双方向CTLによるJava最適化器の生成

方 玲 佐々 政孝
Fang Ling Sassa Masataka

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 Mathematical and Computing Sciences，Tokyo Institute of Technology
{fang3，sassa}@is.titech.ac.jp

本研究は分岐的時相論理の 1 つである双方向 CTL のモデル検査器を実装した．また，双方向 CTL によるコンパイラ最適化記述法を提案し，前記のモデル検査器を使って，典型的なコンパイラ最適化器を実現した．最適化記述において，書換え条件や一時変数などを扱う処理を加えることで，Java 言語の最適化を可能にした．従来，モデル検査器を用いた最適化器は実用的な時間では動作しないといわれていたが，本研究では 7 つの SPECjvm98 ベンチマークについて，1 つを除いて，数十秒から 2，3 分程度で動作する事を確認できた．また，生成される最適化器の処理時間を短くするための工夫や，記述のノウハウなど，本手法の改善に向けた種々の考察を加えた．

1 はじめに

コンパイラの設計において，コード最適化は重要なパスの 1 つであり，目的コードの時間的・空間的効率を向上させる役割を果たしている．

最適化器はプログラムを書きく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，次の条件付き書換え規則 (conditional rewrite rule) 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「$I \Longrightarrow I'$ if φ」の形で記述する．たとえば無用命令除去の例は下記である．

$x := e \Longrightarrow skip$
if $AX((AG \neg use(v)) \vee A \neg use(v)\ U\ def(v))$

本研究は分岐的時相論理の 1 つである双方向 CTL のモデル検査器を実装した．また，双方向 CTL によるコンパイラ最適化記述法を提案し，前記のモデル検査器を使って，典型的なコンパイラ最適化器を実現した．

コンパイラ最適化は将来時制と過去時制を用いると自然かつ簡潔に記述できる．Lacey らが提案した CTL-FV[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である．伝統的なプログラム最適化の仕様の多くは，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．

本研究は論理として CTL-FV を基本として採用した．しかし後述のように，Lacey らの実装は現実的ではないので，本研究では最適化の記述や実装に多くの工夫を行った．

過去時制を扱えるモデル検査器としては，他に番らの研究 [1] がある．番らの論文は 12 個の変換式を使って，過去時制を含む NCTL 式 [5] を未来時制のみを用いる CTL 式に変換することによって過去時制を扱えるようにした．しかし，この変換処理は時間がかかり，変換によって式が長くなるため，モデル検査の時間もかなり長い．また NCTL 式には制限があり，過去時制を自由に書くことができない．

本研究は双方向 CTL モデル検査器を直接に実装した．過去時制を将来時制と対称的に検査するため，変換にかかる処理時間を省け，除去によって式が長くなることがないので処理時間が短い．また，Lacey らは，一旦 μ 計算に変換することで実装しているが，我々の方法では直接実装していることも処理時間の短縮につながっていると思われる．このようにして従来研究の欠点を克服できた．

一方，最適化の記述能力については，番らと山岡らの研究は書換えの条件部分にひとつの条件しか書けず，それに対して一箇所しか書換えられないため，部分冗長性除去など，同時に複数の条件を用いて複数箇所を書換えたい時は手に負えない．また，辺の処理ができない．

モデル検査する前には自由変数を束縛しないといけないため，自由変数が多い場合は，処理時間が現実的ではなくなる．Lacey らの記述法は Kripke 構造のノード番号を用いるため自由変数が多く，記述の

自由度が高いが，Kripke 構造のノード番号の束縛に多くの時間がかかる．

本研究の記述は，Lacey らの方法と異なり Kripke 構造のノード番号を書かなくてよい．モデル検査器は条件式を満たす特定の番号の命令文ではなく，命令文の集合を計算する．そのため，同じ条件式を満たす多数の命令文を書換える処理が容易に記述できるようになった．また，Kripke 構造のノード番号を束縛することを省くことができ，効率を改善した．

さらに，最適化記述において書換え条件や一時変数などを扱う処理を加え，典型的な Java 言語コンパイラ最適化器を実現した．

従来，モデル検査器を用いた最適化器は実用的な時間では動作しないといわれていたが，本研究では以上で述べた手法を用いることで，7 つの SPECjvm98 ベンチマークについて，1 つを除いて，数十秒から 2，3 分程度で動作する最適化器が生成できる事を確認できた．

また，生成される最適化器の処理時間を短くするための工夫や，記述のノウハウなど，本手法の改善に向けた種々の考察を加えた．

2　双方向 CTL

双方向 CTL は時相論理の 1 つである．以下，双方向 CTL について説明する．

双方向 CTL とは　双方向 CTL の提案は [10] 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 を逆向きにしたものである．

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．

構文規則　双方向 CTL の構文規則は以下の通りである．

$$
\begin{aligned}
\text{双方向 } CTL \ni \phi \quad ::= &\ \alpha \mid \neg\phi \mid \phi_1 \wedge \phi_2 \\
&\mid EX\ \phi \mid E\ \phi_1 U\ \phi_2 \\
&\mid \overleftarrow{E}X\ \phi \mid \overleftarrow{E}\,\phi_1 U\ \phi_2
\end{aligned}
$$

他の結合子 $EF\ \phi$, $EG\ \phi$, $AF\ \phi$, $\overleftarrow{A}X\ \phi, \ldots$ 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．

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，$L : S \to 2^{Prop}$ 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 $\pi = (s_0, s_1, \ldots)$ である．ここで，有限のパス $(s_0, s_1, \ldots, s_m)$ 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，$\forall i \geq 0 : (s_{i+1}, s_i) \in R$ となるような極大の列である．

論理式 ϕ 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．

$s \models \alpha$ iff $\alpha \in L(s)$

$s \models \neg\phi$ iff $s \models \phi$ 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 $s' \models \phi$

$s \models \overleftarrow{E}X\ \phi$ iff $\exists s'$; $s'Rs$ かつ $s' \models \phi$

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

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

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 1 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．

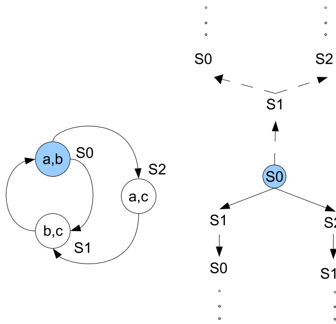

図 1: Kripke 構造とその構造に対応した双方向 CTL 無限木

3 制御フローモデル

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．

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

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

命令の BNF を以下に記す．

$I ::= skip \mid X := E \mid if\ X\ goto\ n\ else\ n \mid goto\ n$
$E ::= X \mid E\ O\ E$
$O ::= + \mid - \mid * \mid / \mid \ldots$
$X ::=$ 変数名
$n ::= 1 \mid 2 \mid 3 \mid \ldots \mid m$

本節では、Kripke 構造として制御フローモデルを定義する．

制御フローモデル コード π に対する制御フローモデルは三つ組み $M(\pi) = (Nodes_\pi, \rightarrow_\pi, L_\pi)$ として定義される．ここで $Nodes_\pi$ は文のラベルの集合であり，関係 → は次のように定義される関係である．$n_1 \rightarrow_\pi n_2$ iff

$$\begin{aligned}
&(I_{n1} \in X := E,\ skip,\ read\ X) \\
&\quad \wedge\ n_2 = n_1 + 1 \\
&\vee\ (I_{n_1} = goto\ n \wedge n_2 = n) \\
&\vee\ (I_{n_1} = if\ X\ goto\ n\ else\ n' \wedge (n_2 = n \vee n_2 = n')) \\
&\vee\ (I_{n_1} = write\ Y \wedge n_2 = n_1)
\end{aligned}$$

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

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

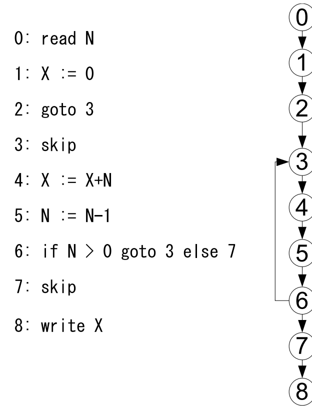

図 2: コードと制御フローモデルの例 (L(n) は略した)

図 2 のコードと制御フローモデルと対応した $L_\pi(n)$ は下記である．

$$\begin{aligned}
\text{L(0)} &= \{\text{stmt(read N), def(N), trans(N-1)}\ldots\} \\
\text{L(1)} &= \{\text{stmt(X:=0), def(X), trans(N)}\ldots\} \\
&\ldots \\
\text{L(8)} &= \{\text{stmt(write X), use(X)}\ldots\}
\end{aligned}$$

4 双方向 CTL モデル検査器

この節では，本研究で実装した双方向モデル検査器について述べる．

双方向モデル検査器は従来の研究のモデル検査器の拡張であり，将来時制の検査は既存のアルゴリズムと同じく，CTL 木の真理値を求める．過去時制の検査は将来時制を逆向きにし．$\overleftarrow{CTL}$木の真理値を求める．

双方向 CTL 式の解析 CTL 式は木構造で表せる．これを CTL 構文木と呼ぶ (CTL 木とは異なることに留意)．木の葉は原子述語である．図 3(左) の CTL 式の構文木を図 3(右) に示す．各部分式は表 1 を参照．

双方向 CTL のモデル検査 CTL 式に対する構文木の葉から根に向かって，対象としている部分式 ϕ_n 毎

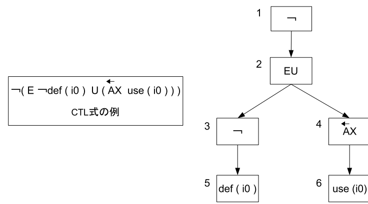

図 3: 双方向 CTL 構文木

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(use(i_0)))
2	EU	3 4	E ¬ def(i_0) U $\overleftarrow{A}$X(use(i_0))
3	¬	5	¬ def(i_0)
4	$\overleftarrow{A}$X	6	$\overleftarrow{A}$ X($use(i_0)$)
5	ap	def(i_0)	def(i_0)
6	ap	use(i_0)	use(i_0)

表 1: 図 3 の CTL 構文木とその部分式

に各状態 s で満たされるかどうかを計算する．すなわち状態 s における部分式 ϕ_n の真理値を $label(\phi_n, s_i)$ とするとき，そのラベルの真理値の計算を行う．

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

モデル検査の結果は後の書換え処理に使用されるため，モデル検査をしながら，書換え処理に必要な結果をデータ構造にしまう．このデータ構造は双方向 CTL 構文木のノードと対応した集合である．

モデル検査の計算量 プログラムの大きさを命令文の数 n_1 とし，CTL 式の大きさを CTL 構文木のノードの数 n_2 とする．モデル検査は CTL 構文木のすべてのノードですべての状態での真理値を計算するので，モデル検査器の計算量は積 $n = n_1 \times n_2$ と比例する．

モデル検査の計算量 $= O(n_1 \times n_2)$

5 最適化の記述

ここでは，本システムでの最適化の記述について説明する．

5.1 最適化記述の仕様

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とき満たすべき双方向 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，くわしくは [6] を参照．

MATCH
変数 := 式
CONDITION
*point_*文字列：CTL 式
*edge_*文字列：*point_*文字列 ⟶ *point_*文字列
PROCESS
*point_*文字列：$Comand$ 命令文
*point_*文字列：$Replace$ 式 → 式
*edge_*文字列：$EdgeSplit$ 命令文

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 $\{x, y, z, x + y\}$ の集合に一対一に関係付け，全組み合わせによる束縛を避けることができる．

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とき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．

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．

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換える処理が容易に記述できるようになった．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
$$point_delete: \; AX((AG \neg use(v)) \vee A \neg use(v) \, U \, def(v))$$
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．

$$n : v := e \Longrightarrow skip$$
$$\text{if } n \models AX((AG \neg use(v))) \vee A \neg use(v) \, U \, def(v))$$

これはが $\{v, e, n\}$ 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 (6 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．

5.2 最適化の双方向 CTL による定式化

この節ではコンパイラ最適化の定式化について述べる．本研究では，従来の定式化を拡張し，データフロー方程式をもとにした定式化も記述できるようになった．

5.2.1 最適化の条件をもとにした CTL 式を記す方法

従来研究の定式化手法と同じであるが，実際の言語の特徴や効率を考える必要がある．前述の無用命令除去はこの例である．

5.2.2 データフロー方程式をもとにした CTL 式を記す方法

部分冗長性除去は複雑な最適化である．多数の条件式が必要であり，同じ条件式を満たす多数の命令文を書換える処理をしないといけない．最適化の条件をもとにした定式化は難しい．

データフロー方程式をもとにした定式化は初めから考えるのに比べてはるかに容易である．

本研究はこの手法を使って，部分冗長性除去を容易に記述できた．

部分冗長性除去は長年にわたり研究され，多数のアルゴリズムがあるが，本研究は Paleri らの方法 [17] を採用して部分冗長性除去を定式化した．このアルゴリズムはシンプルで，変換の結果の計算回数の最適性が証明されている．

そのデータフロー方程式と対応した CTL 式を Appendix に示す．

一般に，データフロー方程式を解くときは，データフロー情報の値が決まる所からはじめて，$AVIN_0$，$AVOUT_0$，$AVIN_1$，$AVOUT_1$，$AVIN_2 \ldots$ のように収束するまで繰り返す方法で解いていく事ができるが，CTL 式によるモデル検査では同じように収束するまで繰り返すことができない．そこで，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した [6]．

6 自由変数の束縛とその計算量

ここでは，自由変数の束縛とその計算量について説明する．

自由変数は CTL 式の述語に表れて，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．プログラムを扱う際には，束縛によって実際のプログラムの変数と関係付ける．たとえば，

MATCH

$v := e$

のような記述では自由変数は $\{v, e\}$ である．仮にプログラムの変数や式が $\{x, y, z, a+b, x+y, -z\}$ であるとすると，自由変数の定義域は

$v \mapsto \{x, y, z \ldots\}$

$e \mapsto \{a+b, x+y, -z \ldots\}$

である．すなわち

$\{v ::= x, e ::= a+b\}$

$\{v ::= x, e ::= x+y\}$

$\{v ::= x, e ::= -z\}$

$\{v ::= y, e ::= a+b\}$

$\ldots$

のように束縛する．

自由変数の束縛は全探索のため，システムの処理時間つまり最適化時間に大きく影響する．

自由変数束縛の計算量

m：CTL 式の自由変数の束縛対象の数 (CTL 式に自由変数 v があった場合，v と束縛するプログラムの中の変数 x，y，z. . . の数)

n： CTL 式の自由変数の数

とする，すると

時間計算量：$O(m^n)$

となる．

7 双方向 CTL による最適化器

双方向 CTL による最適化器を作成した．これは前処理部，モデル検査器と書換え部の三部分から構成される．前処理部は入力をモデル検査に必要な形に直す処理をする．その結果を使ってモデル検査器がモデル検査を行う．書換え部で検査した結果に基づき最適化書換え規則を適用し，コードを出力する．

図 4 は本最適化システムの略図である．

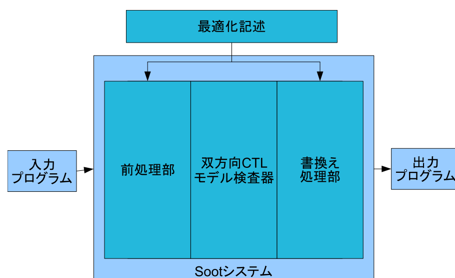

図 4: 最適化システム略図

システムは全体的に Soot[13] 上で動いている．Soot 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) に加えて実行される．

図 5 は，左のプログラムに対して，中央の記述から作られた右の CTL 構文木を用いてモデル検査する．CTL 構文木ではそれぞれの節に「名前付きの部分式」(対象集合と呼ぶ) の値を付加する．

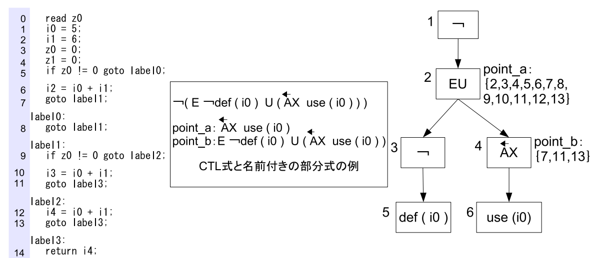

図 5: モデル検査処理

図 6 はモデル検査の結果を使って部分冗長性除去の書換え規則を適用する前と後の例である．

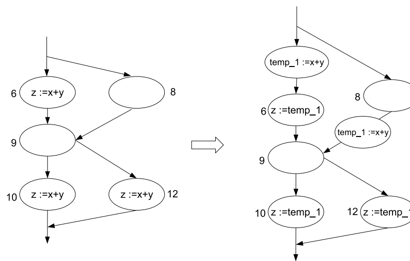

図 6: 書換えの例

8 実験

本研究の実装は番ら [1] の実装を大幅に拡張して作成した．SPECjvm98[16] の中の 7 つのベンチマークと奥村の Java コード [12] を使って，実験を行い，データを取得した．

実験環境　実験環境は下記である．

CPU：Celeron 2GHz

Memory：512MB

Soot：version 2.2.0

計測のオプション：-Xint -Xms128m -Xmx128m(JIT とメモリの影響を取り除いた)
適用した最適化：部分冗長性除去 (共通部分式除去とループ不変式も含む)-コピー伝播 (と定数伝播)-無用命令除去

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 2 に示す.

testcode	compile time
200_check	20
201_compress	29
202_jess	123
209_db	15
213_javac	219
227_mtrt	160
228_jack	316

表 2: SPECjvm98 の最適化時間（単位：秒）

奥村のコードの最適化時間を表 3 に示す.

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 3: 奥村のコードの最適化時間（単位：秒）

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は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.

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 7，8 に示す (最適化なしを 1 に正規化した).

SPECjvm98 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，通常のアルゴリズムにより最適化した結果も載せてあるが，202_jess 以外は殆ど効果がない.

最適化の効果に影響する原因については 9.1 節で考察する.

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.

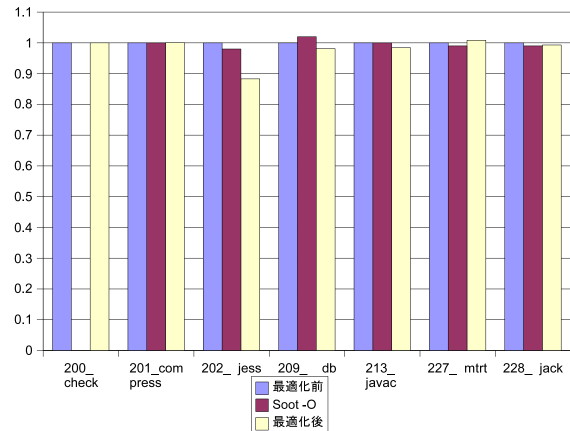

図 7: SPECjvm98 ベンチマーク最適化効果

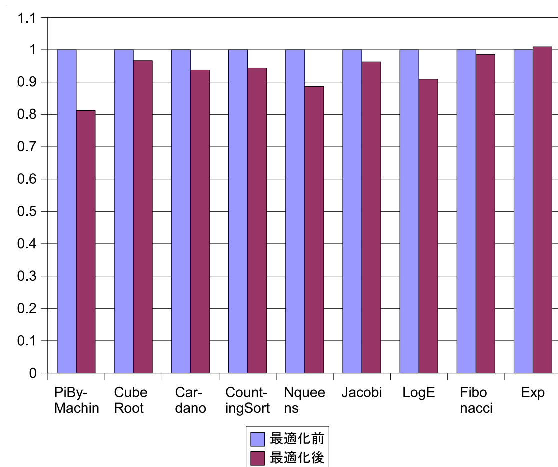

図 8: 奥村の Java コードの最適化効果

番らの手法との比較 番らの手法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 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%になっている.

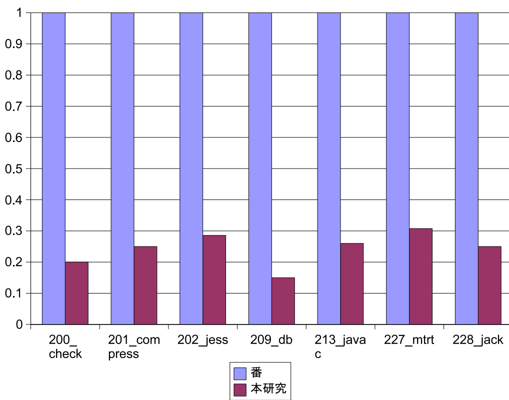

図 9: 番らの手法と本研究の最適化時間の比較

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 10 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．

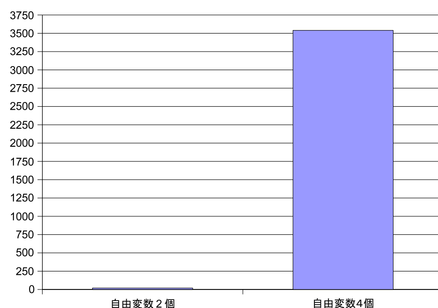

図 10: 同じ最適化を異なる CTL 式で記述したときの最適化時間の比較 (縦軸：秒)

9 考察と今後の課題

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．

9.1 考察

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，通常の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ないが，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 (6 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，効率は通常の最適化器より遅い．

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 [11].

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 11(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 11(中) に示す．CompleteGraph 上で行った場合の例を図 11(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．

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．

9.2 今後の課題

今後の課題は 2 つの方向がある．

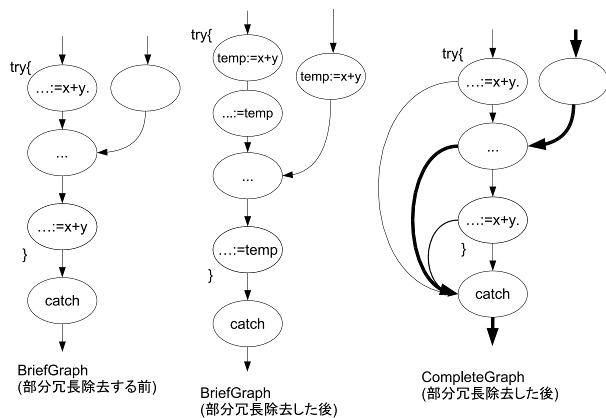

図 11: 例外による最適化の阻害

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．

10 関連研究

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仕様の多くは，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，μ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．最適化時間などのデータもない．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．

山岡らの研究 [19] は，既存のモデル検査器 SMV[15] を使って実装したが，将来時制しか扱えないうえ，述語は def() と use() しかないため，無用命令除去しか扱えない．

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なり長い．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，最適化仕様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．

伊藤らの研究 [7] 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，

なお，多数の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．そのため束縛の数が爆発すると予測される．モデル検査による最適化の効果及び解析の時間について，ベンチマークを使った具体的なデータが提示されていない．

11 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 μ 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．
- 最適化記述を改善し，Kripke 構造のノード番号を書かなくてもよくした．また，モデル検査器は条件式を満たす特定の番号の命令文ではなく，命令文の集合を計算する．そのため，同じ条件式を満たす多数の命令文を書換える処理が容易に記述できるようになった．部分冗長性除去など論理式で書きにくい最適化を定式化できた．
- 実際の最適化処理に欠かせない処理をいくつか加え，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，多くの最適化が，実用的な時間内で行えるようになった．
- 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．

今後は，問題点を解決し，さらに性能を改善できたら，双方向 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るようになることを期待する．

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).

[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] Franficois 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] 方玲．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・計算科学専攻修士論文 2006.

[7] 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．

[8] 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．

[9] 中田育男．コンパイラの構成と最適化．朝倉書店, 1999.

[10] 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.

[11] 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.

[12] 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/

[13] 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/

[14] 佐々 政孝．プログラミング言語処理系．岩波書店, 1989.

[15] SMV Model Checker. http://www.cs.cmu.edu/modelcheck/smv.html

[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

MATCH

$v := e$

CONDITION

$point_comp$: $use(e) \wedge trans(e)$

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

$point_avout$: $point_comp \vee (point_avin \wedge trans(e))$

$point_antout$: $AX(A\ trans(e)\ U\ use(e))$

$point_antin$: $point_comp \vee (point_antout \wedge trans(e))$

$point_safein$: $point_avin \vee point_antin$

$point_safeout$: $point_avout \vee point_antout$

$point_spavin$: $point_safein \wedge \overleftarrow{E}X(\overleftarrow{E}\ (trans(e) \wedge (point_safeout))\ U\ use(e))$

$point_spavout$: $point_safeout \wedge (point_comp \vee (point_spavin \wedge trans(e)))$

$point_spantout$: $point_safeout \wedge EX(E\ (trans(e) \wedge (point_safein))\ U\ use(e))$

$point_spantin$: $point_safein \wedge (point_comp \vee (point_spantout \wedge trans(e)))$

$point_insert$: $point_comp \wedge \neg\ point_spavin \wedge point_spantout$

$point_replace$: $(point_comp \wedge point_spavin) \vee (point_comp \wedge point_spantout)$

$point_edge1$: $point_spavin \wedge point_spantin$

$point_edge2$: $\neg\ point_spavout$

$edge_split$: $point_edge1 \longrightarrow point_edge2$

PROCESS

$point_insert : InsertBefore\ temp := e$

$edge_split : EdgeSplit\ temp := e$

$point_replace : replace\ e \rightarrow temp$