleftarrow{EX} \neg)$$

$$\overleftarrow{A}_{-1}U_{-2} \equiv \neg (\overleftarrow{E} \text{ true } U \neg_{-2}) \quad \neg (\overleftarrow{E} \neg_{-2}U \neg_{-1})$$

## 第4章 時相論理による最適化の記述

本章の目標は、条件付き書換え規則を用いて最適化記述を説明する。そのために、まずプログラムとプログラム上で推論を行うためのモデルを定義する。次に CTL の拡張である CTL-FV の説明をする。最後に最適化記述の例を挙げる。

### 4.1 制御フローモデル

#### 4.1.1 プログラムの構文

ここでは手続きなどがない簡単な命令型な言語を考える。

$= \text{read } X; I_1; I_2; \dots; I_{m-1}; \text{write } Y$

ここで、 $I_1, I_2, \dots, I_{m-1}$  は命令で、最初の read 文と最後の write 文を含めラベル  $n \in \text{Node} = \{0, 1, 2, \dots, m\}$  が付けられている。

命令の BNF を以下に記す。

$I ::= \text{skip} \mid X := E \mid \text{if } X \text{ goto } n \text{ else } n \mid \text{goto } n$

$E ::= X \mid E \ O \ E$

$O ::= + \mid - \mid * \mid / \mid \dots$

$X ::= \text{変数名}$

$n ::= 1 \mid 2 \mid 3 \mid \dots \mid m$

本節では、Kripke 構造として制御フローモデルを定義する。

#### 4.1.2 制御フローモデル

コード に対する制御フローモデルは三つ組み  $M( ) = (\text{Nodes}, \rightarrow, L)$  として定義される。ここで  $\text{Nodes}$  は文のラベルの集合であり、関係  $\rightarrow$  は次のように定義される関係である。 $n_1 \rightarrow n_2$  iff

$(I_{n_1} \in X := E, \text{skip}, \text{read } X)$

$n_2 = n_1 + 1$

$(I_{n_1} = \text{goto } n \quad n_2 = n)$

$(I_{n_1} = \text{if } X \text{ goto } n \text{ else } n' \quad (n_2 = n \quad n_2 = n'))$

$(I_{n_1} = \text{write } Y \quad n_2 = n_1)$

$L(n)$  は次のように  $n \in Nodes$  に対して定義される .

$$\begin{aligned}
 L(n) = & \{stmt(I_n)\} \\
 & \cup \{def(X) \mid I_n \text{ is of the form } X := E \text{ or read } X\} \\
 & \cup \{use(X) \mid I_n \text{ is of the form } Y := E \text{ with } X \text{ in } E, \\
 & \quad I_n = \text{if } X \text{ goto } n \text{ else } n', \text{ or} \\
 & \quad I_n = \text{write } X\} \\
 & \cup \{trans(E) \mid E \text{ is an expression in} \\
 & \quad \text{and for all } X \text{ in vars}(E), \\
 & \quad I_n \text{ is not of the form } : X := E' \text{ or read } X\} \\
 & \cup \{entry(n) \mid n \text{ is an entry of a program}\} \\
 & \cup \{exit(n) \mid n \text{ is an exit of a program}\}
 \end{aligned}$$

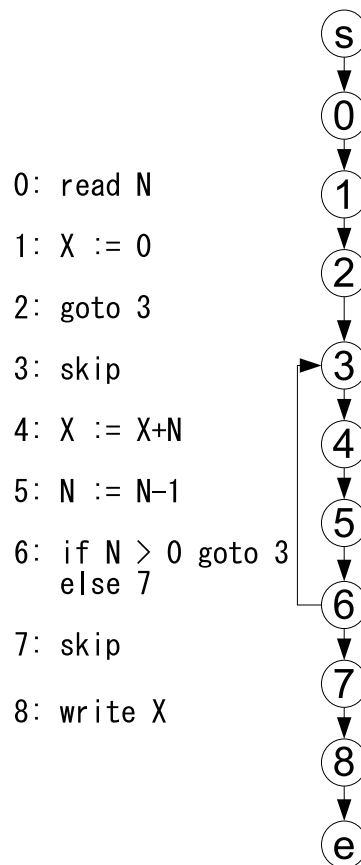


図 4.1: コードと制御フローモデルの例 ( $L(n)$  は略した)

図 4.1 のコードと制御フローモデルと対応した  $L(n)$  は下記である .

$$\begin{aligned}
L(0) &= \{stmt(readN), def(N), trans(N-1) \dots\} \\
L(1) &= \{stmt(X := 0), def(X), trans(N) \dots\} \\
&\dots \\
L(8) &= \{stmt(writeX), use(X) \dots\}
\end{aligned}$$

## 4.2 CTL-FV

プログラム変換を記述するための時相論理として, CTL-FV(CTL with free variables)[4]を用いる. CTL-FVは, CTLを拡張したものである. その拡張点に関して説明をする.

### 4.2.1 自由変数の導入

CTL-FVの一側面は一階述語論理の拡張である. CTL-FVでは(CTL-FVという名前の通り)述語の引数として自由変数が許される. 自由変数はプログラムの定義域を指定する. 自由変数は定義域に束縛されることによって, プログラムの変数や命令文などをモデル検査できる. たとえば,  $A \text{ trans}(e) U \text{ def}(v)$ は自由変数を導入したCTL式である.  $v$ は変数の定義域を指定する,  $e$ は式の定義域を指定する. つまり,

$$\begin{aligned}
v &\mapsto \{x, y, z \dots\} \\
e &\mapsto \{a + b, x + y, -z \dots\}
\end{aligned}$$

### 4.2.2 自由変数の束縛

プログラムを扱う際には, 自由変数は述語に表れて, 特定のプログラムの変数とはまだ関係付けられていない変数のことである. 自由変数の束縛は自由変数を実際のプログラムの変数と関係付けることである. たとえば,

$$\begin{aligned}
v &\mapsto \{x, y, z \dots\} \\
e &\mapsto \{a + b, x + y, -z \dots\}
\end{aligned}$$

のような自由変数がある場合,

$$\begin{aligned}
\{v ::= x, e ::= a + b\} \\
\{v ::= y, e ::= x + y\} \\
\{v ::= z, e ::= -z\} \\
\{v ::= y, e ::= a + b\}
\end{aligned}$$

...

のように束縛できる.

### 4.2.3 自由変数束縛の計算量

自由変数束縛の計算量が

$$n_{freevarBind} \approx O(n_1^{n_2})$$

$n_1$  : プログラムの大きさ (自由変数の定義域の大きさはプログラムの大きさと比例すると仮定する)

$n_2$  : 自由変数の数

### 4.3 最適化の記述

ここまでの準備で条件付き書換え規則  $I \implies I'$  if を用いて最適化を記述することができる．無用命令除去を例として条件付き書換え規則で記述した例を記す．

$$x := e \implies skip$$

$$\text{if } AX \neg E(\text{true } U \text{ use}(x))$$

これは CTL 式  $AX \neg E(\text{true } U \text{ use}(x))$  を満たした命令文  $x := e$  を  $skip$  に書換えることを意味する． $AX \neg E(\text{true } U \text{ use}(x))$  は、この命令文の次から透過のまま再度使用されるという経路がない、つまり無用命令が満たす条件である．

# 第5章 双方向CTLモデル検査器

以下は双方向 CTL のモデル検査器 (双方向 CTL モデル検査器ともいう) について述べる .

## 5.1 双方向 CTL モデル検査器のアルゴリズム

プログラムの最適化に用いるモデル検査は普通の検証と違って、時間の将来方向の流れに沿うとは限らず、過去向きを将来向きと対称的に処理できないといけない .

本研究は対称的な双方向 CTL モデル検査器を実装した . 従来研究のように過去時相を含む CTL 式を変換することなく、過去時制を含む簡潔な式をそのまま双方向モデル検査器に渡して検証することによって、変換処理時間を省いた . また、変換により式が長くなることがないため、効率が大幅に改善された .

双方向 CTL モデル検査器について、下記のプログラムと CTL 式を例として説明する .

<pre>0  r0 := @parameter0: java.lang.String[]; 1  i0 = 5; 2  i1 = 6; 3  z0 = 0; 4  z1 = 0; 5  if z0 != 0 goto label0; 6  i2 = i0 + i1; 7  goto label1; label0: 8  goto label1; label1: 9  if z0 != 1 goto label2; 10 i3 = i0 + i1; 11 goto label3; label2: 12 i4 = i0 + i1; 13 goto label3; label3: 14 return;</pre>	$\neg(E \neg \text{def}(i0) \cup (AX \text{ use}(i0)))$ <p>CTL式の例</p>
--	---

Jimple3番地コード

図 5.1: プログラムと CTL 式 (無用命令除去) の例

### 5.1.1 CTL 式の解析

CTL 式は木構造で表せる，木の葉は原子述語である．図 5.1 の CTL 式の構文木を図 5.2 に示す．各部分式を表 5.1 に示す．各部分式に深さ優先順に番号を付ける．

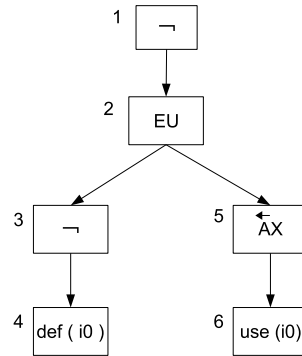


図 5.2: CTL 構文木

節番号	節演算子	子番号	対応した部分式
1	¬	2	¬ E ¬ def( $i_0$ ) U $\overleftarrow{A}X(\text{use}(i_0))$
2	EU	3 4	E ¬ def( $i_0$ ) U $\overleftarrow{A}X(\text{use}(i_0))$
3	¬	5	¬ def( $i_0$ )
5	$\overleftarrow{A}X$	6	$\overleftarrow{A} X(\text{use}(i_0))$
4	ap	def( $i_0$ )	def( $i_0$ )
6	ap	use( $i_0$ )	use( $i_0$ )

表 5.1: CTL 式とその部分式の例

### 5.1.2 モデル検査

CTL のモデル検査アルゴリズムは以下の通りである．

CTL 式を解析するとき付けられた番号の逆順で，構文木の葉から根に向かって，対象としている部分式  $\phi_n$  毎に各状態  $s$  で満たすかどうかを計算する．すなわち部分式  $\phi_n$  の真理値を  $label(\phi_n, s_i)$  の真理値を計算する．

$$label(\phi_n, s_i) = true \text{ iff } s_i \models \phi_n$$

以下は  $\phi_n$  が自明の場合， $label(\phi_n, s_i)$  を  $label_i$  に略す． $marked_i$  は  $s_i$  で探索をしたかどうかの印である．

- $\phi_n = \text{true}$  の場合  
 $label_i = true$  iff  $s_i$  で  $\text{true} = true$   
 $label_i = false$  iff  $s_i$  で  $\text{false} = false$

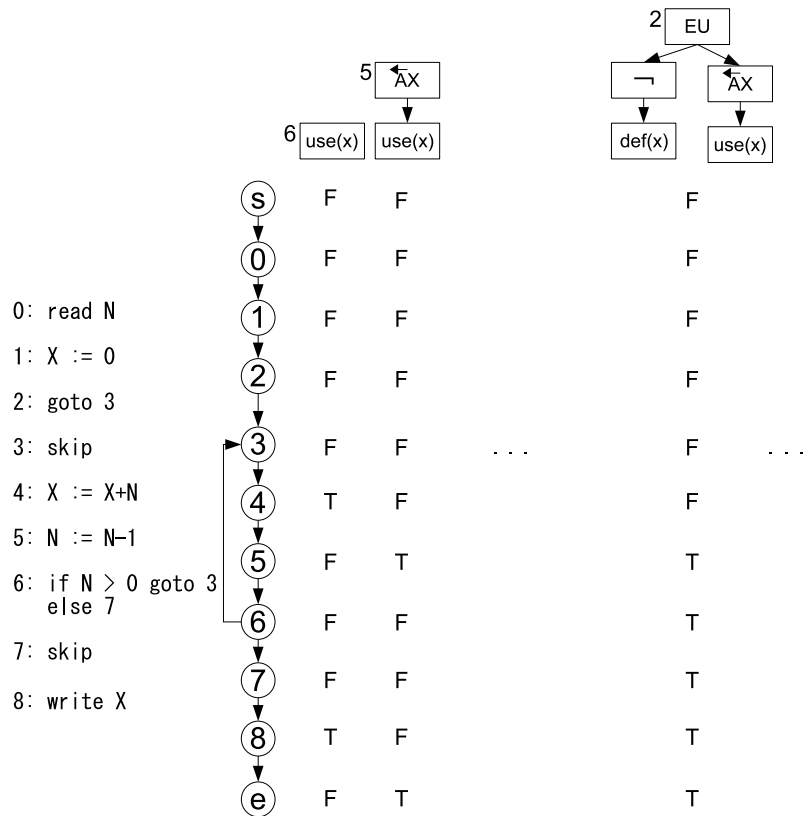


図 5.3: モデル検査の例

- $n = \neg s_n$  の場合  
 $label_i = true$  iff  $s_i$  で  $= false$   
 $label_i = false$  iff  $s_i$  で  $= true$
- $n = s_1 \wedge s_2$  の場合  
 $label_i = true$  iff  $s_i$  で  $s_1 = true$   $s_2 = true$   
 $label_i = false$  iff  $s_i$  で  $s_1 = false$   $s_2 = false$
- $n = s_1 \vee s_2$  の場合  
 $label_i = true$  iff  $s_i$  で  $s_1 = true$   $s_2 = true$   
 $label_i = false$  iff  $s_i$  で  $s_1 = false$   $s_2 = false$
- $n = EX s_j$  の場合  
 $label_i = true$  iff  $s_i$  で  $\exists j \quad s_j = true \quad (i, j) \in R$
- $n = \overline{EX s_j}$  の場合

$label_i = true$  iff  $s_i$  で  $\exists j \quad s_j = true \quad (j, i) \in R$

- $= \overleftarrow{E} \quad {}_1 U \quad {}_2$  の場合

基本的な考えは  ${}_2$  が満たした状態から将来向きにたどって行き,  ${}_1$  が成り立つ状態の label 値を true にする .

初期処理としてすべての状態の label 値を false にする .

${}_2$  が true の状態とその他の状態という二種類に分け, それぞれ集合に入れておく .  ${}_2$  が true の状態を true-marked 集合に入れ, その他の状態を unmarked 集合に入れる .

true-marked 集合の項  $s_i$  に下記の処理をする .

- $s_i$  のすべての後続ノード  $s_j (s_j \in unmarked)$  について調べる

- \*  $s_j (s_j \in unmarked) \models {}_1$  の場合は

- $label_j = true$

- $s_j$  を true-marked 集合に入れる

- $s_j$  を unmarked 集合から削除する

- \*  $s_j (s_j \in unmarked) \not\models {}_1$  の場合は

- $label_j = false$

- $s_j$  を unmarked 集合から削除する

- true-marked が空になるまで繰り返す

- $= E \quad {}_1 U \quad {}_2$  の場合

$= \overleftarrow{E} \quad {}_1 U \quad {}_2$  の場合の後続ノードを先行ノードに読み替える, その他省略する .

- $= A \quad {}_1 U \quad {}_2$  の場合

その状態から始まる経路のすべて  ${}_1$  を満たす状態のみを經由して  ${}_2$  を満たす状態にたどり着くものを, 探索済みの状態に印を付けながら深さ優先探索で探索する .

初期化処理として, すべての状態の探索済みの印を  $false$  に, すべての状態の  $label_i = true$  にする . 探索している状態  $s_i$  に関して

- $s_i$  で探索済みの印がすでに付いているとき,  $label_i$  の値を返す .
- $s_i$  で探索済みの印がまだ付いていないとき, 探索済みの印を付ける .
- $s_i$  で  ${}_1$  が満たしていない, または出口の場合は  $label_i = false$  .
- $s_i$  で  ${}_1$  と  ${}_2$  が満たしている場合は  $label_i = true$  .

- $s_i$ で  $\phi_1$  が満たされ、  $\phi_2$  が満たされない場合は  $s_i$  から遷移可能なすべての状態  $s_j (j \in succ(i))$  に対して再帰的に探索を行い、後続状態の label 値の積とする。

$$label_i = \prod label_j, (i, j) \in R$$

以下に  $s_2$  での  $A \cup U$  値の計算を例として説明を記す。

- 普通の場合

下記の構造  $M$  を例として説明を記す。

$$M = (S, s_{start}, R, L)$$

$$S = \{1, 2, 3\}$$

$$s_{start} = s_2$$

$$R = \{(1, 2), (2, 3)\}$$

$$L = \{1 \mapsto \{\phi_1\}, 2 \mapsto \{\phi_1\}, 3 \mapsto \{\phi_1, \phi_2\}\}$$

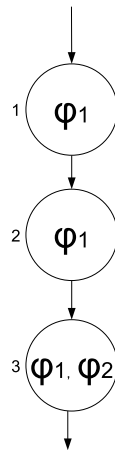


図 5.4: AU の例 (普通の場合)

初期処理として、 $marked_1 \sim marked_3$  を  $false$ 、 $label_1 \sim label_3$  を  $true$  にする。

- \* 探索済みの印がまだ付いていないとき、探索済みの印を付ける。 $marked_2 = true$
- \*  $\phi_1$  が満たされ、 $\phi_2$  が満たされない場合は  $label_i = \prod label_j, (i, j) \in R$  したがって  $label_2 = \prod label_3 = \text{未定}$
- \*  $s_3$  は  $\phi_1$  と  $\phi_2$  が成り立っているため、 $label_3 = true$
- \*  $label_2 = \prod label_3 = true$

- loop がある場合

下記の構造  $M$  で  $s_2$  での  $A_{1U_2}$  値の計算を例として説明を記す .

$$M = (S, s_{start}, R, L)$$

$$S = \{1, 2, 3\}$$

$$s_{start} = s_2$$

$$R = \{(1, 2), (2, 2), (2, 3)\}$$

$$L = \{1 \mapsto \{ \_1 \}, 2 \mapsto \{ \_1 \}, 3 \mapsto \{ \_1, \_2 \}\}$$

初期処理として ,  $marked_1 \sim marked_3$  を  $false$  に ,  $label_1 \sim label_3$  を  $true$  にする .

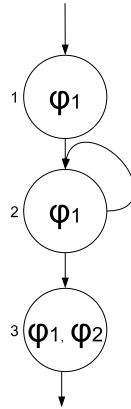


図 5.5: AU の例 (loop の場合)

\* 探索済みの印がまだ付いていないとき , 探索済みの印を付ける .  $marked_2 = true$

\*  $\_1$  が満たされ、  $\_2$  が満たされない場合は

$$label_i = \prod label_j, (i, j) \in R$$

したがって

$$label(A_{1U_2}, 2) = \prod\{label_2, label_3\} = \text{未定}$$

・ 探索済みの印がすでに付いているとき ,  $label_i$  の値を返す .

$$label_2 = label_2(\text{default}) = true$$

・  $\_1$  と  $\_2$  が成り立っている場合は  $label_i = true$

したがって ,  $label_3 = true$

\*  $label_2 = \prod\{label_2, label_3\} = true$

- 互いに行き来するループの場合

初期処理として ,  $marked_1 \sim marked_6$  を  $false$  に ,  $label_1 \sim label_6$  を  $true$  にする .

下記の構造  $M$  を例として説明する .

$$M = (S, s_{start}, R, L)$$

$$S = \{1, 2, 3, 4, 5, 6\}$$

$$s_{start} = s_2$$

$$R = \{(1, 2), (2, 3), (2, 5), (4, 5), (5, 2), (5, 6)\}$$

$$L = \{1 \mapsto \{ \quad \}_1, 2 \mapsto \{ \quad \}_1, 3 \mapsto \{ \quad \}_1, 4 \mapsto \{ \quad \}_1, 5 \mapsto \{ \quad \}_1, 6 \mapsto \{ \quad \}_1, 2\}$$

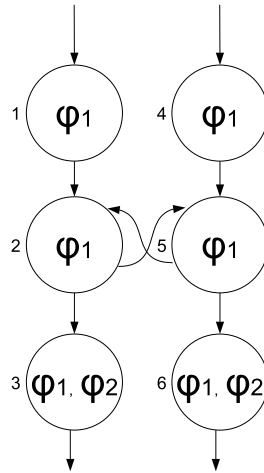


図 5.6: AU の例 (互いに行き来する loop の場合)

- \* 探索済みの印がまだ付いていないとき, 探索済みの印を付ける. したがって,  $marked_2 = true$
- \*  $\phi_1$  が満たされ,  $\phi_2$  が満たされない場合は  $label_i = \prod label_j, (i, j) \in R$  したがって,  $label_2 = \prod\{label_3, label_5\} = \text{未定}$ 
  - ・ 探索済みの印が付いていないとき, 探索済みの印を付ける. したがって,  $marked_5 = true$
  - ・  $\phi_1$  が満たされ,  $\phi_2$  が満たされない場合は  $label_i = \prod label_j, (i, j) \in R$  したがって,  $label_5 = \prod\{label_2, label_6\} = \text{未定}$ 
    - ★ 探索済みの印がすでに付いているとき,  $label_i$  の値を返す. したがって,  $label_2 = label_2(\text{default}) = true$
    - ★ 探索済みの印が付いていないとき, 探索済みの印を付ける. したがって,  $marked_6 = true$
    - ★  $\phi_1$  と  $\phi_2$  が成り立っている場合は  $label_i = true$ , したがって,  $label_6 = true$
- ・  $label_5 = \prod\{label_2, label_6\} = true$
- ・ 探索済みの印が付いていないとき, 探索済みの印を付ける. したがって,  $marked_3 = true$
- ・  $\phi_1$  と  $\phi_2$  が成り立っている場合は  $label_i = true$  したがって,  $label_3 = true$

$$* \text{label}_2 = \prod\{\text{label}_2, \text{label}_3\} = \text{true}$$

- $\overline{A} \text{ }_1 U \text{ }_2$  の場合  
 $= A \text{ }_1 U \text{ }_2$  の場合の後続ノードを先行ノードに置き換える．その他省略する．

### 5.1.3 モデル検査の擬似コード

以下で， $x$  は CTL 構文木のノード番号で， $\text{node } x$  は，その番号のつけられたノードである．ノード番号については，5.1.1 節で説明した．次により，CTL 構文木の葉からはじめて葉から根の向きに，各ノードが処理できる．

```

for x = |   | to 1 step -1 do
  switch node x in
    case ap   :
      foreach s ∈ S do
        label(   ,s) = (   ∈ L(s))
      done
    case NOT (  _1)
      foreach s ∈ S do
        label(   ,s) = ¬ label(  _1 ,s)
      done
    case AND (  _1 ,  _2)
      foreach s ∈ S do
        label(   ,s) = label(  _1 ,s) ∧ label(  _2 ,s)
      done
    case EX(  )
      foreach s ∈ S do
        label(   ,s) = ∑t∈succs(s) label(  _t)
      done
    case  $\overline{E}X$ (  )
      foreach s ∈ S do
        label(   ,s) = ∑t∈preds(s) label(  _1 ,t)
      done
    case EU(  _1 ,  _2)
      foreach s ∈ S do
        label(   ,s) = checkEU(  _1 ,  _2)
      done
    case  $\overline{E}U$ (  _1 ,  _2)
      foreach s ∈ S do

```

```

    label( $\cdot, s$ ) = check $\overleftarrow{E}U(\cdot_1, \cdot_2)$ 
done
case  $AU(\cdot_1, \cdot_2)$ 
  foreach  $s \in S$  do
    label( $\cdot, s$ ) = check $AU(\cdot_1, \cdot_2)$ 
  done
case  $\overleftarrow{A}U(\cdot_1, \cdot_2)$ 
  foreach  $s \in S$  do
    label( $\cdot, s$ ) = check $\overleftarrow{A}U(\cdot_1, \cdot_2)$ 
  done

```

```

check $EX(\cdot)$ :
  foreach  $s \in S$  do
    if labeled( $\cdot, s$ )=true
      foreach  $t \in preds(s)$  do
        labeled( $\cdot, t$ )=true
      done
    end if
  done

```

check $\overleftarrow{E}X(\cdot)$ :  
 check $EX(\cdot)$  の  $preds(s)$  を  $succs(s)$  にする . その他省略する .

```

check $\overleftarrow{E}U(\cdot_1, \cdot_2)$ :
  true-marked =  $\emptyset$ 
  unmarked =  $\emptyset$ 
  foreach  $s \in S$  do
    labeled( $\cdot, s$ )=false
    if label( $\cdot_2, s$ ) = true then
      input s to true-marked
    else input s to unmarked
    end if-else
  done
  foreach  $s \in true\text{-marked}$  do
    delete s form true-marked
    foreach  $t \in succs(s) \quad t \in unmarked$ 
      delete t form unmarked
      if label( $\cdot_1, t$ ) = true then
        label( $\cdot, t$ ) = true
        input t to true-marked
      end if
    end foreach
  end foreach

```

done  
done

$checkEU(\phi_1, \phi_2)$ :  
 $check\overleftarrow{EU}(\phi_1, \phi_2)$  の  $succs(s)$  を  $preds(s)$  にする . その他省略する .

$checkAU(\phi_1, \phi_2)$ :  
foreach  $s \in S$  do  
   $labeled(\phi, s) = true$   
   $marked_s = false$   
done  
foreach  $s \in S$  do  
  if  $\neg marked_s$  then  
     $subAU(\phi, s)$   
  end if  
done

$subAU(\phi, s)$ :  
if  $marked_s$  then  
  return  $label(\phi, s)$   
end if  
 $marked_s = true$   
if  $label(\phi_1, s) = false$  then  
   $label(\phi, s) = false$   
end if  
if  $label(\phi_1, s) = true \wedge label(\phi_2, s) = true$  then  
   $label(\phi, s) = true$   
end if  
if  $label(\phi_1, s) = true \wedge label(\phi_2, s) = false$  then  
   $label(\phi, s) = \prod\{subAU(\phi, t) \mid t \in succs(s)\}$   
end if

$check\overleftarrow{AU}(\phi_1, \phi_2)$   
 $checkAU(\phi_1, \phi_2)$  の  $succs(s)$  を  $preds(s)$  にする . その他省略する .

## 5.2 モデル検査の計算量

モデル検査の計算量が

$$n_{modelcheck} = O(n_1 \times n_3)$$

$n_1$  : プログラムの大きさ

$n_3$  : CTL 構文木のノード数

モデル検査器の計算量はプログラムの大きさと CTL 式の大きさに比例する .

## 第6章 本研究の最適化の記述

本章は本研究の最適化の記述を説明する。

### 6.1 最適化の記述

本研究はCTLによる最適化器生成の実用化に向けて、最適化器生成の重要な一環である最適化記述を改善した。以下は本研究の記述を述べる。

#### 6.1.1 最適化記述の文法

$$\begin{aligned}
\textit{Rule-top} & ::= \textit{MATCH Match-pattern} \\
& \quad \textit{PROCESS Process-pattern} \\
& \quad \textit{CONDITION Condition-pattern} \\
\textit{Match-pattern} & ::= \textit{Statement-pattern} \\
\textit{Process-pattern} & ::= \textit{Process}_1, \textit{Process}_2, \textit{Process}_3, \dots \\
\textit{Condition-pattern} & ::= \textit{Condition}_1, \textit{Condition}_2, \textit{Condition}_3, \dots \\
\textit{Process} & ::= \textit{Point-macro: InsertBefore ProcessStmt-pattern} \\
& \quad | \textit{Point-macro: InsertAfter ProcessStmt-pattern} \\
& \quad | \textit{Point-macro: Delete ProcessStmt-pattern} \\
& \quad | \textit{Point-macro: Replace ProcessStmt-pattern} \\
& \quad | \textit{Point-macro: Replace Expr-pattern} \rightarrow \textit{ProcessExpr-pattern} \\
& \quad | \textit{Edge-macro: EdgeSplit ProcessStmt-pattern} \\
\textit{Condition} & ::= \textit{Point-macro: CTL-pattern} \\
& \quad | \textit{Edge-macro: Point-macro} \longrightarrow \textit{Point-macro} \\
\textit{Point-macro} & ::= \textit{point\_文字列} \\
\textit{Edge-macro} & ::= \textit{edge\_文字列} \\
\textit{CTL-pattern} & ::= \\
& ::= \textit{ap} \mid \neg \mid \_1 \_2 \mid \_1 \vee \_2 \\
& \quad | \textit{EX} \mid \textit{AX} \mid \textit{E} \_1 \textit{U} \_2 \mid \textit{A} \_1 \textit{U} \_2 \\
& \quad | \textit{EF} \mid \textit{AF} \mid \textit{EG} \mid \textit{AG} \\
& \quad | \overleftarrow{\textit{EX}} \mid \overleftarrow{\textit{AX}} \mid \overleftarrow{\textit{E}} \_1 \textit{U} \_2 \mid \overleftarrow{\textit{A}} \_1 \textit{U} \_2 \\
& \quad | \overleftarrow{\textit{EF}} \mid \overleftarrow{\textit{AF}} \mid \overleftarrow{\textit{EG}} \mid \overleftarrow{\textit{AG}} \\
\textit{ap} & ::= \textit{use(Expr-pattern)} \\
& \quad | \textit{def(Expr-pattern)} \\
& \quad | \textit{trans(Expr-pattern)} \\
& \quad | \textit{stmt(Statement-pattern)} \\
& \quad | \textit{true} \\
& \quad | \textit{false} \\
& \quad | \textit{entry} \\
& \quad | \textit{exit} \\
\textit{Statement-pattern} & ::= \textit{v := Expr-pattern} \\
\textit{ProcessStmt-pattern} & ::= \textit{Statement-pattern} \\
& \quad | \textit{temp := Expr-pattern} \\
& \quad | \textit{v := temp} \\
\textit{Expr-pattern} & ::= \textit{e} \mid \textit{b} \mid \textit{r} \mid \textit{v} \mid \textit{c} \\
\textit{ProcessExpr-pattern} & ::= \textit{e} \mid \textit{b} \mid \textit{r} \mid \textit{v} \mid \textit{c} \mid \textit{temp}
\end{aligned}$$

最適化記述の文法の記号について Appendix で説明する .

## 6.2 最適化記述の文法の説明

本システムの最適化の記述はMATCH, CONDITION, PROCESSの3つ部分から成る。MATCHはモデル検査の対象となる命令文の形である。CONDITIONは命令文が最適化される時満たすべき双方向CTL式である。PROCESSはCONDITIONの条件式が満たされたとき、どのように処理するかを書く所である。最適化記述は次のような形をしている。

**MATCH**

変数 := 式

**CONDITION**

point\_文字列 : CTL 式

edge\_文字列 : point\_文字列  $\rightarrow$  point\_文字列

**PROCESS**

point\_文字列 : Comand 命令文

point\_文字列 : Replace 式  $\rightarrow$  式

edge\_文字列 : EdgeSplit 命令文

### 6.2.1 MATCH 部

MATCH部は最適化の対象式の形を決める。たとえば

**MATCH**

$v := b$

は右辺が二項式の形の文を対象とする。 $v$ は変数, $b$ は二項式である。 $z := x + y$ は対象となる。 $x := y$ は対象とならない。文 $z := x + y$ が最適化の対象となったとき, $\{v \mapsto z, b \mapsto x + y\}$ のようにの集合 $\{v, b\}$ をプログラムの変数や式の $\{x, y, z, x + y\}$ の集合に一对一に関係付け,全組み合わせによる束縛を避けることができる。

### 6.2.2 CONDITION 部

CONDITION部では条件式と部分式を複数書いたり,それらに名前をつけることができる。

条件式は書換えをする時に成り立つべき条件である。条件式にPROCESS部での処理と対応させるための名前を付ける。

部分式は長い条件式を書きやすいように,分解して書けるようにするためのものである。条件式に部分式の名前が書かれているときは,その名前が代表する論理式に置換えてモデル検査を行う。

辺についても条件式を書くことができる。辺についての条件式はCTL木構造と関係がなく,どのような条件式を満たしたノードからどのような条件式を満たしたノードを指しているかを示している。これもCTLモデル検査の結果によって計算される。

これらの目的で付けた名前は従来研究でのノード番号と異なり，自由変数ではない．

### 6.2.3 PROCESS 部

PROCESS 処理部には CONDITION 部の条件式を満たした命令文または辺の集合をどのように処理するかが書かれている．処理式に条件式と同じ名前を付けることによって，条件式との対応関係をつける．

処理式には命令文 *Comand* の *InsertBefore* , *InsertAfter* , *Delete* , *Replace* と辺 *Comand* の *EdgeSplit* がある．各 *Comand* の意味は文字通りで命令文を前に挿入，後に挿入，削除，書換えと辺に挿入を行う．式の一部を書き換えるときは，*Replace* 式 → 式で表す．

point の処理は命令文を対象，edge は辺を対象とする．

命令文や式は PROCESS 部に現れるとき一時変数を含むことができるが，CONDITION 部に現れるときは含むことができない．

上記の記号は複数個使用するとき，添え字をつけて区別する．たとえば  $v_1$  ,  $c_2$  ,  $temp_0$  . . . .

## 6.3 本記述の利点

既存研究では定式化に命令文と対応した Kripke 構造のノード番号を書くことになっている．

このような記述は大きな欠点がある．まず，Kripke 構造のノード番号の束縛による計算の爆発が予測される．次は，実際のプログラムでは，条件式を満たすのは特定の番号の命令文ではなく，特定の種類の命令文であるとき，番号を用いる定式化ができるが強引的である．Kripke 構造のノード番号を記述しないのは以下の 2 つの利点がある．

### 6.3.1 複雑な定式化が容易に

モデル検査器が計算するのは条件式を満たす特定の番号の命令文ではなく，条件式を満たす命令文の集合である．これにより同じ条件式を満たす多数の命令文を書き換える処理が容易に記述できるようになった．

### 6.3.2 自由変数の減少

また，後述のように効率を向上させることもできる．

以下は無用命令文除去の最適化記述である：

## MATCH

$v := e$

## CONDITION

$point\_delete : AX((AG \neg use(v))$   
 $A \neg use(v) U def(v))$

## PROCESS

$point\_delete : delete\ v := e$

上記の最適化記述にはが  $\{v, e\}$  の2つある。  
Lacey らの研究では，下記のように記述する。

$n : v := e \implies skip$   
if  $n \models AX((AG \neg use(v))$   
 $A \neg use(v) U def(v))$

これは  $\{v, e, n\}$  の3つあり，下記のように，このことは効率に影響する。

従来研究の最適化記述は命令文と対応した Kripke 構造のノード番号を書くことになっているが，本研究の記述は Kripke 構造のノード番号を書かなくてよい。上記の例では，自由変数を1つ減らした。

自由変数はシステムの効率に大きく影響する (4.2.3 節を参照) ため，自由変数を減らすと，最適化時間が大きく短縮できる。

## 6.4 本研究で実装した定式化

本研究は下記の最適化を定式化した。

- 無用命令除去
- コピー伝播 (定数伝播を含む)
- 部分冗長性除去 (共通部分式除去，ループ不変式移動を含む)

無用命令除去 (2.1.1 節) やコピー伝播 (2.1.2 節) は簡単な最適化であり，本研究において最適化を定式化するとき，最適化の条件をもとにした CTL を記すことができる。

部分冗長性除去は複雑な最適化 (2.1.5 節) であり，最適化の条件をもとにした CTL を記すことが可能と思われるが，複雑な論理式を証明するのが難しい。

部分冗長性除去は長年にわたり研究された，多数のアルゴリズムがある。本研究は Paleri らの「A Simple Algorithm for Partial Redundancy Elimination」を採用した [17]。このアルゴリズムの特徴はシンプルであること。変換の結果の共通部分式の計算回数の最適性が証明されていること，である。

本研究では下記の最適化を CTL で定式化し，そこから実用的な時間で最適化できる最適化器を生成した。

### 6.4.1 無用命令除去

以下で  $v$  は変数,  $e$  は式である,  $v := e$  が  $point\_delete$  の条件式を満たすと無用命令である. 本論文の例とした無用命令除去の CTL 式は簡単の場合しかないが, 実装にあたって, プログラムの入口から到達しない, これから使われない, 再定義するまで使われないの 3 つの場合が考えられる.

- 無用命令除去の CTL 最適化記述

*MATCH*

$v := e$

*CONDITION*

$point\_unreachable : \neg(\overleftarrow{E} \text{ true } U \text{ entry})$

$point\_unuse : AG \neg use(v)$

$point\_unuseTillDef : A \neg use(v) U def(v)$

$point\_delete : stmt(v := e) \quad AX(point\_unuse \vee point\_unuseTillDef)$

$\vee point\_unreachable$

*PROCESS*

$point\_delete : delete \ v := e$

- CTL 式の説明

この式の意味を, 双方向 CTL 木 (図 3.1) を用いて説明する. 違う色の影はその部分木をカバーしている, 影の一番下にこの部分木の説明を記す.

- CTL 式の証明

図 6.1 からわかるように, 証明は自明である.

### 6.4.2 コピー伝播 (定数伝播を含む)

以下で  $v$  は変数,  $r$  は式である,  $v := r$  が  $point\_copypropagate$  の条件式を満たすときはコピー伝播できる.

- コピー伝播 (定数伝播) の CTL 最適化記述

*MATCH*

$v := r$

*CONDITION*

$point\_trans : trans(v) \quad trans(r)$

$point\_avail : \overleftarrow{A}(point\_trans \ U \ (\overleftarrow{A}X(stmt(v := r))))$

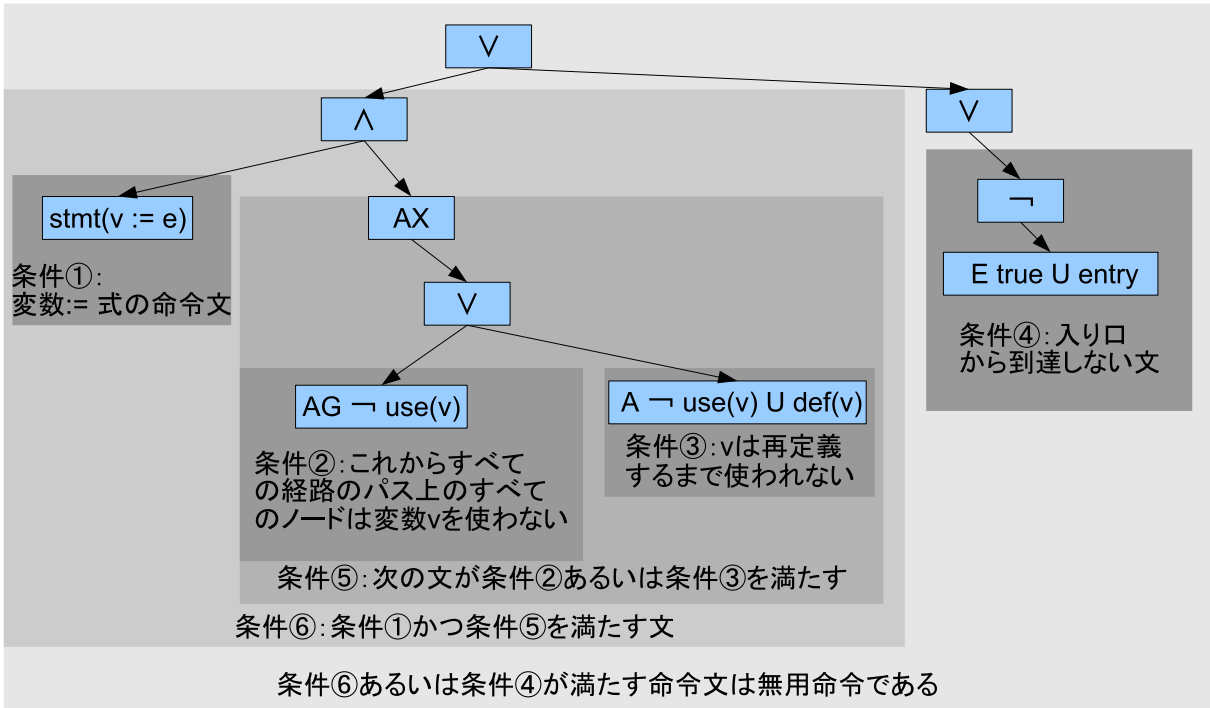


図 6.1: 双方向 CTL 木による CTL 式の説明-無用命令除去

$point\_next : \overleftarrow{A} X stmt(v := r)$

$point\_copypropagate : use(v) \wedge (\overleftarrow{A} X point\_avail \vee point\_next)$

PROCESS

$point\_copypropagate : replace\ r \rightarrow r$

- CTL 式の説明

この式の意味を，双方向 CTL 木 (図 6.2) を用いて説明する．違う色の影はその部分木をカバーしている，影の一番下にこの部分木の説明を記す．

- CTL 式の証明

図 6.2 からわかるように，証明は自明である．

### 6.4.3 部分冗長性除去 (共通部分式除去，ループ不変式の移動)

前述のとおり，Paleri らのデータフロー方程式をもとに CTL 式を記す．

- 部分冗長性除去のデータフロー方程式

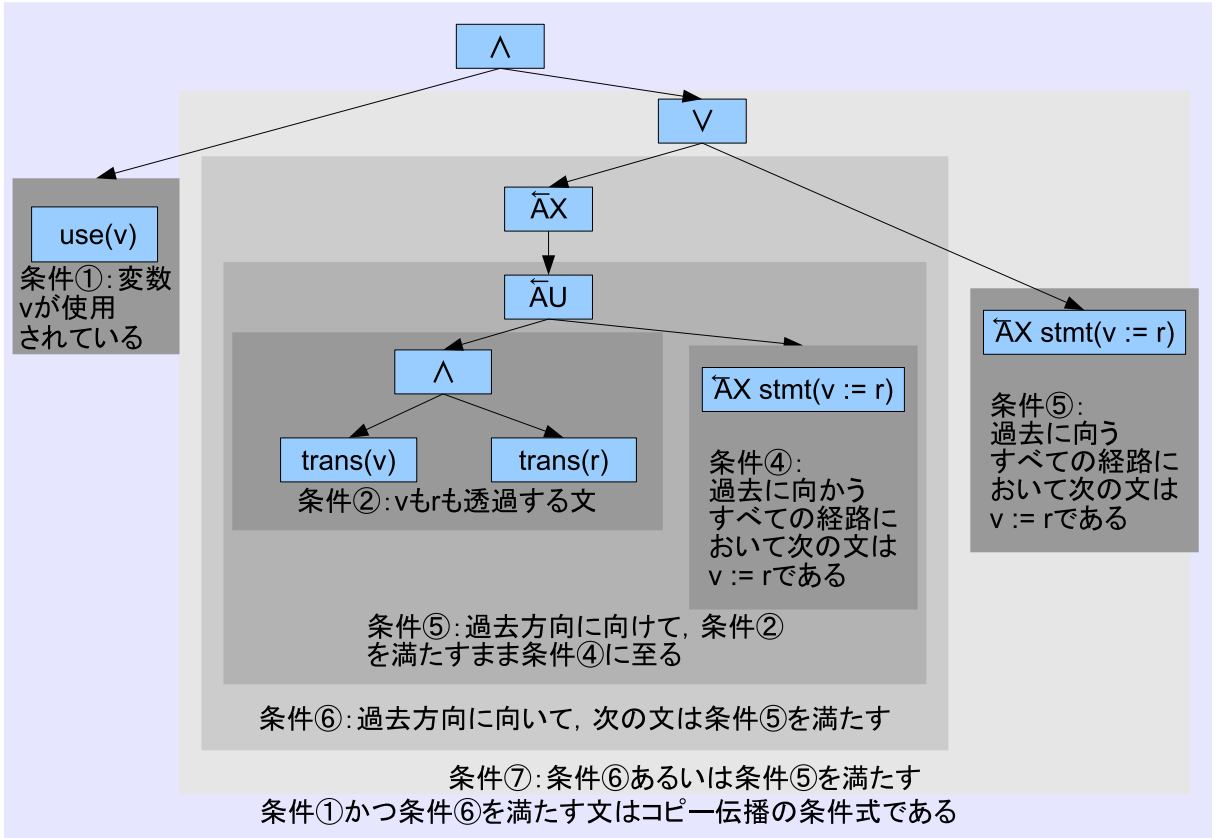


図 6.2: 双方向 CTL 木による CTL 式の説明-コピー伝播

$$\begin{aligned}
 TRANSP_i &= trans(e) \\
 COMP_i &= use(e) \cdot TRANSP_i(e) \\
 ANTLOC_i &= use(e) \cdot TRANSP_i(e) \\
 AVIN_i &= \begin{cases} false & \text{if } entry \\ \prod_{j \in preds(i)} AVOUT_j & \text{if } otherwise \end{cases} \\
 AVOUT_i &= COMP_i + AVIN_i \cdot TRANSP_i \\
 ANTOUT_i &= \begin{cases} false & \text{if } exit \\ \prod_{j \in preds(i)} ANTIN_j & \text{if } otherwise \end{cases} \\
 ANTIN_i &= ANTLOC_i + ANTOUT_i \cdot TRANSP_i \\
 SAFEIN_i &= AVIN_i + ANTIN_i \\
 SAFEOUT_i &= AVOUT_i + ANTOUT_i \\
 SPAVIN_i &= \begin{cases} false & \text{if } i = entry \text{ or } \neg SAFEIN_i \\ \sum_{j \in preds(i)} SPAVOUT_j & \text{if } otherwise \end{cases} \\
 SPAVOUT_i &= \begin{cases} false & \text{if } i = entry \text{ or } \neg SAFEOUT_i \\ COMP_i + SPAVIN_i \cdot TRANSP_i & \text{if } otherwise \end{cases} \\
 SPANTOUT_i &= \begin{cases} false & \text{if } i = entry \text{ or } \neg SAFEOUT_i \\ \sum_{j \in preds(i)} SPANTIN_j & \text{if } otherwise \end{cases} \\
 SPANTIN_i &= \begin{cases} false & \text{if } i = entry \text{ or } \neg SAFEIN_i \\ ANTLOC_i + SPANTOUT_i \cdot TRANSP_i & \text{if } otherwise \end{cases} \\
 INSERT_i &= COMP_i \cdot \neg SPAVIN_i \cdot SPANTOUT_i \\
 INSERT_{i,j} &= \neg SPAVOUT_i \cdot SPAVIN_j \cdot SPANTIN_j
 \end{aligned}$$

$$REPLACE_i = ANTLOC_i \cdot SPAVIN_i + COMP_i \cdot SPANTOUT_i$$

図 6.3(a) から (b) への変換はこの方程式による部分冗長性除去の例である .

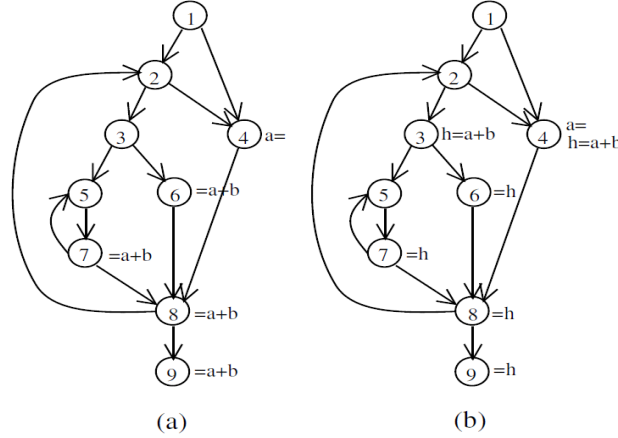


図 6.3: A Simple Algorithm for Partial Redundancy Elimination の例

- 部分冗長性除去 (共通部分式とループ不変式も含む) の CTL 最適化記述

以下で  $v$  は変数,  $e$  は式である,  $point\_insert$  は式の計算を挿入する場所,  $point\_replace$  は式の計算を書換える場所,  $edge\_split$  は式の計算を挿入する辺である .

*MATCH*

$v := e$

*CONDITION*

$point\_comp : use(e) \quad trans(e)$

$point\_avin : \overleftarrow{A} X(\overleftarrow{A} \ trans(e) \ U \ use(e))$

$point\_avout : point\_comp \ (point\_avin \ trans(e))$

$point\_antout : AX(A \ trans(e) \ U \ use(e))$

$point\_antin : point\_comp \ (point\_antout \ trans(e))$

$point\_safein : point\_avin \ point\_antin$

$point\_safeout : point\_avout \ point\_antout$

$point\_spavin : point\_safein \ \overleftarrow{E} X(\overleftarrow{E} \ (trans(e) \ (point\_safeout)) \ U \ use(e))$

$point\_spavout : point\_safeout \ (point\_comp \ (point\_spavin \ trans(e)))$

$point\_spantout : point\_safeout \ EX(E \ (trans(e) \ (point\_safein)) \ U \ use(e))$

$point\_spantin : point\_safein \ (point\_comp \ (point\_spantout \ trans(e)))$

$point\_insert : point\_comp \quad \neg point\_spavin \quad point\_spantout$   
 $point\_replace : (point\_comp \quad point\_spavin) \quad (point\_comp \quad point\_spantout)$   
 $point\_edge1 : point\_spavin \quad point\_spantin$   
 $point\_edge2 : \neg point\_spavout$   
 $edge\_split : point\_edge1 \longrightarrow point\_edge2$   
 PROCESS  
 $point\_insert : InsertBefore \ temp := e$   
 $edge\_split : EdgeSplit \ temp := e$   
 $point\_replace : replace \ e \rightarrow temp$

• CTL 式の説明

前述のデータフロー方程式をもとに CTL 式を記すとき，殆どデータフロー方程式をそのまま書けばよい，たとえば  $point\_comp$  . しかし，データフロー方程式は，データフロー情報の値が決まる所からはじめて，収束するまで繰り返す方法で解いていく事ができる．一方，CTL 式はデータフロー方程式と同じように収束するまで繰り返すことができないので，データフロー方程式のセマンティクスを保ちつつ，繰り返しなしで計算できるような式にするように工夫する必要がある．

たとえば

$$\begin{aligned}
 AVIN_i &= \begin{cases} false & \text{if entry} \\ \prod_{j \in preds(i)} AVOUT_j & \text{if otherwise} \end{cases} \\
 AVOUT_i &= COMP_i + AVIN_i \cdot TRANS_i
 \end{aligned}$$

のようなデータフロー方程式は

- $AVIN_i$  の値はすべての先行ノードの  $AVOUT_j (j \in preds(i))$  によって計算される．
- $AVOUT_i$  の値は  $AVIN_i$  の値によって計算される．

を意味する．

この式はデータフロー方程式の解法では

$$\begin{aligned}
 AVIN_0 &= false \\
 AVOUT_0 &= COMP_0 + AVIN_0 \cdot TRANS_0 \\
 AVIN_1 &= AVOUT_0 \\
 AVOUT_1 &= COMP_1 + AVIN_1 \cdot TRANS_1
 \end{aligned}$$

$AVIN_2$

...

のようにして解けるが、このような解き方は CTL 式のモデル検査では実現できない。

上の式のセマンティクスは

- $AVIN_i$  は式  $e$  の計算がされた後、すべての経路を経ても値が変わらなく、その値の再計算をしなくても良い場所でのみ true .
- $AVOUT_i$  は式が計算されかつ変更されない個所、またはそれ以前の計算がまだ利用可能であって、このノードの計算によって値が変更しない個所でのみ true .

となっている。

以上の意味に沿い、CTL 式は

$point\_avin : \overleftarrow{A} X(\overleftarrow{A} trans(e) U use(e))$

$point\_avout : point\_comp (point\_avin trans(e))$

のようにすればよい。

- CTL 式の証明

CTL 式証明は文献 [17] に上のようなセマンティクスを保つ変換を加えればよい。

#### 6.4.4 CTL による定式化のノーハウ

同じ効果の最適化は、CTL による定式化が異なると、モデル検査の時間に大きい差が出る。

- 自由変数の影響

一番時間に影響するのが自由変数の数であるので、できるだけ自由変数を少なめに書くのがコツである。コピー伝播を例とする。結果は下記の通りである。

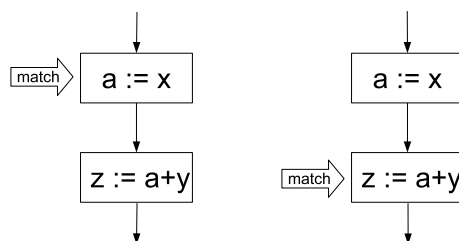


図 6.4: 同じ最適化を行う異なった CTL 式

図 6.4 左のように上の文に match させる方法で記述すると，下記の CTL 式になる．

*MATCH*

$v1 := v2$

*CONDITION*

$point\_next : \overline{A}X(stmt(v1 := v2))$

$point\_trans : trans(v1) \quad trans(v2) \quad trans(v4)$

$point\_copypropagate : stmt(v3 := v4) \quad (\overline{A}X(\overline{A} point\_trans U point\_next))$

*PROCESS*

$point\_copypropagate : replace\ v4 \rightarrow v2$

ここで図 6.4 左の下の文は *point\_copypropagate* の条件式を満たす．自由変数は  $v1, v2, v3, v4$  の 4 つがある．

一方，図 6.4 右のように下の文に match させる方法で記述すると，CTL 式は 6.4.2 節のように書ける．自由変数は  $v, r$  の 2 つである．

ここで図 6.4 右の下の文は *point\_copypropagate* の条件式を満たす．

6.4.2 節のように記述すると自由変数が 4 から 2 に減り，実験の結果はで後述するように，最適化の時間が大幅に減る．

- 式の長さの影響

CTL による定式化のしかたは一通りではなく，違う見方でとらえると違う CTL 式になる．無用命令除去を例として説明する．

- 「これから使わないか，再定義するまで使わない」ととらえると下記の式になる．

- $\neg AX((AG \neg use(v)) \quad (A \neg use(v)Udef(v)))$

- 「次の文から定義しないまま再使用するまでに至る経路がない」ととらえると下記の式になる．

- $\neg EX(E(\neg def(v))Uuse(v))$

このように二通りの方法で定式化できるが，後者のほうが式が短いため，最適化の効率がよい．式の長さによる時間への影響はおよそ式の長さに線形である．

## 第7章 双方向CTLによる最適化器

本章は双方向CTLによる最適化器の生成について述べる。

双方向CTLによる最適化器は前処理部，モデル検査器と書換え部から構成される。前処理部は，入力をモデル検査に適切な形に変換し，それらの情報をモデル検査器に渡す。モデル検査器がモデル検査を行い，その結果を書換え処理部に渡す。書換え部は最適化の書換え規則を適用し，その結果の中間コードを出力する。

### 7.1 前処理部

前処理部の主な部分を説明する。

前処理部は入力プログラムと最適化オプションを処理する。最適化オプションはユーザがしたい最適化または最適化の組み合わせである。たとえば，`rule-pre/rule-cp/rule-dce` は「`rule-pre`(部分冗長性除去)，`rule-cp`(コピー伝播)，`rule-dce`(無用命令除去)を順番に適用せよ」の命令である。

入力プログラムがJavaコードの場合，3番地コードJimpleに変換し，制御フローモデルを作成する(4.1.2節を参照)。

最適化オプションに応じたCTL式を変換されたプログラムの変数や式に束縛する(4.2.2節を参照)。

### 7.2 モデル検査

各束縛したCTLに対して，モデル検査器が動作し，書換えに必要な情報を覚えておく(5章を参照)。モデル検査が終わったら，その結果を書換え部に渡す。

### 7.3 書換え部

モデル検査の結果に応じて変換処理を行う。書換え処理が必要なとき，書換え処理をし，ユーザが指定した中間言語を出力する。書換え処理の内容は6.2.3節を参照。

### 7.4 最適化器の計算量

最適化器の計算量は前処理部の計算量(4.2.3節を参照)とモデル検査の計算量(5.2節を参照)に比例する。つまり， $n_{optimizer} = O(n_{freevarBind} \times n_{modelCheck})$  である。

$$n_{optimizer} = O(n_1^{n_2} \times n_1 \times n_3) = O(n_1^{n_2+1} \times n_3)$$

$n_1$  : プログラムの大きさ

$n_2$  : 自由変数の数

$n_3$  : CTL 構文木のノード数

## 第8章 実装

本章は実装について説明する。

実装の説明にあたって図 5.1(左) のプログラムと 6.4.3 節の最適化記述を例とする。

### 8.1 Soot と Jimple

Soot[12] は Java 最適化フレームワークであり，新しい最適化処理の開発テスト環境として使用される．追加された新しい最適化処理は，Soot があらかじめ持っている最適化処理 (のうちのユーザが指定したもの) に加えて実行され，単一のクラスファイルあるいはアプリケーション全体に対して適用できる．Java コードは，Soot 内部でさまざまな中間表現を經由して処理が行なわれる．Soot は，入力されるコードの形式として Java クラスファイルあるいは Jimple 中間表現を受け付け，出力として Soot の中間表現のうち任意のものを出力することができる．

Java 仮想機械はスタックベースであるため，Java バイトコードもスタックに対する処理になっている．数ある Soot の中間表現のうちの一つの Jimple は，型付けされた 3 番地表現であり，コード変換処理に適した形式をしている．Soot はバイトコードを解析が容易な 3 番地形式に変換し，型付けを行ない，Jimple 中間表現を得る．たとえば Java の Sum クラスのメソッド `sum` を次のように定義し，

```
int sum(int n)
{

int r = 0;
if(n < 0)
return -1;
while(n >= 0)
r += n--;
return r;
}
```

これをコンパイルすると，以下のバイトコードを得る．

```
Method int Sum(int)
0:  iconst_0
1:  istore_2
2:  iload_1
```

```
3 : ifge 8
6 : iconst_m1
7 : ireturn
8 : iload_1
9 : iflt 22
12 : iload_2
13 : iload_1
14 : iinc 1 -1
17 : iadd
18 : istore_2
19 : goto 8
22 : iload_2
23 : ireturn
```

バイトコードを解析や変換の対象として扱うのは困難である．これに対して3番地形式である Jimple は以下のようなになる．

```
int sum(int)
{
    Sum r0;
    int i0 , i1 , $i2;
    r0 := @this : Sum;
    i0 := @parameter0 : int;
    i1 = 0;
    if i0 >= 0 goto label0;
    return -1;
    label0 :
    if i0 < 0 goto label1;
    $i2 = i0;
    i0 = i0 + -1;
    i1 = i1 + $i2;
    goto label0;
    label1 :
    return i1;
}
```

Jimple 形式で表現されていると，バイトコードそのままの形式と比較して，制御フローグラフを得る処理やデータフロー解析が非常に容易になる．

## 8.2 プログラム変換

本研究では Jimple 上で最適化を行うため，Java 言語を最適化の対象とするときは，Java プログラムを Soot によって Jimple3 番地コードに変換する．Jimple コードを最適化の対象とするときは，変換しない．

出力プログラムをユーザが指定した Soot がサポートする形式で出力もできる．このような入力，出力及び変換はすべて Soot 上で行う．

## 8.3 システムの構造

本研究で作成したシステムは，最適化オプション，及び最適化の対象プログラムを入力し，最適化処理を経て，中間プログラムあるいはクラスファイルを出力とする．ユーザが開発した最適化仕様は一つのフェーズとして，変換に加えられる．

システムの略図を図 8.1 に示す．システムの詳細図を図 8.2 に示す．

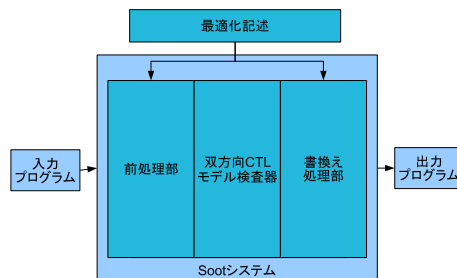


図 8.1: システム略図

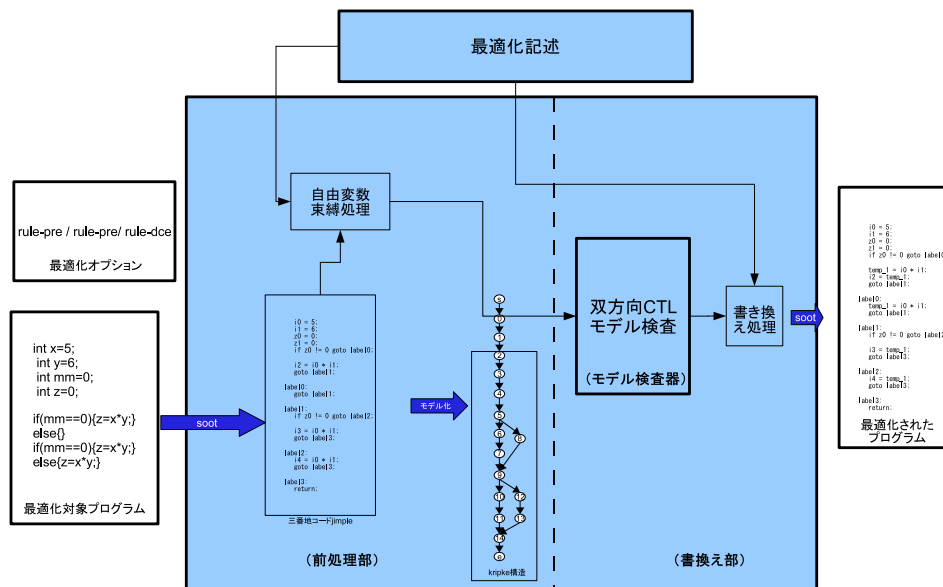


図 8.2: システム詳細図

## 8.4 処理の擬似コード

```
begin /*work*/  
  
  begin /*prepare(入力を読み込み, 下準備する)*/  
    read 最適化対象となるプログラム  
      入力をモデル検査に適切な形に変換する  
      入力プログラムの制御フローモデルを得る  
      危険辺を除去する  
    read 最適化オプション  
  end /*prepare*/  
  
  for each method  
    束縛に必要な情報を収集する  
    for each rule file  
      for each 命令文 in method  
        命令文をマッチする  
        for each 束縛  
  
          begin /*model check(CTL 構文木を bottom-up)*/  
            if  $labeled(i, s_i) = true$  が条件式である  
               $i$  を条件式と同じ名前の処理式の対象集合に入れる  
              書換え  $flag = true$   
            end if  
          end /*model check*/  
  
          if(書換え  $flag == true$ )  
            break(処理できる束縛があったら, 書換え処理へ)  
          end if  
        end for  
  
        begin /*process(書換え処理)*/  
          for each 処理式  
            case:命令文の処理式  
              for each 対象集合の要素  
                書き換え処理する  
              end for  
            case:辺の処理式  
              辺を計算する  
              for each 対象集合の要素  
                書き換え処理する  
              end for  
            end for  
          end for  
        end /*process*/  
  
      end for  
    end for  
  end for  
  
end /*work*/
```

## 8.5 処理の詳細

この節は処理の詳細について説明する。

入力から出力までの処理は下記のように行われる。

まず最適化オプション及び最適化の対象になるプログラムをを読み込み，入力をモデル検査に適切な形に変換し，危険辺を除去する。また，モデル検査に必要な情報を収集し，自由変数の束縛のために下準備をする。

ユーザが最適化のオプションで最適化または最適化の組み合わせを指定する。

入力プログラムはCTL式と束縛によって関係付けられる。

それから各束縛に対して，モデル検査を行う，各条件式を満たしたノード番号を処理式の対象集合に入れる。

書換え処理部は対象集合の要素を処理式に書かれた内容に従って最適化処理を行う。

最適化処理済みの出力プログラムをユーザが指定した `soot` がサポートする形式で出力する。

以下は入力から出力まで処理の流れの主な部分を説明する。

### 8.5.1 入力

入力はプログラムと最適化オプションを読み込み，下準備をする。入力プログラムはJavaの場合は，3番地コード `Jimple` に変換する。最適化オプションは `rule-pre/rule-cp/rule-dce`(部分冗長性除去/コピー伝播/無用命令除去) というふうを書く。

### 8.5.2 ルールファイル

ルールファイル (`rulefile`) は最適化記述 (6章を参照) のファイルである。

ひとつの最適化手法はひとつのルールファイルに書かれている。最適化記述を適用するときは，入力オプションで指定する。多数の最適化も指定できる。たとえば `rule-pre/rule-cp/rule-dce` は `rule-pre`(部分冗長性除去)，`rule-cp`(コピー伝播)，`rule-dce`(無用命令除去) を順番に適用するオプションである。

新しい最適化記述を開発するとき，記述の文法を沿ったルールファイルをプロジェクトの `rule` フォルダに入れることによってシステムに組み込む。

### 8.5.3 危険辺除去

任意の制御フローグラフでは，コードの移動がその過程で危険辺 (`critical edge`) によって阻まれることがある。危険辺とは後続ノードを複数持つノードから先行ノードを複数持つノードへの辺のことである。計算の挿入をしようとする時，他のパスに影響を与えてしまう。そこであらかじめ，危険辺の除去 (`edge splitting`) をしておくことで，この問題を回避する。

#### 8.5.4 プログラム情報を収集する

CTL 式の自由変数と束縛するため、プログラムにある単項式、二項式、変数、定数... など、モデル検査に必要な情報を収集し、保存する。たとえば、

r(単項式) :  $\{x, -y, \dots\}$

b(二項式) :  $\{x + y, a + b, a + 2, \dots\}$

v(変数) :  $\{x, y, z, \dots\}$

c(定数) :  $\{0, 22, 100, \dots\}$

...

#### 8.5.5 自由変数の束縛

自由変数の束縛は 4.2.2 節を参照する。

#### 8.5.6 モデル検査

モデル検査器のアルゴリズムは 5 章を参照されたい。実装上で下記の点に注意する必要がある。

- 3.3.4 節の変換を用いればすべての演算子を  $\neg$ ,  $\wedge$ ,  $\vee$ ,  $\text{EU}$ ,  $\text{EX}$ ,  $\overline{\text{EU}}$ ,  $\overline{\text{EX}}$  のみを用いるものに変換することができる。しかし変換を行うと式が一般的に長くなって、効率が悪くなるため、演算子の  $\text{AU}$ ,  $\text{AF}$ ,  $\text{AG} \dots$  を  $\neg$ ,  $\wedge$ ,  $\vee$ ,  $\text{EU}$ ,  $\text{EX}$ ,  $\overline{\text{EU}}$ ,  $\overline{\text{EX}}$  に変換せず、 $\text{AU}$ ,  $\text{AF}$ ,  $\text{AG} \dots$  をそれぞれの手続きによって実装した。
- 実装のしかたは一通りではないが、効率よい実装を心かけた。  
たとえば  $\text{EX}(\phi)$  の実装には
  - $\phi$  が true になったノードの先行ノードを true とする。
  - ノードの後続ノードを順番に調べ、 $\phi$  が true の後続ノードがあれば、値を true とする。

の 2 つの方法がありうるが、前の方が無駄な検証がないため効率がいい。実装の時はそういうことを考えて、より効率のよいモデル検査器になるようにした。

- $\text{AU}$  などの演算 (5.1.2 節を参照) は

$$\text{label}(n, s_i) = \prod \text{label}(n, s_j), (i, j) \in R$$

という定義なので後続ノードの label 値の積となる。後続ノードの  $\text{label}(n, s_j)$  に false なものがあれば、それ以降の後続ノードの label 値を調べなくても  $\text{label}(n, s_i) = \text{false}$  になる。しかし毎回  $\text{label}(n, s_j)$  が false かどうかをチェックするのは計算しないといけない。実際のプログラムの場合は先行ノードも後続

ノードもたくさんあるわけではないので，毎回 false かどうかチェックするより，全計算のほうが早い．実験の結果は全計算の方が約 5% 位早い．

...

### 8.5.7 辺の計算

最適化処理は辺を処理する必要もある．本研究は辺の計算はノードの結果によって計算できる．

例：

$$point_1 = \{6, 10, 12\}$$

$$point_2 = \{7, 13, 14\}$$

$$edge_1 = point_1 \mapsto point_2$$

のとき，実際の Kripke 構造が  $\{(3, 4) \dots (6, 7), (10, 14) \dots\}$

だとすると，

$$edge_1 = \{(6, 7), (10, 14)\}$$

となる．

### 8.5.8 書換え部

書換え部には条件式を満たした場所をどう処理するかが書かれている，書換え処理するとき，処理式を順番に適用する．適用の順序が不適切だと最適化の結果が合わなくなってしまう可能性がある．新しい最適化記述を開発するとき，処理式を適切な順序に書かないといけない．

#### 命令文の全体の処理

命令文全体に対して削除や挿入などを行う．たとえば， $\{b \mapsto x + y, v \mapsto z\}$  に束縛したとする．

$$point\_insert : \text{InsertBefore } temp := b$$

という記述は，命令文の前に式  $temp := x + y$  を挿入する．

$$point\_delete : \text{delete } v := b$$

とい記述は，命令文  $z := x + y$  を丸ごと削除する．

#### 命令文の一部の処理

命令文の一部の処理は 2 つの機能がある．

- 命令文の一部の処理はコピー伝播の範囲を広げた．  
たとえば， $\{v1 \mapsto x, v2 \mapsto y, v3 \mapsto z\}$  に束縛したとする．

$$point\_replace : \text{replace } v2 \mapsto v3$$

という記述に対して，命令文  $z := x + y$  は  $z := x + z$  に書き換えられる．

- 異なる命令文の対して同じ書換え処理したいときは便利である。

たとえば， $\{b \mapsto x + y\}$  に束縛したとする。

*point\_replace* : *replace*  $b \rightarrow temp$

という記述に対して，式の中の  $b$  と束縛した部分だけを  $temp$  に書き換え，命令文  $z := x + y$  は  $z := temp$  に， $w := x + y$  は  $w := temp$  に書き換えられる。

山岡らと番らの研究では命令文を丸ごとを書き換えることしかできないため，コピー伝播に関しては，図 8.3(左) の場合しか処理できない。命令文の一部を処理することによって，図 8.3 の左と右両方処理できるため，コピー伝播の範囲を広げた。

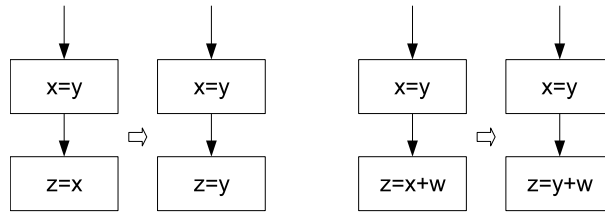


図 8.3: コピー伝播

## 辺の処理

*edge\_split* : *EdgeSplit*  $temp := b$

は辺に命令文  $temp := b$  を挿入する。

辺の処理は *EdgeSplit* しかできない。

本実装ではあらかじめ危険辺除去をすることにしたため，すべての辺について，ノード A がノード B を指しているとするとき，ノード B の先行ノードが 1 つしかないか，ノード A の後続ノードが 1 つしかないという条件のどちらかを満たす。

B の先行ノードが 1 つしかないならば，B の前に挿入する。B の先行ノードが多数個ある場合，A の後に挿入する。

## 一時変数の処理

実際の最適化では部分冗長性除去など一時変数を入れたい場合がある。本研究ではまた，実際の最適化に欠かせない一時変数の処理や辺の処理を加えた。CTL 式の一時変数  $temp$  を実際にプログラムに入れる時は，その場所に合った型をつける必要がある。型の付け方はその式に対応する。

$temp := b$  の場合は  $b$  の型と同じ型を付ける。

$b \rightarrow temp$  の場合は  $b$  の型と同じ型を付ける。

$v := temp$  の場合は  $v$  の型と同じ型を付ける。

また，部分冗長性除去を行う際には，その度に違う一時変数をいれないといけないため，入れる度に  $temp1, temp2, \dots$  のように添え字をつける。

### 8.5.9 その他実用上必要ないくつかの処理

本研究では実用上必要な処理をいくつか実装した。

- 「収束するまで回す」アルゴリズムの解を求められるようにした  
 従来の CTL を用いた最適化の研究では，一回検査するだけでは最適化の結果が不完全である可能性がある，本研究では，その不完全性をなくした．たとえば図 8.4 のように，代入文  $x := y$  は  $x$  が  $z := x$  によって使われるため，無用命令文ではない．しかし， $z := x$  が無用命令文として除去されたら， $x$  の使用も消えるため， $x := y$  は無用命令文として削除できる．このように最適化の結果が収束するまで計算できるようにした．

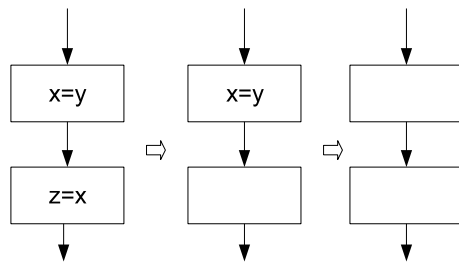


図 8.4: 無用命令除去

- シンプルな記号を導入した  
 過去時相演算子は数学的記法では「 $\leftarrow$ 」または「 $-1$ 」などで表現したが，実際に使う時，簡単に式を書けるようにシンプルな記号を導入した．

$$\begin{aligned} \overleftarrow{A}U &\Rightarrow AS \\ \overleftarrow{A}X &\Rightarrow AY \\ \overleftarrow{A}F &\Rightarrow AR \\ \overleftarrow{A}G &\Rightarrow AM \\ \overleftarrow{E}U &\Rightarrow ES \end{aligned}$$

...

したがって双方向 CTL 構文は下記の通りである．

双方向 CTL 式

$$\begin{aligned} \text{双方向 CTL } \ni \quad & ::= ap \mid \neg \mid \_1 \_2 \mid \_1 \_2 \\ & \mid EX \mid AX \mid E \_1 U \_2 \mid A \_1 U \_2 \\ & \mid EF \mid AF \mid EG \mid AG \\ & \mid EY \mid AY \mid E \_1 S \_2 \mid A \_1 S \_2 \\ & \mid ER \mid AR \mid EM \mid AM \end{aligned}$$

- 複数の最適化を自動化した  
 たとえば，実行オプション `rule-pre/rule-cp/rule-dce/rule-hli` をつける事により，自動的に `rule-pre`，`rule-cp`，`rule-dce`，`rule-hli` を順番に適用する．

## 8.5.10 最適化処理の例

図8.5(左)はプログラムの例,図8.5(中)はJimpleに変換したプログラム,図8.5(右)は部分冗長性除去の最適化記述(6.4.3節を参照)に従ってモデル検査をした結果である。

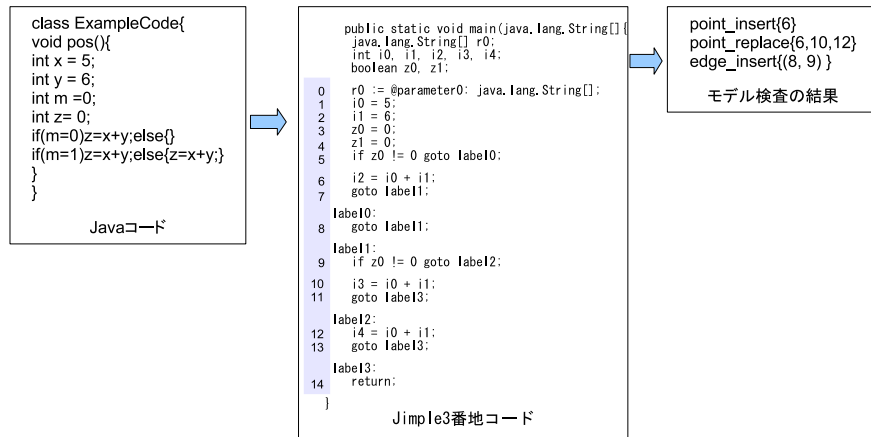


図 8.5: モデル検査の結果の例

図8.6はプログラムの流れ図によって表現した最適化前(左)と最適化後(右)のプログラムである。書き換え処理の内容は

- ノード6の前に一時変数 `temp1` を含む命令文 `temp1 := i0 + i1;` を挿入した;
- 辺  $\{8 \rightarrow 9\}$  に一時変数 `temp1` を含む命令文 `temp1 := i0 + i1` を挿入した;
- ノード6: `i2 = i0 + i1`, 10: `i3 = i0 + i1` と 12: `i4 = i0 + i1` の右辺 `i0 + i1` を一時変数 `temp1` に書き換えた;

である。

図8.7は図8.5(中)のプログラムが部分冗長性除去された後のプログラムである。

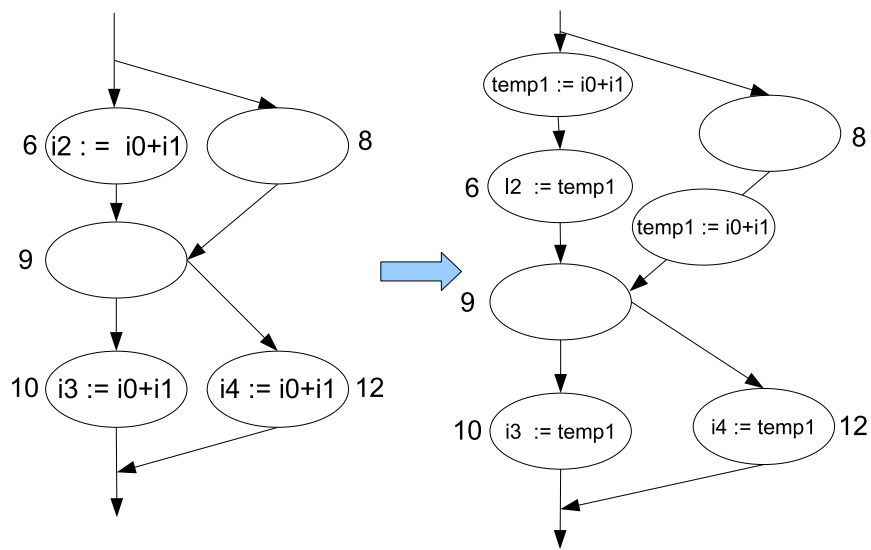


図 8.6: 書換えの例

```

java.lang.String[] r0;
int i0, i1, i2, i3, i4, temp1;
boolean z0, z1;

r0 := @parameter0: java.lang.String[];
i0 = 5;
i1 = 6;
z0 = 0;
z1 = 0;
if z0 != 0 goto label0;

temp1 = i0+i1;
i2 = temp1;
goto label1;

label0:
temp1 = i0+i1;
goto label1;

label1:
if z0 != 0 goto label2;

i3 = temp1;
goto label3;

label2:
i4 = temp1;
goto label3;

label3:
return;

```

部分冗長性除去した後のJimple3番地コード

図 8.7: 図 8.5 のプログラム書換え後の例

## 第9章 実験と考察

本研究では SPECjvm98 の中の 7 つのベンチマークと奥村らの Java コードを使って、実験を行い、データを取得した。

### 9.1 実験環境

実験環境：

マシンのスペック

CPU：celeron 2G Hz

Memory：512MB

Soot：version 2.2.0

計測のオプション

-Xint -Xms128m -Xmx128m を使って、JIT とメモリの影響を取り除いた。

### 9.2 ベンチマークの説明

SPECjvm98 [16] については表 9.1 を参照。奥村らの Java コード (正しくは [11] のサポートページにあるコードであるが、以下のように略称する) については表 9.2 を参照。

番号	名称	機能	ソース行数
200	check	checks JVM and Java features	5145
201	compress	A popular utility used to compress/uncompress files	5050
202	jess	a Java expert system shell	26,674
209	db	A small data management program	5,316
213	Javac	the Java compiler, compiling	57,937
227	mtrt	a dual-threaded program that ray traces an image file	9,713
228	jack	a parser generator with lexical analysis	21,895

表 9.1: SPECjvm98 ベンチマークの説明

奥村らの PiByMachin については、while と for 文を if 文と do-while 文に変換してから最適化した。こうした理由は、for 文は判断条件によって実行がループ内に入らない可能性がある。そうすると本研究で採用した部分冗長性除去によってループ不変式をループの外へ移動することができない。while と for 文を if(ループ条件判断式)

名称	機能	行数	備考
PiByMachin	円周率 (Machin の公式)	108	for 文を do while 文に改造した
CubeRoot	Newton 法で立方根を求める	224	入力 100000000.99
Cardano	3 次方程式	242	入力 10 20 30 40
CountingSort	分布数えソート	175	
NQueens	N 王妃の問題	247	入力 6
Jacobi	Jacobi (ヤコビ) 法	2061	入力 100
LogE	自然対数 $\log(x)$ を求める	1496	
Fibonacci	Fibonacci (フィボナッチ) 数列	206	
Exp	指数を求める	1045	

表 9.2: 奥村らの Java コードの説明

と do-while 文に直すことによって、ループ不変式をループの外へ移動することができるため、最適化の効果が高められる。

### 9.3 実験の結果

実験の結果は下記の通りである。適用した最適化は、部分冗長性除去、コピー伝播、無用命令除去であるが、部分冗長性除去は共通部分式除去とループ不変式のループ前移動を含んでいるので、コンパイラにおける典型的な最適化をほぼカバーしていると考えられる。

#### 9.3.1 本研究の手法による最適化の処理時間

SPECjvm98 ベンチマークの本研究の手法による最適化時間を表 9.3 に示す。部分冗長性除去、コピー伝播、無用命令除去を順に適用した時間である。

benchmark num	200	201	202	209	213	227	228
compile time	20	29	123	15	219	160	316

表 9.3: SPECjvm98 ベンチマークの本研究の手法による最適化時間 (単位: 秒)

奥村らのコードは小さいため、最適化時間はいずれも数秒から数十秒程度である。奥村らのコードの本研究の手法による最適化時間を表 9.4 に示す。

通常のコンパイラの最適化器ではミリ秒から秒単位で最適化できると考えられるが、本研究は数秒から数分までかかるのが遅い。しかし、時相論理による最適化すると全探索などの処理によって遅くなるのは仕方ないと思われる。また、本研究は番らの研究と比べると、最適化の時間ははるかに速くなっている。Lacey らの研究はデータがないため、比べるできないが、本研究は  $\mu$  計算への変換を省いたため、速くなると予測できる。

testcode	compile time
PiByMachin	0.8
CubeRoot	1.5
Cardano	7.1
CountingSort	2.5
NQueens	3.7
Jacobi	48.6
LogE	20.4
Fibonacci	1.4
Exp	12.1

表 9.4: 奥村らのコードの本研究の手法による最適化時間 (単位: 秒)

### 9.3.2 最適化前と最適化後の実行時間の比較

最適化前後の実行時間の比較を図 9.1, 9.2 に示す (最適化なしを 1 に正規化した)。

SPECjvm98 では 202\_jess のベンチマークが本手法によって 10%以上早くなった。209\_db, 213\_javac も 2%くらい効果が出ている。ほかは殆ど変わらない。

この図には, soot の最適化オプションをつけて, 通常のアプローチにより最適化した結果も載せてあるが [15], 202\_jess 以外は殆ど効果がない。本手法は通常のアプローチよりややよい結果を得た。

一方, 奥村らのコードの本研究の手法による最適化は約 3 分の 1 が最適化によって実行時間が数%から 20%くらい速くなった。

最適化の効果に影響する原因については 9.4.1 節で考察する。

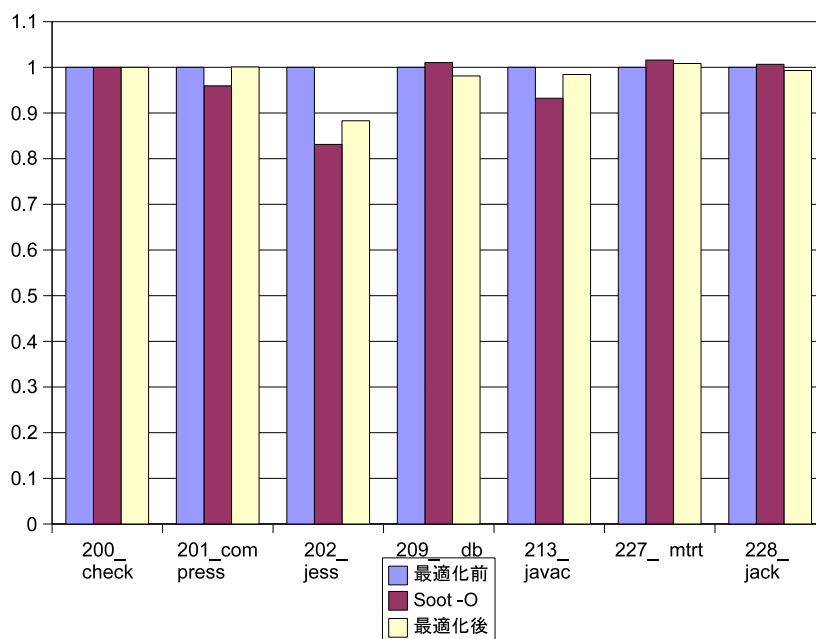


図 9.1: SPECjvm98 ベンチマークの最適化効果 (目的コードの実行時間, 最適化なしを 1 に正規化)

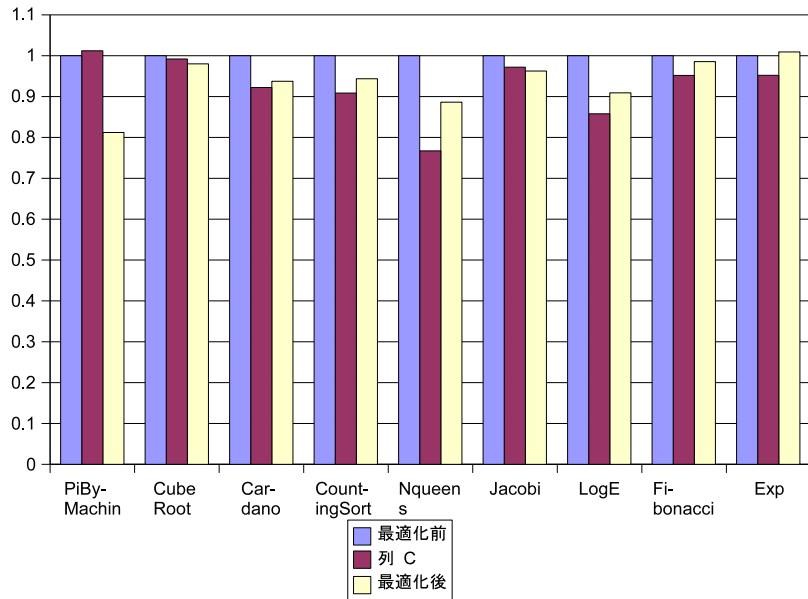


図 9.2: 奥村らの Java コードの最適化効果 (目的コードの実行時間, 最適化なしを 1 に正規化)

### 9.3.3 Lacey らの手法との比較

Lacey らは最適化時間や実行時間のデータを与えていない。

### 9.3.4 番らの手法との比較

番らの手法 [1] は過去時制を除去することによってコピー伝播ができるようになったのが特徴であるため, コピー伝播を適用した最適化時間を比較した。将来時制のみからなる最適化だと本研究と同じだが, 過去時制を含んだコピー伝播の記述により最適化を行う時間の比較を図 9.3 に示す (番らの手法による最適化を 1 に正規化した)。

番らの手法と比べると, 本研究の最適化時間は 15% から 30 と大変短くなっている。

### 9.3.5 同じ最適化を異なる CTL 式で記述したときの最適化時間の比較

図 9.4 に示したのはコピー伝播を例とし, 異なる CTL 式を用いて, 自由変数が 2 個 (左) から 4 個 (右) に増えたとき, 計算時間の爆発 (最適化時間が 20 秒から 59 分に変わった) が起こる例である。

この結果から自由変数の数がどの程度最適化の時間に影響するかが明らかになる。

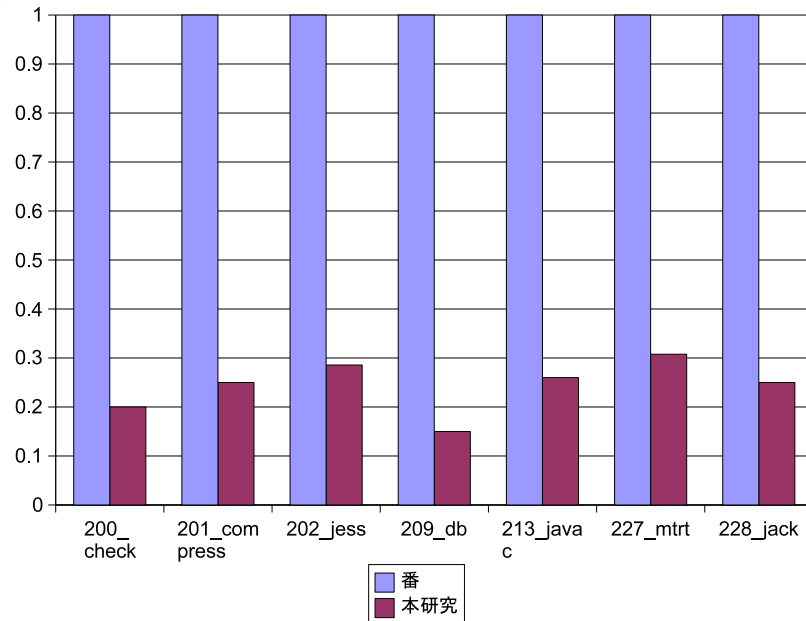


図 9.3: 番らの手法と本研究の最適化時間の比較 (コピー伝播の例．番らの方法を 1 に正規化)

自由変数の数	最適化時間 (秒)
2	20
4	3540

表 9.5: 自由変数による解析時間の差の例

CTL による最適化時間は自由変数の増加により爆発するため、CTL 式を定式化するとき、自由変数を減らすか無くす努力はもっとも大切な工夫である。

自由変数が 2 つの時も、マーチの処理で対象文と一対一に関係を付けられる (6.2.1 節を参照)。CTL-FV は最適化時間を爆発させるため、実用上では使えない。

## 9.4 実験データの分析

以下に、実験データの分析、問題点、考察を述べる。

### 9.4.1 分析と考察

以下、現在のシステムについて種々考察するが、本研究は、双方向 CTL 式による記述からモデル検査器と Java コンパイラの最適化器を実装し、その可能性と問題点を明らかにすることが主目的であるので、考察項目が多いことは決して悪いことではないことに留意してほしい。

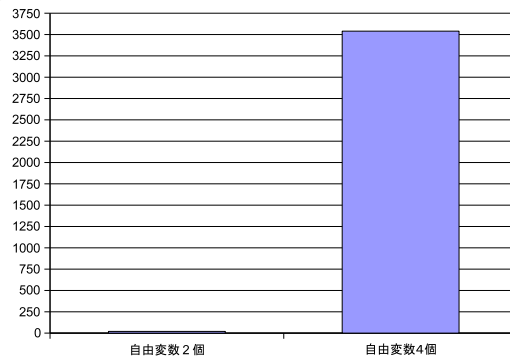


図 9.4: 同じ最適化を異なる CTL 式で記述したときの最適化時間の比較 (縦軸: 秒)

### CTL の表現力

CTL による最適化は簡単に数行で最適化記述を書け、簡潔であるが、通常最適化アルゴリズムより表現力が劣っている点がある。

細かい処理を CTL 式で書こうとすると、式が長くなるし、式の正しさの証明も難しい。

たとえば、「部分冗長性除去を行うとき、計算をあまり通らないパスからよく通るパスに移動したら、負の効果が生じる。そのときは最適化をしない」とか、「コピー伝播はもとの命令が無用命令除去により消されるとき行う、消されないときは行わない」などを CTL 式で書くのは困難である。

また、複雑なアルゴリズムを用いた最適化を書くことはできない。たとえば、条件付定数伝播 [18] は、データフロー方程式では定式化できず、表を用いた複雑なアルゴリズムによる最適化である。このような最適化は「モデル検査の結果はすぐ適用できない。途中結果として覚えておく必要がある、ある状態に至ったら、適用するか適用しないかを定める」のように書かないといけませんが、CTL ではできない。

### 最適化器の効率

CTL-FV に基づいた最適化器では自由変数の束縛はプログラム変数集合と CTL 変数集合の全組み合わせになる (??節を参照)。そのうえ、モデル検査は全探索によって行っている。そのため、効率は通常最適化器より遅い。

### 最適化器の効果

本研究は Java を対象としているため、命令文の移動は例外を超えることはできない。配列、割り算、余算など実行時例外を起こし得るものも全部対象外になるといった制限がある [10]。

Soot では、BriefGraph というプログラムの制御フローグラフを表すものがある。一方、CompleteGraph というグラフもある。これは制御フローグラフの上に try 文に囲まれたすべての命令文から catch 文に辺を引いたグラフである、図 9.5(左)の

プログラムの部分冗長性除去を BriefGraph 上で行った場合の例を図 9.5(中) に示す . CompleteGraph 上で行った場合の例を図 9.5(右) に示す .  $x + y$  の移動は太線のパスに影響するため , 保守的な部分冗長性除去のアルゴリズムでは移動を諦めることになる .

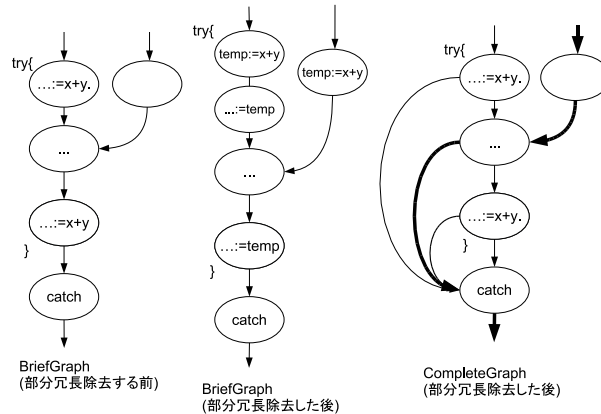


図 9.5: 例外による最適化の阻害

コード移動が例外を超えられない問題は , 本研究だけでなく , 通常の Java 最適化器にも存在する .

## 9.4.2 今後の課題

今後の課題は 2 つの方向がある .

### 最適化時間の短縮

モデル検査器を早くするために , BDD を導入したり , 部分評価などの手法で , モデル検査器の全探索を避けることが考えられる .

自由変数の束縛はもっとも最適化時間に影響するため , 自由変数の数と自由変数の束縛を減らせれば , 最適化時間の減少に一番効果的である .

束縛の数を改善できる例は下記のようなものである . 次のプログラムを例とする .

```

1:  $x := 100;$ 
2:  $y := 1;$ 
3:  $z := 2;$ 
4:  $w := 3;$ 
5:  $x := w + 1;$ 
6:  $z := x + y;$ 
...

```

- ほかの束縛によって束縛しなくて良い束縛をなくす．たとえば，6:  $z := x + y$  の  $x$  をコピー伝播の対象とするとき，この  $x$  は 5:  $x := w + 1$  によって代入されているから，さらに前の文 1:  $x := 100$  と束縛をする必要はない．
- プログラムにない式は束縛しなくてよい．たとえば  $AX(stmt(v = c))$  に対して  $\{v \mapsto z, c \mapsto 3\}$  も一つの束縛であるが， $z := 3$  という文はプログラムにないため，このような束縛はしなくてよい．
- 時相経路上にない式と束縛しなくてもよい．過去時制のみの双方向 CTL 式は過去向きの経路上にある変数だけに束縛し，将来時制のみの双方向 CTL 式は将来向きの経路上にある変数だけに束縛する．また  $AX$  や  $EX$  は次の文だけに束縛する，さらに遠い命令文と束縛しない．  
この手法を無用命令除去 (順方向のみ) の最適化に適用した実験の結果，最適化の時間はおよそ 3 分の 1 になった．今回の研究では時間の関係で完成していない．
- MATCH 文より賢い束縛方法があるかもしれない．  
...

## 最適化の効果の向上

最適化の効果を高めるために，例外による最適化の阻害を克服することや，ループ，for 文，goto 文など細かい解析が必要である．

双方向 CTL による例外の分析も考えられる．

条件付定数伝播など複雑な最適化をどう書くのかも将来課題である．

双方向  $CTL^*$  という理論体系がある，これを実装する事によって，もっと強力な表現力をもつ論理式を書ける．こうすると，正確性を保ちやすく，もっと複雑な最適化を書けるようになる．しかし一方モデル検査器の実装は難しくなる可能性がある．これも時間の関係で試していない．

# 第10章 関連研究

ここでは、関連研究について述べる。関連研究は各々の提案があるが、それぞれ欠点もある。また、いずれも抽象的な言語あるいは小さな言語しか対象としていない。モデル検査による解析時間と最適化の効果について、実用上具体的なデータが提示されているものもない。

## 10.1 Lacey らの研究

従来の研究として、Lacey らの研究 [3] は、時相論理 CTL に対して過去の時制を扱う演算子を加え、自由変数を導入して拡張した時相論理 CTL-FV を提唱した。伝統的なプログラム最適化の多くは、CTL-FV による条件の記述と、その結果を用いた命令文の書換えで記述できることを示した。[2][4] の論文はその理論の提案と正当性を述べた。また、いくつかの式について最適化の正しさを証明した。[4] の論文はこの理論体系を使って最適化する手法の詳細を述べているが、実装については、 $\mu$  計算に変換することによって不動点解を求めるという簡単な説明しかない。また、いくつかの最適化について、定式化してあるが、典型的な例で扱えないものがある。下記の例が扱えないものとして挙げられる。

- いくつかの最適化について、定式化してあるが、典型的な所で扱えないものがある。

共通部分式除去の例

$$n : (a := e[b]) \implies (a := e[v])$$
$$if\ n \models \overline{A}(trans(b) \quad \neg\ def(v) \ U\ stmt(v := b))$$

これは、下記のケースを扱えない。

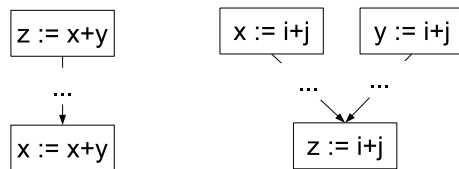


図 10.1: Lacey の式が扱えない例

Lazy code motion の例

...

MATCH

$n \models \text{partial\_avail}(e) \quad \overline{A}(\text{trans}(e) \cup \text{pp}(n, e) \quad \text{avail}(e))$

...

これは部分冗長を扱えるが、全冗長を扱えない。

- 定式化に命令文と対応した Kripke 構造のノード番号を書くことになっている，そのため束縛が爆発すると予測できる．ノード番号による計算回数増加は：*Kripke* 構造のノード数<sup>CTL 式に現れた命令文番号の数</sup>となる．Lacey の論文には小さな言語を対象とした説明しかなく，その範囲では最適化は数分で終ると述べられているが，実際の大きな言語を対象した場合，現実的には不可能な程の計算量になる．
- 部分冗長性除去など複数箇所を同時に書換えないといけない場合が処理しにくい．
- ...

## 10.2 山岡らの研究

山岡らの研究 [19] は，既存の優秀なモデル検査器 SMV [14] を使って，700 行くらいの行数のプログラムで CTL 式から最適化器を作成した．

述語は `def()` と `use()` しかないうえ，将来時制しか扱えないため，無用命令除去しか扱えない．

## 10.3 番らの研究

番らの論文 [1] は 12 個の変換式を使って，過去時制を含む NCTL 式を扱えるようにした．除去処理に時間がかかり，除去によって式が長くなる．その結果，モデル検査の時間がかかり長い．自由変数が多い場合は現実的ではないと予測される．また，最適化記述にひとつの条件しか書けない，命令文しか扱えない，などの欠点があるため，無用命令除去とコピー伝播しかできない．

## 10.4 伊藤らの研究

伊藤らの研究 [6] は命令文の依存関係を時相論理式により分析する．この研究は抽象化言語だけを対象としている，また，全探索による手法は現実的には困難である，

# 第11章 今後の課題

今後の課題は2つの方向がある。

## 11.1 最適化時間の短縮

モデル検査器を早くするために、BDDを導入したり、部分評価などの手法で、モデル検査器の全探索を避けることが考えられる。

自由変数の束縛はもっとも最適化時間に影響するため、自由変数の数と自由変数の束縛を減らせれば、最適化時間の減少に一番効果的である。

束縛の数を改善できる例は下記のようなものである。次のプログラムを例とする。

```
1: x := 100;
2: y := 1;
3: z := 2;
4: w := 3;
5: x := w + 1;
6: z := x + y;
...
```

- ほかの束縛によって束縛しなくて良い束縛をなくす。たとえば、 $6: z := x + y$  の  $x$  をコピー伝播の対象とするとき、この  $x$  は  $5: x := w + 1$  によって代入されているから、さらに前の文  $1: x := 100$  と束縛をする必要はない。
- プログラムにない式は束縛しなくてよい。たとえば  $AX(stmt(v = c))$  に対して  $\{v \mapsto z, c \mapsto 3\}$  も一つの束縛であるが、 $z := 3$  という文はプログラムにないため、このような束縛はしなくてよい。
- 時相経路上にない式と束縛しなくてもよい。過去時制のみの双方向 CTL 式は過去向きを経路上にある変数だけに束縛し、将来時制のみの双方向 CTL 式は将来向きを経路上にある変数だけに束縛する。また  $AX$  や  $EX$  は次の文だけに束縛する、さらに遠い命令文と束縛しない。

この手法を無用命令除去 (順方向のみ) の最適化に適用した実験の結果、最適化の時間はおよそ3分の1になった。今回の研究では時間の関係で完成していない。

- MATCH 文より賢い束縛方法があるかもしれない。

...

## 11.2 最適化の効果の向上

最適化の効果を向上するのは難しいが、以下のことが考えられる。

- Java を対象としたため、HIR で行われる最適化はあまり効果がない問題は本研究だけではなく、他の最適化器にも存在している。

最も最適化の効果を影響するのは例外の阻害であるため、それを克服できるような工夫は重要である。双方向 CTL による例外の分析も考えられる。

- ループ, for 文, goto 文など細かい解析ができると、最適化できる範囲も広がる。
- 条件付定数伝播など複雑な最適化をどう書くのかも将来課題である。
- *CTL\** という理論体系がある、これを実装する事によって、もっと強力な表現力をもつ論理式を書ける。こうすると、正確性を保ちやすく、もっと複雑な最適化を書けるようになる。しかし一方モデル検査器の実装は難しくなる可能性がある。これも時間の関係で試していない。

## 第12章 まとめ

本研究の主な貢献は次のとおりである。

- 双方向 CTL について，過去時制除去も  $\mu$  計算への変換もせずに直接処理できるモデル検査器を実装した。
- コンパイラの典型的な最適化の方式のいくつかを定式化し，実際の最適化処理に欠かせない処理をいくつか加え，従来研究の欠点を克服し，実用的な言語である Java 言語を対象として，双方向 CTL による実用的な Java 最適化器を開発した。
- 最適化の時間や最適化の効果について，ベンチマークやテストコードを使ってデータを取った。その結果，多くの最適化で，本研究の手法によってモデル検査すなわち最適化の効率が，従来研究に比べて大幅に改善し，実用的な時間内で最適化できるようになった。
- この経験を通じて，CTL による最適化の問題点をいくつか明らかにすることができた。また各種のデータを提示した。それらの経験とデータはこの分野の研究において重要な参考になるだろう。

本研究は CTL による実用的な最適化器の生成への初めての試みとして，Java 言語を対象とした実装を行うことでベンチマークでのデータを取り，種々の問題点をあきらかにした，これは将来の発展の足かかりとなる。

モデル検査による最適化時間については，大きなプログラムでも 5 分以内であるが，まだ実装上の問題などで大きく改善する余地がある (11 章を参照)。見込みとしては 1 分以内に抑えられる。

今後は，まだ解決していない問題点の克服に向けて努力し，CTL 式に基づく実用的な Java 最適化器を作成したい。また，予想としては，まだ解決できていない問題点 (9.4.1 節を参照) を克服できたとすれば，CTL 式に基づく実用的な Java 最適化器が実現でき，伝統的な最適化器の一部や最適化式の証明などに役に立つ。

# 謝辞

本研究を進めるにあたって，東京工業大学 大学院情報理工学研究科 数理・計算科学専攻教授の佐々政孝先生には多大な御指導，御鞭撻を頂きました．深く感謝の意を表します．

東京大学の卒業生の番伸宏さんからは，PCTL を NCTL に変換し，順方向 CTL のモデル検査を行うコードをコピーさせて頂いた．本研究はそのモデル検査器に大幅な改善を加えたものであるが，このコードのおかげで実装の手間が軽減された．深く感謝いたします．

また，研究に多くの助言を頂いた中谷俊晴さん，佐原聡一郎さん，また計算工学専攻の伊藤宗平さん，東京大学の胡振江先生にも心からお礼を申し上げます．

## 参考文献

- [1] 番 伸宏, 胡 振江, 一彦, 武市 正人. Java プログラム最適化の宣言的記述とその効率的な実装. 第6回プログラミングおよびプログラミング言語ワークショップ (PPL2004). pp.65-75, 2004.
- [2] David Lacey, Neil D. Jones, Eric Van Wyk, Carl Christian Frederiksen. Compiler Optimization Correctness by Temporal Logic. Higher-Order and Symbolic Computation, Vol.17, pp.173-206, 2004.
- [3] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In Proceedings of Symposium on Principles of Programming Languages, pp.283-294, 2002.
- [4] David Lacey. Program transformation using temporal logic specifications. Dissertation submitted for DPhil examination at the University of Oxford, 2003.
- [5] Francois Laroussinie and Philippe Schnoebelen. Specification in CTL+Past for verification in CTL. Information and Computation, Vol.156, No.1/2, pp.236-263, 2000.
- [6] 伊藤宗平, 萩原茂樹, 米崎直樹. 意味的制約の書換えによるコンパイラのコード最適化. 日本ソフトウェア科学会第22回大会(2005年度)論文集.
- [7] Jan Van Leeuwen(編). 広瀬ほか訳. コンピュータ基礎論理ハンドブック 形式的モデルと意味論. 丸善株式会社.
- [8] 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.
- [9] O. Kupferman, and A. Pnueli. Once and For All. In Proceedings of the 10th IEEE Symposium on Logic in Computer Science (LICS 1995), pages 25-35, 1995.
- [10] 大平 怜, 平木 敬. 例外依存関係を超える部分冗長性除去. 情報処理学会論文誌: プログラミング. Vol. 46, No. SIG1(PRO 24), 2005.
- [11] 奥村晴彦:Java によるアルゴリズム事典のソースコード. <http://oku.edu.mie-u.ac.jp/~okumura/Java-algo/>

- [12] Raja Vallee-Rai , Laurie Hendren , Vijay Sundaresan , Patrick Lam , Etienne Gagnon , and Phong Co . Soot-a Java optimization framework . In Proceedings of CASCON 1999 , pp.125-135 , 1999 . <http://www.sable.mcgill.ca/soot/>
- [13] 佐々 政孝 . プログラミング言語処理系 . 岩波書店 , 1989 .
- [14] SMV Model Checker .  
<http://www.cs.cmu.edu/modelcheck/smv.html>
- [15] Soot A Java Bytecode Optimization Framework .  
<http://www.cs.rutgers.edu/ryder/oosem99/talks/isaila-soot.pdf>
- [16] SPEC JVM98 Benchmarks .  
<http://www.spec.org/osg/jvm98>
- [17] Vineeth Kumar Paleri , Y.N.Srikant , Priti Shankar . A Simple Algorithm for Partial Redundancy Elimination . ACM SIGPLAN Not. , Vol. 33 , Issue 12 , pp.35-43 , 1998 .
- [18] Wegman , M.N. , and Zadeck , F.K , Constant propagation with conditional branches , ACM Trans. Prog. Lang. Syst. , Vol . 13 , No . 2 , pp . 181-210 , 1991.
- [19] 山岡裕司 , 胡振江 , 武市正人 , 小川瑞史 . モデル検査技術を利用したプログラム解析器の生成ツール . 情報処理学会論文誌 , Vol . 44 , No . SIG13(PRO18) , pp.25-37 , 2003 .

# Appendix

本研究で使う記号の説明：

$v$ :	変数
$c$ :	定数
$e$ :	式 (二項式, 単項式, 変数, 定数)
$b$ :	二項式
$r$ :	単項式
$temp$ :	一時変数
$point$ :	Kripke 構造のノード
$edge$ :	Kripke 構造の辺
$succs_i$ :	Kripke 構造の $i$ 番ノードの後続ノード
$preds_i$ :	Kripke 構造の $i$ 番ノードの先行ノード
$ap$ :	論理式の述語
$\equiv$ :	常に同等な値を持つ
$n \longrightarrow n'$ :	ノード $n$ からノード $n'$ へ指している辺
$x \rightarrow x'$ :	$x$ が $x'$ によって書き換えられる