双方向CTLによるJava最適化器の生成

方 玲[†] 佐々 政孝[†]

近年，時相論理によるコンパイラ最適化器生成の研究が行われている．しかし，実際のプログラムを対象として最適化時間や目的コードの実行時間を提示した研究はない．本研究は従来研究の弱点を克服し，時相論理による記述から Java 言語の効率良い最適化器を生成するシステムを作成した．論理としては，分岐時相論理の 1 つで過去時制と未来時制をともに扱える双方向 CTL である CTL-FV[9] を用いた．最適化変換を記す仕様記述も，従来法と異なり，条件式を満たす特定の番号の個々の命令文ではなく，命令文の集合を計算するようにしたので，複雑な最適化が容易に記述できるようになった．さらに，本研究は一時変数を記述できるようにするなど種々の実用化の工夫を行い，Java 言語に対する典型的なコンパイラ最適化器を実現した．本研究が実装したモデル検査器は何の変換も行わずに，過去時制と未来時制を直接扱えるため，十分な効率で動作する．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，実験により，本研究では SPECjvm98 の 7 つについて，4 秒〜4 分で最適化が行えることを確認できた．また，通常の最適化器に近づいた性能を持った，CTL による最適化器の実装は本研究が初めてであるため，その可能性と問題点を明らかにした．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．

Generating Java Compiler Optimizers Using Bi-directional CTL

Ling Fang[†] and Masataka Sassa[†]

There have been several research works that analyze and optimize programs using temporal logic. However, no evaluation of optimization time or execution time of these implementations has been done for any real programming language. In this paper, we present a system that generates a Java optimizer from specifications in a kind of bidirectional temporal logic CTL-FV[9]. We also present a new specification language based on the bidirectional CTL that can express typical optimization rules very naturally. By adding rewriting conditions to allow for temporary variables and considering real-world language features such as exceptions, the system can perform optimization of Java programs. We implemented a model checker that can check future and past temporal CTL operators symmetrically without any conversion. So far, a compiler optimizer using temporal logic was assumed to be impractical, because it consumes too much time. However, with our method, the generated Java compiler optimizer can compile seven of the SPECjvm98 benchmarks with a compile time from 4 seconds to 4 minutes. To our knowledge, our system is the first system that can make optimizers for real Java programs from specifications in CTL by using a model checker. So, the possibility and problems of this approach and consideration on how to overcome the problems is clarified by our work. We also gained insights into improving existing techniques for decreasing the compilation time and in specifying compiler optimizations.

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[1),12),16)]．

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[4),22)] 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，すぐ後で述べる条件付き書き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．

この手法のメリットは，

- 通常の最適化器のように数百行ものプログラムを書くことなく，論理式で 1 行か，多くても十数行で最適化を記述できる，
- 数少ない論理式で書かれるため，解析や証明がしやすい，

ということである．

条件付き書き換え規則（conditional rewrite rule）

† 東京工業大学大学院情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology

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．

$v := e \Longrightarrow skip$
if $\neg EX(E\ \neg def(v)\ U\ use(v))$

本研究で用いる時相論理は，Lacey らが提案した CTL-FV [7] である．CTL-FV は過去時制が未来時制と対称的に使えるうえ，自由変数が導入されている．

本研究は，CTL-FV に基づくコンパイラ最適化記述法を提案した．この記述法は実際のプログラム最適化を自然な形で容易に記述できる．

さて，モデル検査を行う前には自由変数を束縛しないとならない．従来研究[9] では，自由変数は計算量に大きな影響がないと主張していたが，本研究では自由変数は最適化時間に大きく影響することを明らかにした．自由変数が多い場合は，処理時間が長くなり，現実的ではなくなる．

従来研究[3),9),24)] の記述法は Kripke 構造のノード番号を用いているため自由変数が多くなりやすい，また，多数の命令文が条件式を満たしたとき，このような記述法では記述が不自然になる．本研究の記述は，従来の方法と異なり，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き換える処理が容易に記述できるようになった．また，本研究の定式化は自由変数を減らすように努力した．したがって，本研究が提案した双方向 CTL に基づくコンパイラ最適化記述法は実際のプログラム最適化を自然な形で容易に記述できる．最適化記述についてはさらに，書き換え条件や一時変数などを扱う処理を加え，これにより典型的な Java 言語コンパイラ最適化器を実現した．

本研究では双方向 CTL モデル検査器を実装した．従来研究は，NCTL [10] や μ 計算に変換することで実装しているが，我々の方法では過去時制を未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く，処理時間が短い．このようにして処理の能力を向上させ，従来研究の欠点を克服できた．モデル検査器は自作したので，改良，拡張がしやすい．

前記の最適化記述法とモデル検査器を用いることにより，本研究では従来研究に比べ，通常の最適化の性能に大幅に近づけることができた．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，SPECjvm98 ベンチマークの 7 つについて，4 秒から 4 分で動作する最適化器が生成できることを確認できた．従来研究が処理できない最適化も多く処理できるようになった．

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．

我々の知る限り，通常の最適化器に近づいた性能を持った，時相論理による最適化器の実装は本研究が初めてである．

2. CTL-FV

CTL（computation tree logic）[4),22)] は分岐時相論理（branching-time temporal logic）の 1 つである．経路限量子として，A=(All)，E=(Exist) がある．時相演算子として，U=(Until)，X=(neXt)，G=(Globally)，F=(Future) がある．CTL では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．したがって，CTL は $A(\phi_1\ U\ \phi_2)$，$EX(\phi)$，... などの形をしている式しか書けない．経路限量子と直後の時相演算子のペアを本論文では結合子と呼ぶ．

CTL-FV は CTL に自由変数を導入したものであり，提案は文献 7) に始まる．CTL-FV は次のような特徴がある．

- 過去時制は未来時制とまったく同等，対称的な存在である．CTL-FV では CTL の未来時制演算子と対称的な過去時制演算子を持つ．CTL と異なり，経路限量子は A，E のほかに $\overleftarrow{A}$，$\overleftarrow{E}$ を持ち，それぞれ A，E を逆向きにしたものである．
- 論理式に自由変数の出現を許した．

以上の特徴があるため，簡潔性と表現力および効率が優れ，制御フローやデータフローの性質の記述に適している．

2.1 構文規則

CTL-FV の構文規則は以下のとおりである．

$$\begin{aligned}\phi ::= & \mathit{true} \mid \mathit{false} \mid \alpha \\ & \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \\ & \mid E\psi \mid A\psi \mid \overleftarrow{E}\psi \mid \overleftarrow{A}\psi \\ \psi ::= & X\phi \mid \phi\ U\ \phi\end{aligned}$$

ここで α は原子命題である．また，構文規則には現れないが，下記の結合子が使われることもある．それらは EX，EU，AU を用いて表せる[4)]（過去時制は自明なので略す）．

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

$EF\phi = E[true\ U\ \phi]$
$AG\phi = \neg EF(\neg\phi)$
$AF\phi = A[true\ U\ \phi]$
$EG\phi = \neg AF(\neg\phi)$
$A[\phi_1\ R\ \phi_2] = \neg E[\neg\phi_1\ U\ \neg\phi_2]$
$E[\phi_1\ R\ \phi_2] = \neg A[\neg\phi_1\ U\ \neg\phi_2]$

2.2 意 味 論

CTL-FV の意味論は Kripke 構造によって与えられる．Kripke 構造 K は 3 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原子命題の集合を割り当てる関数である．

K における s_0 からの経路とは，$\forall i \geq 0 : (s_i, s_{i+1}) \in R$ となるような状態の無限の列 $p = (s_0 \to s_1 \to s_2 \ldots)$ である．s_0 からの逆向きの経路とは，$\forall i \geq 0 : (s_{i+1}, s_i) \in R$ となるような無限列である．

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．

状態式：

$s \models true$ iff $true$
$s \models false$ iff $false$
$s \models \alpha$ 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 $s \models \phi_2$
$s \models \phi_1 \vee \phi_2$ iff $s \models \phi_1$ または $s \models \phi_2$

$s \models E\psi$ iff $\exists path(s = s_0 \to s_1 \to s_2 \ldots) :$
$\quad (s_i)_{i \geq 0} \models \psi$
$s \models A\psi$ iff $\forall path(s = s_0 \to s_1 \to s_2 \ldots) :$
$\quad (s_i)_{i \geq 0} \models \psi$
$s \models \overleftarrow{E}\psi$ iff $\exists path(\ldots s_2 \to s_1 \to s_0 = s) :$
$\quad (s_i)_{i \geq 0} \models \psi$
$s \models \overleftarrow{A}\psi$ iff $\forall path(\ldots s_2 \to s_1 \to s_0 = s) :$
$\quad (s_i)_{i \geq 0} \models \psi$

経路式：

$(s_i)_{i \geq 0} \models X\phi$ iff $s_1 \models \phi$
$(s_i)_{i \geq 0} \models \phi_1\ U\ \phi_2$ iff $\exists k \geq 0 :$
$\qquad [s_k \models \phi_2 \wedge \forall i : [0 \leq i < k \Rightarrow s_i \models \phi_1]]$

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 2 つの木構造を持つ

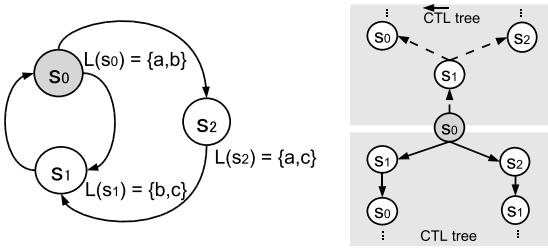


図 1 Kripke 構造とその構造に対応した s_0 からの双方向 CTL 無限木
Fig. 1 Kripke structure (left) and its CTL (infinite) trees (right) for s_0.

(以後，明らかな場合は双方向 CTL をたんに CTL と書くことがある)．

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．

3. 制御フローモデル

3.1 プログラムの構文

説明のため，ここでは手続きなどがない簡単な命令型言語を考える．

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

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

命令文の BNF を以下に記す．

$I ::= skip | X := E | if\ X\ goto\ n\ else\ n$
$\quad | goto\ n \mid read\ X \mid write\ X$
$E ::= X | E\ O\ E$
$O ::= + \mid - \mid * \mid / \mid \ldots$
$X ::=$ 変数名
$n ::= 0|1|2| \ldots |m$

本章では，Kripke 構造として制御フローモデルを定義する．

3.2 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 3 つ組 $M_\pi = (Node_\pi, \to_\pi, L_\pi)$ として定義される．ここで $Node_\pi$ は命令文のラベルの集合であり，関係 $\to_\pi$ は次のように定義される関係である．以下，明らかな場合は π を省略する．

$n_1 \to_\pi n_2$ iff
$\quad (I_{n_1} \in X := E, skip, read\ X) \wedge n_2 = n_1 + 1$
$\vee\ (I_{n_1} = goto\ n \wedge n_2 = n)$

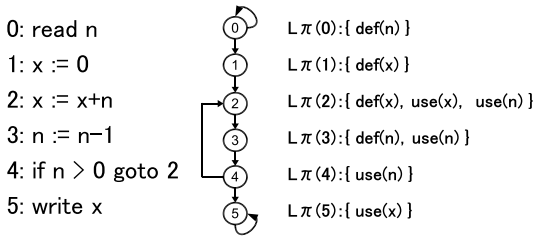


図 2 コードと制御フローモデルと原子命題の例
Fig. 2 Example of code, its control flow model and atomic formulas.

$\vee\ (I_{n_1} = if\ X\ goto\ n\ else\ n' \wedge (n_2 = n \vee n_2 = n'))$
$\vee\ (I_{n_1} = read\ X \wedge n_2 = n_1)$
$\vee\ (I_{n_1} = write\ X \wedge n_2 = n_1)$

この最後の 2 つの式から分かるように，最初の命令文の前のノードと最後の命令文の次のノードは自分自身とする．

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

$L_\pi(n) = \{stmt(I_n)\}$
$\bigcup \{def(x) \mid I_n$ は $x := E$ あるいは $read\ x$ の形式である $\}$
$\bigcup \{use(x) \mid I_n$ は $X := E$ の形式であり，x は E で使用される．
あるいは，$I_n = if\ x\ goto\ n\ else\ n'$．
あるいは $I_n = write\ x\}$
$\bigcup \{trans(E) \mid E$ は π の式であり，かつ，E の中のすべての変数 x に対して，I_n は $x := E'$ あるいは $read\ x$ の形式ではない $\}$
$\bigcup \{entry(n) \mid n$ はプログラムの入口である $\}$
$\bigcup \{exit(n) \mid n$ はプログラムの出口である $\}$

図 2 の左はコードであり，中はそのコードと対応した制御フローモデル，右は $L_\pi(n)$ のうちの def と use である．

4. 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．この記述は簡単で自然に通常の最適化を記述できる．

4.1 最適化記述の構成

本システムの最適化の記述は **MATCH**，**CONDITION**，**PROCESS** の 3 つの部分からなる．**MATCH** はモデル検査の対象となる命令文の形を示す．**CONDITION** は命令文が最適化されるとき満たすべき CTL 式を表す．**PROCESS** は **CONDITION** の条件式が満たされたとき，どのように変換するかを書くところである．最適化記述は次のような形をしている，詳しくは文献 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：コマンド 文
*point*_文字列：*Replace* 式 → 式
*edge*_文字列：EdgeSplit 文

以下は無用命令除去の最適化記述である：

MATCH
$v := e$
CONDITION
$point\_delete:\ \neg EX((E\ \neg def(v)\ U\ use(v))$
PROCESS
$point\_delete:\ delete\ v := e$

MATCH 部は，最適化の対象式の形を決めるとともに，CONDITION 部に現れる自由変数をプログラムの中の変数と束縛する．たとえば，

MATCH
$v := b$

と書くと，右辺が二項式の形の命令文を対象とする．v は変数，b は二項式である．したがって，図 2 のプログラムに対して，$x := x + n$ は対象となるが．$x := 0$ は対象とならない．命令文 $x := x + n$ が最適化の対象となったとき，$\{v \mapsto x, b \mapsto x + n\}$ のように束縛する．これにより，束縛されるのは $x = x + n$ と $n = n - 1$ の 2 つになり，プログラム中に存在しない命令文との束縛を避けることができる．

CONDITION 部では，条件式とすぐ後で述べる部分式を複数書いたり，それらに式名をつけたりすることができる．

条件式は書き換えをするときに成り立つべき条件である．条件式には PROCESS 部での処理と対応させるための式名を付ける．上の例での $point\_delete$ は式名であり，その名前が表す CTL 式「$\neg EX((E\ \neg def(v)\ U\ use(v))$」は不要で削除する条件を表す．

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．条件式に部分式の式名が書かれているときは，その名前が表す論理式に置き換えてモデル検査を行う．この例には現れないが，4.2.2 項および付録の例での *point_comp* などは部分式の式名である．

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 木構造とは関係がなく，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式名は従来研究[3),9),24)] でのノード番号と異なり，自由変数ではない（自由変数とは論理式で束縛されていない変数，という意味で，プログラムの変数とは異なることに注意）．

PROCESS 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変換するかの処理を書く．変換を表す処理に条件式と同じ式名を付けることによって，条件式との対応関係を付ける．上の例では，条件式

$point\_delete:\ \neg EX((E\ \neg def(v)\ U\ use(v))$

と対応した処理

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

が条件式が成り立った命令文 "$v := e$" を削除することを指示する．必要があればこの部分に一時変数を導入することができる．

処理としてはプログラムの命令文を処理する「コマンド」として *InsertBefore*，*InsertAfter*，*Delete* および *Replace* がある．辺を処理するコマンドとして *EdgeSplit* がある．各コマンドの意味は文字どおりで，命令文を前に挿入，後に挿入，削除を行う．「*Replace* 式 → 式」は式の一部を書き換える．*EdgeSplit* は辺に命令文を挿入する．

point で始まる式名は命令文を対象とし，*edge* で始まる式名は辺を対象とする．

さきに，式名は従来研究でのノード番号と異なり，自由変数ではない，と述べた．これについて説明する．

従来研究の記述法は Kripke 構造のノード番号を用いているため，自由変数が多くなりやすい．これに対し，本研究の記述は Kripke 構造のノード番号を書かなくてよい．CTL 式の前に現れるのはこの式を満たすノードの集合を代表する名前である，実際の最適化器を作るとき，番号を書く方法と比べ，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．

自由変数の数はシステムの効率に大きく影響する（5 章を参照）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．

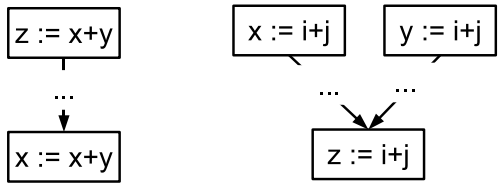


図 3 従来研究が扱える例（左）と扱えない例（右）
Fig. 3 Example which can not be processed by perious work.

また，従来研究の最適化記述では Kripke 構造のノード番号を書くことになっているため，いくつの最適化について，定式化してあるものの，典型的な例で扱えないものがある．

たとえば，文献 8) の共通部分式除去の記述：

$n : (a := e[b]) \Longrightarrow (a := e[v])$
$if\ n \models \overleftarrow{A}(trans(b) \wedge\ \neg def(v)\ U\ stmt(v := b))$

これは，**図 3** の左のケースを扱えるが，右のケースを扱えない．

4.2 最適化の CTL による定式化

この節ではコンパイラ最適化の定式化の 2 つの方法について述べる．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．

4.2.1 最適化の条件をもとにした方法

これは従来研究[3),9),24)] の定式化手法と同じである．前述の無用命令除去はこの例である．この方法をもとにして定式化するときは，実際の言語の特徴やモデル検査の効率を考える必要がある．たとえば，コピー伝播はコピー元を視点として未来時制を用いて定式化ができるが，コピー先を視点として過去時制を用いて定式化することもできる．このとき，定式化した論理式の自由変数の数や式の長さによって，モデル検査の効率が変わるので，効率の良い定式化方法を選ぶことが望ましい．

4.2.2 データフロー方程式をもとにした方法

時相論理とデータフロー解析との関係は Steffen [19)] の研究以来指摘されている．我々は実用的な立場からデータフロー方程式を利用した定式化も CTL の枠組みで記述できるようにした．

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き換える処理をしないとならない．これを 4.2.1 項のように時相論理の考え方で初めから定式化するのは難しい．

これに対し，データフロー方程式をもとにした定式化は初めから時相論理式を考えるのに比べてはるかに容易である．

本研究で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[15] 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簡明で，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に基づいて記述した CTL 式について説明する．記述全体は付録に示す．

Paleri らのデータフロー方程式をもとに CTL 式を記すときは，ほとん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* はそうである．しかし，データフロー方程式では，データフロー情報の値が決まるところから始めて，収束するまで繰り返す方法で解いていくことができる．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で解を求める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 $AVIN_i$，$AVOUT_i$ がそれぞれブロック i の入口，出口で該当の式が利用可能であることを表すとするとき，式が計算されることを $COMP_i$，式のオペランドがノード i の計算によって変更されることがないことを $TRANSP_i$ で表すと，次のようなデータフロー方程式が成り立つ．

$$AVIN_i = \begin{cases} false & \textbf{if } entry \\ \prod_{j \in preds(i)} AVOUT_j & \textbf{if } other \end{cases}$$

$$AVOUT_i = COMP_i \vee AVIN_i \wedge TRANSP_i$$

これは

- $AVIN_i$ 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，

ことを意味する．

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ところから始めて，$AVIN_0$，$AVOUT_0$，$AVIN_1$，$AVOUT_1$，$AVIN_2$，... のように収束するまで繰り返す方法で解いていくことができるが，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．

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ず，その値の再計算をしなくてもよい場所でのみ 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

$$\begin{aligned} &point\_comp: \ use(e) \wedge trans(e) \\ &point\_avin: \ \overleftarrow{A} X(\overleftarrow{A} \ trans(e) \ U \ point\_comp) \\ &\qquad\qquad \wedge \neg entry \\ &point\_avout: \ point\_comp \vee (point\_avin \wedge trans(e)) \end{aligned}$$

のようにすればよい．

これにより，データフロー方程式のセマンティクスを保ちつつ，繰返しなしで計算できるような式に書き換えることができた[6]．

なお，Lacey の論文[8] の付録にも，同様の考え方が記されているが，マクロを多用し，記述も複雑で，かつ実装されたデータが載っていないため，考え方の提示にとどまっていると思われる．

5. 自由変数

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 Lacey ら[7] の研究で導入された．

自由変数とは，CTL 式の述語に現れて，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，自由変数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．図 2 のプログラムを例とすると，自由変数の定義域は

$v \mapsto \{n, x\}$
$e \mapsto \{x + n, n - 1\}$

である．すなわち，自由変数の定義に忠実に従うと

$\{v \mapsto n, e \mapsto x + n\}$
$\{v \mapsto n, e \mapsto n - 1\}$
$\{v \mapsto x, e \mapsto x + n\}$
$\{v \mapsto x, e \mapsto n - 1\}$

のように束縛することになる．

自由変数の束縛は全探索になるため，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．

本研究を含めて，自由変数は多くの従来研究[3),9),24)]で採用された．これは，自由変数を導入することによって，CTL 式の表現力と便利さが向上するからである．しかし，以上述べたように，自由変数束縛の計算量は大きいので，実際の最適化器では自由変数をできるだけ避けるべきである．しかし MATCH 段階での自由変数をいかに減らすかは将来の課題である（8.2 節参照）．

6. 双方向 CTL による最適化器

本研究では双方向 CTL による最適化器を作成した．これは前処理部，モデル検査器と書き換え部の 3 部分から構成される．**図 4** は本最適化システムの略図である．前処理部は入力プログラムを中間言語の 3 番地コードに替え，自由変数をプログラムの変数や式に束縛する．モデル検査器は前処理部から渡された 3 番地コードと束縛した最適化の CTL 式を使ってモデル検査を行う．書き換え部はモデル検査器で検査した結果に基づき最適化の書き換え規則を適用し，コードを出力する．

システムの実装には Soot[21)] を利用した．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，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 内部で様々な中間表現を経由して処理が行われる．Jimple は中間表現のうちの 1 つの型付けされた 3 番地表現であり，プログラム変換処理に適した形をしている．また，Soot が提供したメソッドを利用してプログラムの分析やデータフロー解析などが行えるので便利である．

6.1 前処理部

前処理部はモデル検査をするための前段階として入力を 3 番地コードに変換し，CTL 最適化の記述を解析し，自由変数の束縛を行う．前処理部の処理手順は以下のとおりである．

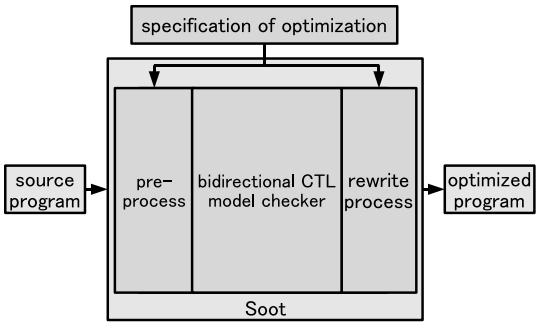


図 4 最適化システム略図
Fig. 4 Outline of the optimization system.

- 入力される Java プログラム，あるいはクラスファイルを中間表現 Jimple に変換する．
- 入力プログラムに対して行う CTL 最適化記述を読み込み，CTL 解析木を生成する．
- CTL 式の自由変数と入力プログラムの式や変数などを束縛する．
- Jimple に変換されたコードと自由変数が束縛された CTL 解析木をモデル検査器に入力する．

6.2 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．

完全に対称な過去時制と未来時制を持つような CTL のモデル検査自体は，同じ状態集合 S に対して Kripke モデルの関係 R と R^{-1} の 2 種類の様相を与えた場合のモデル検査器であり，アルゴリズムは本質的に通常の CTL モデル検査のそれと変わらない．未来時制の検査は CTL 木の真理値を求める．過去時制の検査は未来時制を逆向きにし，$\overleftarrow{CTL}$木の真理値を求める．モデル検査アルゴリズムは，EX，EU は文献 4) を参考にした．AU は文献 2) を参考にし，速く収束するように工夫した．詳細は，文献 6) に記載してある．

このモデル検査器は explicit state モデル検査器である．本研究では将来，効率向上のため，bit vector や部分計算などの様々な手法を使ってデータ構造やアルゴリズムの改善を図れるようにするため，既存のモデル検査器を利用せず自作した．

通常のモデル検査器は結合子 EX，EU，EG をプリミティブな結合子とするが，EG の実装は強連結成分（SCC）分解がともなう．本システムの場合は，書き換えが発生するたび SCC の計算が必要となり，計算の手間が増えることが予想されたので，本研究の実装は EX，EU，AU およびその過去時制版をプリミティブな結合子として実装し，ほかの結合子はそれによって導出することにした．未来時制の部分の導出関

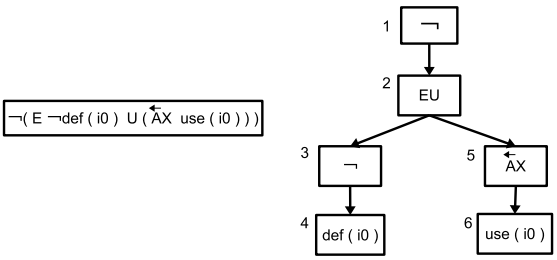


図 5 双方向 CTL 論理式と双方向 CTL 構文木
Fig. 5 Bidirectional CTL formula and it's syntax tree.

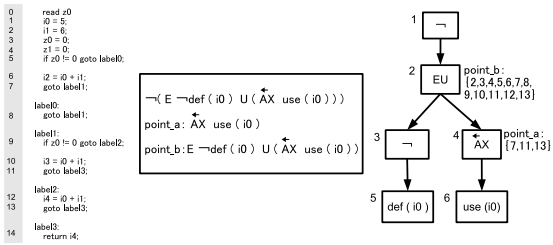


図 6 モデル検査処理
Fig. 6 Model checking.

表 1 図 5 の双方向 CTL 構文木の節と部分式
Table 1 Bidirectional CTL syntax tree nodes and partial formulas of Fig. 5.

節	演算子	子	対応した部分式
1	$\neg$	2	$\neg(E\ \neg def(i_0)\ U\ \overleftarrow{A}X(use(i_0)))$
2	EU	3 5	$E\ \neg def(i_0)\ U\ \overleftarrow{A}X(use(i_0))$
3	$\neg$	4	$\neg def(i_0)$
5	$\overleftarrow{A}X$	6	$\overleftarrow{A}X(use(i_0))$
4	ap	$def(i_0)$	$def(i_0)$
6	ap	$use(i_0)$	$use(i_0)$

係は 2.1 節を参照.

双方向 CTL 式の解析

双方向 CTL 式は木構造で表せる．これを双方向 CTL 構文木と呼ぶ（双方向 CTL 木とは異なることに留意）．木の葉は原子命題である．**図 5**（左）の双方向 CTL 式の構文木を図 5（右）に示す．各部分式は **表 1** のようになる．

双方向 CTL のモデル検査

双方向 CTL のモデル検査は，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 ごとに各状態 s で満たされるかどうかを計算する．すなわち状態 s_i 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き換え処理に使用されるため，モデル検査をしながら，書き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．

図 6 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過程を示したものである．CTL 構文木ではそれぞれの節に式名がある場合，この式名が代表する CTL 式を満たす集合（対象集合と呼ぶ）の値を付加する．たとえば，右図の構文木の葉から根に向かって，対象としている部分式 $use(i0)$，$def(i0)$，$\overleftarrow{A}X(i0)\ldots$ が各状態 s $(0 : read\ z0, 1 : i0 = 5, 2 : i1 = 6, \ldots)$ で満たされているかどうかを計算する．書き換えに必要な条件式名，たとえば $point\_a$ の式を満たした状態のノード番号 $\{7, 11, 13\}$ を対応した $point\_a$ 集合に入れる．このような条件式を満たした状態の集合を書き換え部の入力とする．

モデル検査の計算量

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，モデル検査の計算量は $O(|\phi|(|S| + |R|))$ となる[4]．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 とすると，これは近似的に $O(n_1 \times n_2)$ と書け，プログラムの大きさと CTL 構文木の大きさに比例する．しかし，場合によって，n_1 や n_2 が非常に大きくなると，計算時間が劇的に増える可能性がある．

モデル検査時間短縮の工夫

モデル検査の理論的計算量は上記のとおりであるが，実際のモデル検査の時間は実装によって大きく変わる．本研究でも，アルゴリズム，コーディング，データ構造などの工夫によってモデル検査の時間を，ある例では 10 分の 1 以下に短縮することができた．たとえば次のような工夫を行った．

- 結合子の計算が速く収束するようにした．
  たとえば $AU_i[\phi_1, \phi_2]$ の計算はすべての後続ノードの真理値 $AU_{succ(i)}[\phi_1, \phi_2]$ を再帰的に計算する top-down な方式もあるが，ϕ_2 が $true$ となる個所から逆方向にたどって，ϕ_1 が成立つノードのうち，後続ノードで AU がすでに成り立っているときにだけ，新たにそこを AU が成り立つノードとする bottom-up な方式もある．後者のほうが，ϕ_1 と ϕ_2 が成り立ったノードしか探索しないため，効率が良い．
- CTL 構文木の計算について，いったん計算が済んだ CTL ノードの情報は hash table に入れることによって重複計算を避ける．
- プログラムを作成する際，再帰から例外を使って

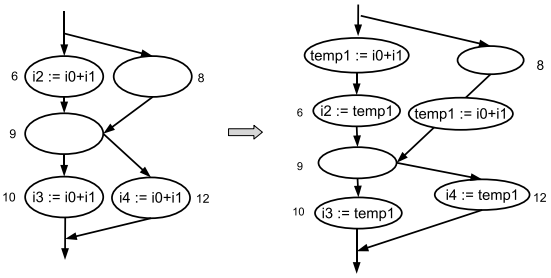


図 7 書き換えの例
Fig. 7 Example of rewriting.

脱出するのは便利だが，頻繁に脱出すると遅くなるため，そのような方法はできるだけ避ける．

6.3 書き換え部

書き換え部はモデル検査部の結果に対して，条件式名と対応した式について，書き換え処理を行い，最適化後のコードを出力する．

図 7 はモデル検査の結果を使って部分冗長性除去の書き換え規則を適用する前と後の例である．

7. 実　　験

SPECjvm98[18] の中の 7 つのベンチマークと奥村の Java コード[14] を使って，実験を行い，データを取得した．

7.1 実 験 環 境

実験環境は下記である．

CPU：Celeron 2 GHz

Memory：512 MB

Soot：version 2.2.0

JDK version：1.5.0_06-b05

計測のオプション：-Xint -Xms128m -Xmx128m（JIT とメモリの影響を取り除くため）

適用した最適化（本研究）：部分冗長性除去（共通部分式除去とループ不変式移動も含む），コピー伝播（と定数伝播），無用命令除去

適用した最適化（Soot，比較用）：共通部分式除去，部分冗長性除去，コピー伝播，定数伝播と畳み込み，条件分岐畳み込み，無用命令除去，到達不能命令除去，分岐不能分岐除去，無用ローカル変数除去

7.2 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．

奥村のコードの最適化時間を**表 3** に示す．

最適化時間は SPECjvm98 ベンチマークの 7 つについて 4 秒から約 4 分であり，ミリ秒から秒単位で最適化できる通常のコンパイラの最適化器と比べて遅いが，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やむをえない面もある．しかし後述するように，本研究は従来研究[3] と比べ，最適化時間がはるかに短くなった．また，228_*jack* については，巨大な switch-case 命令文によって，Kripke 構造 $K : (S, R, L)$ の遷移関係 $R \subseteq S \times S$ が大変大きいため，最適化時間が大きくなることが判明した．この点は今後検討したい．

表 2 SPECjvm98 の最適化時間（単位：ミリ秒）
Table 2 The optimization time of the SPECjvm98 benchmark (unit: millisecond).

コード名	束縛	モデル検査	書き換え	その他	合計
200	140	2,138	48	939	3,265
201	328	4,326	32	1,236	5,922
202	296	13,027	125	3,624	17,072
209	158	2,687	31	1,095	3,971
213	529	24,797	46	6,592	31,964
227	264	18,080	93	1,810	20,247
228	499	236,069	202	3,363	240,133

表 3 奥村のコードの最適化時間（単位：ミリ秒）
Table 3 The optimization time of Okumura's Java code (unit: millisecond).

コード名	束縛	モデル検査	書き換え	その他	合計
PiByMachin	125	344	16	46	531
CubeRoot	94	408	15	170	687
Cardano	110	1,062	31	109	1,312
CountingSort	125	468	15	79	687
NQueens	110	656	16	93	875
Jacobi	171	3,690	32	482	4,375
LogE	109	2,362	48	263	2,782
Fibonacci	109	344	16	187	656
Exp	141	1,439	32	327	1,939

7.3 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 8**，**図 9** に示す（最適化なしを 1 に正規化した）．

本研究は Soot で行う最適化の一部しか適用していないが，それなりの効果を得た．SPECjvm98 では 202_*jess*，213_*javac*，228_*jack* のベンチマークが本手法によって最適化なしに比べて数%速くなった．

この図には，Soot の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，201_*compress*，202_*jess*，213_*javac*，228_*jack* 以外はほとんど効果がない．

最適化の効果に影響する原因については 8.1 節で考察する．

7.4 従来研究との比較

Lacey ら[9] は最適化時間や実行時間のデータを与えていないので比較できない．

番ら[3] の手法は過去時制を含む NCTL[10] から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

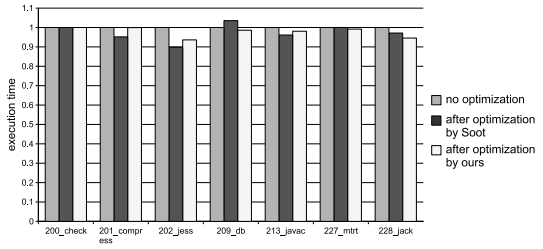


図 8 SPECjvm98 ベンチマーク最適化効果
Fig. 8 Effect of optimization for SPECjvm98 benchmark.

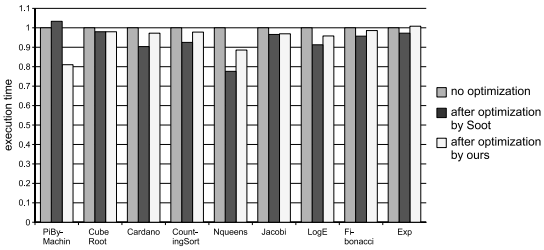


図 9 奥村の Java コードの最適化効果
Fig. 9 Effect of optimization for Okumura's Java code.

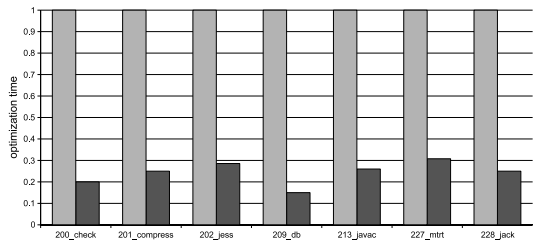


図 10 番らの手法と本研究の最適化時間の比較
Fig. 10 Optimization time of our system compared with Ban's work.

にしたことが特徴である．そこで，コピー伝播について，番らの手法による変換後の式と，本研究の定式化を使って本システムで実行した最適化時間とを前述の同じマシンで比較した．未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 10** に示す（番らの手法による最適化を 1 に正規化した）．

この結果，番らの手法と比べると，本研究の最適化時間は 15%から 30%と短くなっている．

7.5 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 11 に示したのはコピー伝播を例とし，異なる CTL 式を我々のシステムに入力し，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 4 秒から 39 分 28 秒に変わった）が起こる例である．

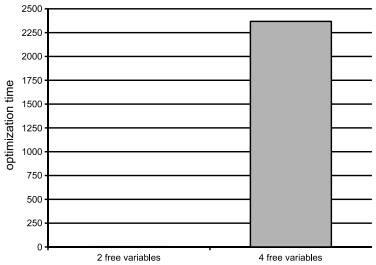


図 11 同じ最適化を異なる CTL 式で記述したときの最適化時間の爆発（縦軸：秒）
Fig. 11 Example of optimization time explosion (Vertical axis: second).

8. 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．通常の最適化器に近づいた性能を持った，CTL による最適化器の実装は本研究が初めてであるため，その可能性と問題点を明らかにすることが本章の主目的であるので，考察項目が多いことはけっして悪いことではないことに留意してほしい．

8.1 考　　察

8.1.1 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，「コピー伝播はもとの命令文が無用命令除去により消されるときのみ行う，消されないときは行わない」とか，「部分冗長性除去を行うとき，プロファイル情報を用いて計算をよく通るパスからあまり通らないパスに移動したら，正の効果が生じる．これは通常の保守的な最適化ではないが，このような最適化を行いたい」，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，条件分岐を考慮した定数伝播[23] 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「モデル検査の結果をすぐ適用しないで途中結果として覚えておく．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ならないが，CTL では記述できない．

また，部分冗長性除去が時間的に最適な結果を得ることを証明するには，CTL 式に基づく証明は難しく，データフロー方程式に基づく証明が現実的である．

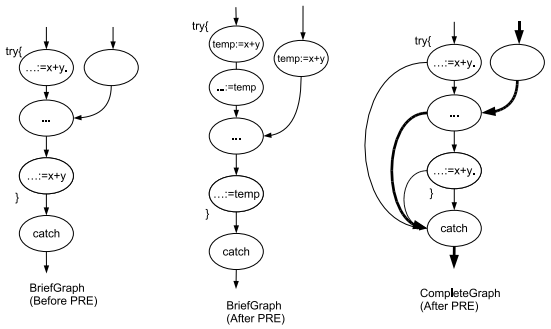


図 12 例外による最適化の阻害
Fig. 12 Obstruction of optimization by exception.

8.1.2 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 CTL の自由変数集合とプログラムの変数集合との全組合せになる（5 章参照）．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．

8.1.3 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越えることはできない．配列，割り算，剰余算など実行時例外を起こしうるものも全部対象外になるといった制限がある[13]．

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．後者は制御フローグラフの上にさらに try 命令文に囲まれたすべての命令文から catch 命令文への辺を引いたグラフである，**図 12**（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 12（中）に示す．CompleteGraph 上で行った場合の例を図 12（右）に示す．図 12（右）では $x+y$ の移動は太線のパスに抵触するため，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．現在の実装は CompleteGraph 上で行っており，保守的なアルゴリズムとなっている．

コード移動が例外発生点を越えられないという問題は，本研究だけでなく，通常の Java 最適化器にも存在する．

8.2 今後の課題

今後の課題は 2 つの方向がある．

8.2.1 最適化時間の短縮

モデル検査器を速くするために，BDD[20] を導入したり，bit vector や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．

自由変数の束縛は最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．

モデル検査の効率と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．

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

- ほかの束縛によって束縛しなくてよい束縛をなくす．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 $5 : x := z + 1$ によって代入されているから，さらに前の命令文 $1 : x := 100$ 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，素朴には $\{v \mapsto z, c \mapsto 3\}$ も 1 つの束縛であるが，$z := 3$ という命令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．
- 時制経路上にない式と束縛しなくてもよい．過去時制のみの CTL 式は過去向きの経路上にある変数だけに束縛し，未来時制のみの CTL 式は未来向きの経路上にある変数だけに束縛する．また AX や EX は次の命令文だけに束縛し，さらに遠い命令文とは束縛しない，などである．
  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この実装は完成していない．

8.2.2 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．

条件分岐を考慮した定数伝播など複雑な最適化をどう書くのかも将来の課題である．

9. 関 連 研 究

従来の研究として，Cousot ら[5] は最初に時相論理をプログラム解析に関係づけた．

Lacey らの研究[7] は，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き換えで記述できることを示した．文献 8)，9) 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．文献 8) 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[24] は，既存のモデル検査器 SMV [17] を使って実装したが，未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[3] 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．過去時制に制限があるうえ，除去処理に時間がかかり，除去によって CTL 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 1 つの条件しか書けない，辺を扱えない，などの欠点があるため，無用命令除去とコピー伝播しか行えない．

Lerner ら[11] はドメイン指向言語による自動証明付きの最適化器を提唱した．本研究の時相論理による最適化手法とは異なる．

なお，多数の既存研究では定式化にプログラムの命令文と対応した Kripke 構造のノード番号を書くことになっている．それらではモデル検査による最適化の効果およ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が，そのような定式化の仕方から自由変数束縛の数が爆発すると予測される．

10. ま と め

本研究の主な貢献は次のとおりである．

- 従来研究[8] にあった APPLY_ALL や複雑なマクロなどを使わずに簡潔な形で記述できる最適化記述を提案した．Kripke 構造のノード番号を書かなくてもよくした．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き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．
- 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効率の良い双方向 CTL モデル検査器を実装した．
- 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 による実用性に近い 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得した．その結果，多くの最適化が，実用性に近い時間内で行えることが分かった．
- この経験を通じて，双方向 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と思われる．

今後は，問題点を解決し，さらに性能を改善し，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るようにすることを目指したい．

謝辞 有益なコメントをいただいた査読者，所属研究室の中谷俊晴，佐原聡一郎，計算工学専攻の伊藤宗平の各氏，また東京大学の胡振江先生，卒業生の番伸宏氏に感謝する．

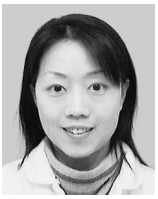
本研究の一部は科学研究費補助金の補助を受けた．

参 考 文 献

1) Aho, A.V., Lam, M.S., Ravi, S. and Ullman, J.D.: *Compilers Principles, Techniques and Tools*, 2nd ed., Addison Wesley (2006).
2) Berard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L. and Schnoebelen, P.: *Systems and Software Verification: Model-Checking Techniques and Tools*, The Springer Press (2001).
3) 番 伸宏, 胡 振江, 筧 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).
4) Clarke, Jr. E.M., Grumberg, O. and Reled, D.A.: *Model Checking*, The MIT Press (1999).
5) Cousot, P. and Cousot, R.: Automatic synthesis of optimal invariant assertions: Mathematical foundations, *SIGPLAN Not.*, Vol.12, No.8, pp.1–12 (1977). MIT Press (1999).
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.
http://www.is.titech.ac.jp/%7Esassa/lab/papers-written/04M54028.pdf
7) Lacey, D., Jones, N.D., Van Wyk, E. and Frederiksen, C.C.: Proving correctness of compiler optimizations by temporal logic, *Proc. Symposium on Principles of Programming Lan-*

guages, pp.283–294 (2002).
8) Lacey, D.: Program transformation using temporal logic specifications, Ph.D. Thesis, University of Oxford (2003).
9) Lacey, D., Jones, N.D., Van Wyk, E. and Frederiksen, C.C.: Compiler optimization correctness by temporal logic, *Higher-Order and Symbolic Computation*, Vol.17, No.3, pp.173–206 (2004).
10) Laroussinie, F. and Schnoebelen, P.: Specification in CTL+Past for verification in CTL, *Information and Computation*, Vol.156, No.1/2, pp.236–263 (2000).
11) Lerner, S., Millstein, T., Rice, E. and Chambers, C.: Automatically proving the correctness of compiler optimizations, *Proc. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.364–377 (2005).
12) 中田育男：コンパイラの構成と最適化，朝倉書店 (1999).
13) 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.SIG 1 (PRO24), pp.134–148 (2005).
14) 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/
15) Paleri, V.K., Srikant, Y.N. and Shankar, P.: A Simple algorithm for partial redundancy elimination, *ACM SIGPLAN Not.*, Vol.33, No.12, pp.35–43 (1998).
16) 佐々政孝：プログラミング言語処理系，岩波書店 (1989).
17) SMV Model Checker. http://www.cs.cmu.edu/modelcheck/smv.html
18) SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98
19) Steffen, B.: Generating data flow analysis algorithms from modal specifications, *Science of Computer Programming*, Vol.21, No.2, pp.115–139 (1993).
20) 田辺良則，高橋孝一，山本光晴，佐藤貴洋，萩谷昌己：BDD を用いた 2 方向 CTL 論理式充足可能性決定手続きの実装，コンピュータソフトウェア，Vol.22, No.3, pp.154–166 (2005).
21) Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P. and Sundaresan, V.: Soot —A Java optimization framework, *Proc. CASCON 1999*, pp.125–135 (1999). http://www.sable.mcgill.ca/soot/
22) Van Leeuwen, J. (Ed.): *Handbook of Theoretical Computer Science Vol.B, Formal Models and Semantics*, Elseveir Science Publishers B.V. (1990). 広瀬ほか（訳），丸善.
23) 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).
24) 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プログラミング，Vol.44, No.SIG 13 (PRO18), pp.25–37 (2003).

(平成 18 年 12 月 16 日受付)
(平成 19 年 3 月 13 日採録)



方 玲

2006 年東京工業大学大学院情報理工学研究科数理・計算科学専攻修士課程修了．2006 年同大学大学院情報理工学研究科数理・計算科学専攻博士課程進学，現在に至る．

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．現在同大学大学院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．

付録 部分冗長性除去の最適化記述

MATCH

$v := e$

PROCESS

$point\_insert:\ InsertBefore\ temp := e$

$edge\_insert:\ InsertBefore\ temp := e$

$point\_replace:\ replace\ e \rightarrow temp$

CONDITION

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

$point\_avin:\ \overleftarrow{A}X(\overleftarrow{A}trans(e)\ U\ point\_comp) \wedge \neg entry$

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

$point\_antout:\ AX(Atrans(e)\ U\ point\_comp) \wedge \neg exit$

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

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

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

$point\_spavin:\ point\_safein \wedge \overleftarrow{E}X(\overleftarrow{E}(trans(e) \wedge (point\_safeout))\ U\ point\_comp) \wedge \neg entry$

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

$point\_spantout:\ point\_safeout \wedge EX(E(trans(e) \wedge (point\_safein))\ U\ point\_comp) \wedge \neg exit$

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

$point\_insert:\ point\_comp \wedge \neg point\_spavin \wedge point\_spantout$

$point\_replace:\ (point\_comp \wedge point\_spavin) \vee (point\_comp \wedge point\_spantout)$

$point\_edge1:\ point\_spavin \wedge point\_spantin$

$point\_edge2:\ \neg point\_spavout$

$point\_start:\ point\_edge1 \wedge point\_edge2 \wedge point\_insert \wedge point\_replace$

$edge\_insert:\ point\_edge1 \rightarrow point\_edge2$