日本ソフトウェア科学会第 26 回大会 (2009 年度) 講演論文集

時相論理CTL*を用いたJAVA最適化器の生成の試み

藤原 一貴　佐々 政孝

プログラムの最適化器は一般的にプログラムで書き下されるものであるが，近年，時相論理，特に CTL といわれる論理を使った最適化器の研究がなされている．具体的には，最適化の方法を CTL で表しモデル検査の手法を利用して最適化を行うというものである．この方法によるメリットは，プログラムで書き下された最適化に比べて証明が可能であり，また検査する論理式を多くても十数行で最適化を記述できるというところである．従来の研究として，Lacey ら [3] [4] は古典的な最適化器を時相論理によって表現できることを証明した．その後，山岡や番ら，さらに方などにより，時相論理を使った最適化器の実装に関する研究がなされてきた．本研究では，従来の研究で最適化を表現するのに使われていた，時相論理 CTL より表現力の強い CTL*を用いて，CTL では表現できない最適化について CTL*でそれを表し最適化を行うことを試みた．

1　はじめに

伝統的なコンパイラの最適化は，コントロールフロー解析やデータフロー解析といったプログラム解析の結果を用いて，適切なプログラム変換をするものである．この最適化を自動で行うツールである最適化器は，手動で実装されるものが普通であるが，ここ 10 年ほど，モデル検査の手法を用いて最適化ツールを作成する試みが研究されてきた．

近年考えられている最適化のアルゴリズムは複雑なものが多く，それを手動で実装した最適化器は，ソースコードが複雑でその量も長くなりバグの混入を防ぐのは困難である．一方モデル検査の手法を用いた最適化器は，手動で実装した最適化器に比べてアルゴリズムが簡単であり，一般的に実装にかかるコードの量が少なくてすむため，バグの混入する確率が少ないといわれている．

モデル検査とはシステムがある仕様を満たしているかどうかを検証する形式的検証法の一つであり，検証対象であるシステムの有限状態モデルとそのシステムが満たすべき仕様を表す時相論理式が与えられた時，対象のシステムがその仕様を満たしているかを網羅的に検査する．

モデル検査技術をプログラム解析に利用するには，プログラム解析の仕様が時相論理で記述される必要がある。近年，時相論理 CTL に自由変数を導入した CTL-FV によって多くの最適化が自然に記述できることが Lacey ら [3] [4] によって示された．その後，その理論をもとにモデル検査を用いたコンパイラの最適化器の実装の研究がなされてきた。

本研究では，その最適化器の実装に関する研究では扱えない種類の最適化を，時相論理 CTL よりも表現力の強い時相論理 CTL*を拡張した CTL*-FV というものを使って実装する手法を提案する。

以下本論文は次のように構成される．2 節ではモデル検査を用いた最適化器生成において基幹をなす時相論理について述べる．3 節では既存研究とその問題点，さらに本研究との相違点について述べる．4 節では最適化を時相論理によって表した最適化記述について述べる．5 節では本研究で提案する最適化の流れを具体的な例を用いて述べる．6 節では今回提案するシ

A Trial to Generating Java Compiler Optimizers Using CTL*.

Kazutaka Fujiwara, 東京工業大学数理・計算科学専攻, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology,

ステムでの実装面について述べる．7 節ではまとめと今後の課題について述べる．

2 時相論理

2.1 CTL-FV

CTL-FV は，分岐時間時相論理 (branching-time temporal logic) の一種である CTL(Computation Tree Logic) に自由変数を引数に持つ述語を導入し，さらに過去の時制を扱う逆向きの経路限量子が導入されている時相論理である．

そもそも分岐時間時相論理である時相論理 CTL は経路限量子 (path quantifier) と時相演算子 (temporal operator) の組によって時相を表現する．経路限量子は経路の分岐構造についての記述に使用され，A, E の 2 種類がある．時相演算子は経路の性質についての記述に使用され，F, G, X, U の 4 種類がある．

CTL-FV の構文を表 1 に示す．図中の記号は，! は否定，& は連言，| は選言，$\rightarrow$ は条件をそれぞれ表す．CTL-FV が CTL と異なる点は自由変数を持つ述語が導入されている点と，$\overleftarrow{A}$，$\overleftarrow{E}$ という逆向きの経路限量子が導入されている点である．

ϕ	$\in$	$CTL-FV\ proposition$
ϕ	::=	$predicate(x_1, \cdots, x_n)$
	|	$!\,\phi$
	|	$\phi\ \&\ \phi$
	|	$\phi \mid \phi$
	|	$\phi\ \rightarrow\ \phi$
	|	$E\ \psi$
	|	$A\ \psi$
	|	$\overleftarrow{E}\ \psi$
	|	$\overleftarrow{A}\ \psi$
ψ	::=	$X\ \phi$
	|	$F\ \phi$
	|	$G\ \phi$
	|	$\phi\ U\ \phi$

表 1 CTL-FV の構文規則

経路限量子 A，E およびその逆向きの $\overleftarrow{A}$，$\overleftarrow{E}$ は結果として状態式 ϕ を返すような経路式 ψ に関する演算子で，それぞれ以下のような意味を持つ．

- $A\psi$
  これ以降の全ての経路で ψ を満たせば真の状態
- $E\psi$
  これ以降に ψ を満たす経路が存在すれば真の状態
- $\overleftarrow{A}\psi$
  ここからさかのぼる全ての経路で ψ を満たせば真の状態
- $\overleftarrow{E}\psi$
  ここからさかのぼって ψ を満たす経路が存在すれば真の状態

また，時相演算子 F, G, X, U は結果として経路式 ψ を返すような状態 ϕ に関する演算子で，それぞれ以下のような意味を持つ．

- $F\phi$
  この経路上でいつか ϕ を満たせば真の経路
- $G\phi$
  この経路上で常に ϕ を満たし続ければ真の経路
- $X\phi$
  この経路上の次の時間で ϕ を満たせば真の経路
- $\phi_1\ U \phi_2$
  この経路上でいつか ϕ_2 を満たしかつそれまで ϕ_1 を満たし続ければ真の経路

次に CTL-FV 式中で用いるいくつかの述語の中で，今回必要になるものを説明する．

- $trans(x)$：変数 x について，値が定義されていないような場所で真
- $stmt(x := v)$：文 $x := v$ がある場所で真

2.2 CTL*-FV

CTL*-FV は，CTL と同じ分岐時間時相論理の一種である CTL* [1] に自由変数を引数にもつ述語と，過去の時制を扱う逆向きの経路限量子 $\overleftarrow{A}$，$\overleftarrow{E}$ を導入した時相論理である．

CTL*とは，CTL と同様に経路限量子と時相演算子を使って時相を表現するものである．特徴としては，CTL ではこの経路限量子と時相演算子を組にして使用しなくてはいけないという構文上の制限があったが，CTL*ではこの制限が無く，その分 CTL より

も強い表現力を持っている点が挙げられる．表 2 に CTL*-FV の構文を示す．意味は CTL-FV のそれと同様である．

$$
\begin{array}{lcl}
\phi & \in & CTL*-FV\ proposition \\
\phi & ::= & predicate(x_1, \cdots, x_n) \\
 & | & !\ \phi \\
 & | & \phi\ \&\ \phi \\
 & | & \phi\ |\ \phi \\
 & | & \phi\ \rightarrow\ \phi \\
 & | & E\ \psi \\
 & | & A\ \psi \\
 & | & \overleftarrow{E}\ \psi \\
 & | & \overleftarrow{A}\ \psi \\
 & & \\
\psi & ::= & \phi \\
 & | & !\ \phi \\
 & | & \phi\ \&\ \phi \\
 & | & \phi\ |\ \phi \\
 & | & \phi\ \rightarrow\ \phi \\
 & | & X\ \phi \\
 & | & F\ \phi \\
 & | & G\ \phi \\
 & | & \phi\ U\ \phi
\end{array}
$$

表 2 CTL*-FV の構文規則

3 既存研究と本研究の相違点

3.1 既存研究

Lacey ら [3] [4] が CTL-FV によって最適化を表せることを示した後，山岡ら [6] は既存のモデル検査器 SMV を用いて Lacey らの理論の一部である，無用コード除去の最適化を実装した．

その後，番ら [5] は自前のモデル検査器を使用することでモデル検査の効率化をはかり，同時に無用コード除去の最適化に加えてコピー伝播の最適化を実装した。

また，方 [7] では番らからさらにモデル検査器を改良し，無用コード除去・コピー伝播に加えて部分冗長性除去の最適化を実装した．

最近では，Warburton [8] が方と同様に無用コード除去・コピー伝播・部分冗長性除去の最適化を実装し，一部では方より性能の良い結果を出している．

3.2 既存研究の問題点

今までの既存研究の問題点として，以下の点が挙げられる．

- 手動で実装された最適化器に比べて，扱える最適化の種類が少ない
- 手動で実装された最適化器に比べて，最適化にかかる時間が長い

伝統的な最適化のアルゴリズムは，今までの研究の蓄積も多いため最適化の種類もモデル検査を用いた最適化の手法に比べて多い．また，そのアルゴリズムの多くは複雑で効率的な最適化変換を可能とするものなので，実際的な効率を考えるとモデル検査を用いた最適化器は今だ手動で実装された最適化器に比べて実用的とは言えないのが現状である．

3.3 本研究との相違点

以上の問題点を踏まえて，本研究では扱える最適化の種類を増やすという方向で改良を行い，具体的には算術的なコピー伝播について最適化ができるようにした。

一般的なコピー伝播とは，図 1 のように $x = a$ のような代入文があった場合，この代入文の後に現れる x を使用している式が以下の条件が成り立つ時に，この式の x の使用を a に置き換えることができるというものである．

- $x = a$ から x を使用している式までの全経路において，x の値が更新されていない
- $x = a$ から x を使用している式までの全経路において，a の値が更新されていない

既存研究で実装されているコピー伝播の最適化は，全てこのコード変換のことである。

一方算術的なコピー伝播とは，上にあげたような条件を満たしてはいないが，四則演算における $+$ や $*$ の可換性を考慮すれば，コピー伝播ができるというものである．例えば図 2 のように，x や y という変数だけを見ていたらコピー伝播が不可能であるが，

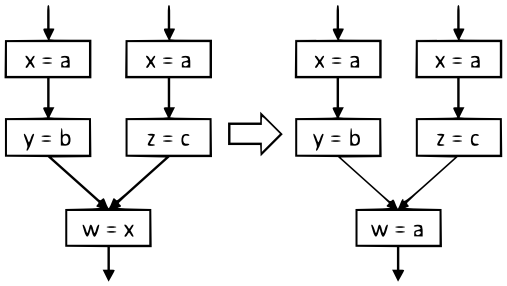

図 1　一般的なコピー伝播の例

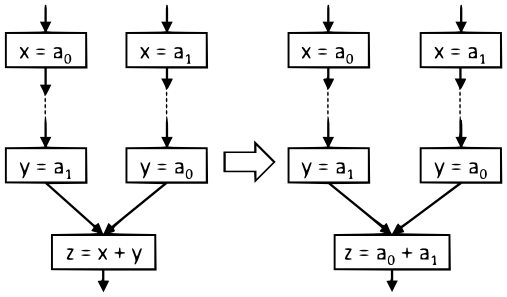

図 2　算術的なコピー伝播の例

$z = x + y$ という式において x と y の可換性を考慮すれば，どういった経路をたどろうとも z には $a_0 + a_1$ が代入されることがわかる．これを算術的なコピー伝播と呼ぶ．この算術的なコピー伝播は，既存研究では実装されていない．

4　最適化記述

4.1　最適化記述の構成

本システムの最適化記述は **MATCH**，**CONDITION**，**PROCESS** の 3 つから成る．

- **MATCH** はモデル検査の対象となる命令文の形を表す．
- **CONDITION** は命令が最適化されるとき満たすべき CTL*-FV 式を表わす．
- **PROCESS** は **CONDITION** の条件式が満たされるとき，どのように変換するかを表す．

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*-FV 式

PROCESS

$point_$文字列 : $Replace$ 式 → 式

以下は算術的なコピー伝播の最適化記述である:

MATCH

$z := x + y$

CONDITION

$point\_trans1 : trans(x) \wedge trans(a_0)$

$point\_trans2 : trans(y) \wedge trans(a_1)$

$point\_trans3 : trans(x) \wedge trans(a_1)$

$point\_trans4 : trans(y) \wedge trans(a_0)$

$point\_avail1 : point\_trans1\ U\ stmt(x := a_0)$

$point\_avail2 : point\_trans2\ U\ stmt(y := a_1)$

$point\_avail3 : point\_trans3\ U\ stmt(x := a_1)$

$point\_avail4 : point\_trans4\ U\ stmt(y := a_0)$

$point\_trace1 : point\_avail1 \wedge point\_avail2$

$point\_trace2 : point\_avail3 \wedge point\_avail4$

$point\_acp : \overleftarrow{A}(point\_trace1 \vee point\_trave2)$

PROCESS

$point\_acp : Replace\ x \rightarrow a_0$

$point\_acp : Replace\ y \rightarrow a_1$

4.2　MATCH 部

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束縛する．例えば，

MATCH

$v := s$

において v が変数，s が二項式であるとする．このとき $z := x + y$ は対象となるが，$x := y$ は対象とならない．式 $z := x + y$ が最適化の対象となったとき，$\{ v \mapsto z, s \mapsto x + y \}$ のように表す．この集合 $\{ v, s \}$ はプログラムの代入文の左辺が変数，右辺が二項式であるときのみ束縛の対象になり，v と s をばらばらに束縛することはない．

しかし，先にあげた算術的なコピー伝播の最適化記述では，MATCH 部に現れる変数は z, x, y だけであり，CONDITION 部に現れる自由変数 a_0, a_1 を束縛することができない．そこで，a_0, a_1 に入る可能性の

ある変数の組み合わせ全通りに対して，束縛してやればよい。

例えば図 3 を考える．この図のコードで代入文の左辺に来る変数の候補は x, y, z の三つで，それぞれの変数に代入されうる右辺の変数の集合は

$VAR_x = \{a_0,\ a_1,\ b_0\}$

$VAR_y = \{a_0,\ a_1,\ b_0,\ b_1\}$

$VAR_z = \{x + y\}$

となる．今 $z = x + y$ が MATCH 部によって束縛されたとすると，a_0, a_1 に束縛される可能性のある変数の集合は，

$VAR_x \cap VAR_y = \{a_0,\ a_1,\ b_0\}$

となり，これらの組み合わせ 6 通り全てに対して検査してやればよい．

以上の手順を一般化すると，この最適化記述を使って最適化によるコード変換を行う前に，最適化の対象であるコードの代入文に対して，左辺にくる変数（図 3 の例では x，y，z）ごとにその変数に代入されうる右辺の変数，または式の情報（図 3 の例では a_0，a_1，b_0，b_1，$x + y$）を解析しておき，いざ最適化を行う段階で MATCH 部により x，y に変数が束縛されたとき，先ほど解析しておいた変数の情報から a_0，a_1 に束縛される可能性のある変数の候補（図 3 の例では a_0，a_1，b_0）を絞って，その可能性の分だけ総当たり的検査を行うというものである．

4.3 CONDITION 部

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．条件式には PROCESS 部での処理と対応させるための式名を付ける．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．条件式の右側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．

4.4 PROCESS 部

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を記す．変換を表す処理文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つける．上の例では，式名 *point_acp* が表す条件式が成り立った命令 $z = x + y$ に対して，x を a_0 に，y を a_1 に置き換える．必要があればこの部分は一時変数を導入することができる．

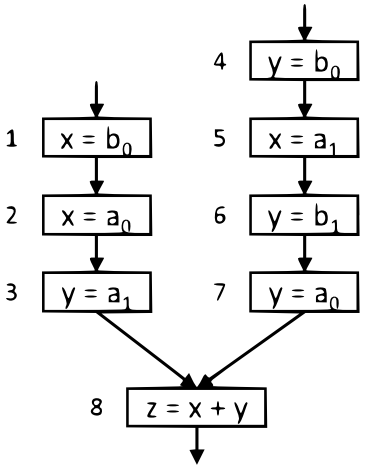

図 3 自由変数の束縛の例

5 最適化の流れ

本節では，本研究で提案する最適化の流れを図 3 を用いて説明する．最適化の大まかな流れは以下のようになっている．

1. 最適化の対象であるソースコードから，コード解析をして *trance* や *stmt* といった情報を得る
2. コード解析の結果から，ソースコードをモデル化し最適化記述の自由変数の束縛を行う
3. ソースコードのモデルと自由変数の束縛を行った最適化記述を用いて，モデル検査の手法でコード変換を行う

5.1 コード解析

図 3 において，コード解析の結果得られる各命令で真となる述語は以下である．

1. $\{stmt(x = b_0)\}$
2. $\{trans(b_0),\ stmt(x = a_0)\}$
3. $\{trans(b_0),\ trans(a_0),\ trans(x),$

$stmt(y = a_1)\}$

4. $\{stmt(y = b_0)\}$
5. $\{trans(b_0),\ trans(y),\ stmt(x = a_1)\}$
6. $\{trans(b_0),\ trans(a_1),\ trans(x),$
   $stmt(y = b_1)\}$
7. $\{trans(b_0),\ trans(a_1),\ trans(x),\ trans(b_1),$
   $stmt(y = a_0)\}$
8. $\{trans(b_0),\ trans(a_1),\ trans(x),\ trans(b_1),$
   $trans(y), trans(a_0),\ stmt(z = x + y)\}$

また，各代入文で左辺にくる変数とそれに対応する右辺にくる変数の集合は以下である．

$VAR_x = \{a_0,\ a_1,\ b_0\}$
$VAR_y = \{a_0,\ a_1,\ b_0,\ b_1\}$
$VAR_z = \{x + y\}$

5.2 ソースコードのモデル化

ソースコードのモデル化は，ソースコードの一つ一つの命令を 1 状態とするコントロールフローグラフ (Control Flow Graph,CFG) によって作られる．この CFG の各状態に，コード解析の結果を用いてどの述語が真となるかという情報を加えていったものが，モデル検査で使われるソースコードのモデルである．図 3 にコード解析で得られた情報を合わせたものが，ソースコードのモデルである．

5.3 モデル検査とコード変換

検査対象のモデルが出来たら以下の手順でモデル検査とコード変換を行う．

1. モデルの初期状態から最適化記述の MATCH 部に適合する命令を探す．無かったら終了．
2. MATCH 部 $z = x + y$ の x，y を束縛する変数の VAR 集合の積集合 $VAR_{x \cap y}$ を作る．
3. $VAR_{x \cap y}$ の要素の 2 つ組全組み合わせに対して，それぞれ CONDITION 部の検査式を作る．
4. 作った全ての CONDITION 部の検査式とモデルを用いてモデル検査を行い，真となったものは PROCESS 部に書かれている処理を行う．
5. 1 へ戻る．

図 3 では，命令文 8：$z = x + y$ が最適化記述の MATCH 部に適合し，自由変数 a_0，a_1 を束縛する同名の変数が CONDITION 部を満たす変数である．そこで，PROCESS 部に書かれている処理にしたがい，命令文 8：$z = x + y$ を $z = a_0 + a_1$ に変換する．

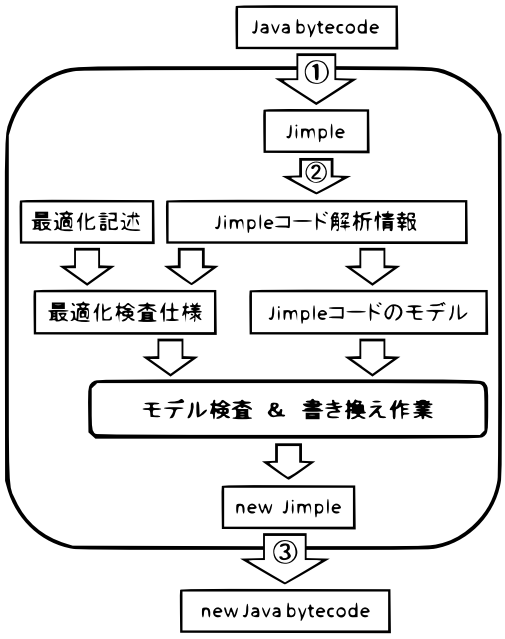

図 4 提案システムの概要

6 実装手法の提案

この節では 5 節で述べた最適化の流れに従って実装するシステムについて述べる．システムの概要図を図 4 に示す．

図 4 において，一番外側で囲んだものが今回提案するシステムの全体である．①，②，③ の矢印は，soot [2] と呼ばれる Java バイトコードの解析系である Java ライブラリによって行うものである．

Jimple [2] は三番地コードの中間言語の種類の一つで，システムに入力された最適化対象となる Java バイトコードを, soot によって Jimple 形式の中間言語に変換しこれに対してコード解析を行う．Jimple コード解析情報から Jimple コードのモデルを，Jimple コード解析情報の結果と算術的なコピー伝播の最適化記述からモデル検査に必要な最適化検査仕様を作成し，その結果をもとにモデル検査&書き換え作業部分で最適化記述の PROCESS 部に従ってコード変換

を行う．その後，コード変換によって最適化された新しい Jimple コードから新しい Java byte code に変換して，それを出力する．

7 まとめと今後の課題

本研究ではモデル検査を用いたコンパイラの最適化器生成において，時相論理 CTL*を用いることで従来では扱えなかった種類の最適化である算術的なコピー伝播が行えることを，Java 最適化器生成の手法を示すことで提案した．今後は，提案した手法を実装して今回提案した手法の性能を実験するとともに，算術的なコピー伝播の最適化が，最適化記述で表した CTL*の表現で正しく表わされていることを証明していくことが課題である．

謝辞 本研究で貴重なアドバイスを下さった伊藤宗平氏 (東工大) に感謝いたします．なお，本研究の一部は科学研究費補助金の援助を受けた．

参 考 文 献

[1] M. Reynolds. *An axiomatization of full computation tree logic*, Journal of Symbolic Logic, Vol.66, No.3, 2001, pp. 1011–1057.

[2] Vallée-Rai, Raja and Co, Phong and Gagnon, Etienne and Hendren, Laurie and Lam, Patrick and Sundaresan, Vijay : Soot - a Java bytecode optimization framework, CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, 1999, pp. 13.

[3] David Lacey , Neil D. Jones , Eric Van Wyk and Carl Christian Frederiksen : Proving correctness of compiler optimizations by temporal logic, Proc. the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL, 2002, pp. 283–294.

[4] David Lacey , Neil D. Jones , Eric Van Wyk and Carl Christian Frederiksen : Compiler Optimization Correctness by Temporal Logic, *Higher Order Symbol. Comput.*, Vol.17, No.3, 2004, pp 173–206.

[5] 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), 2004, pp. 65–75.

[6] 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.SIG13(PRO18), 2003, pp. 25–37.

[7] 方 玲 , 佐々 政孝 : 双方向 CTL による Java 最適化器の生成, 情報処理学会論文誌:プログラミング, Vol.48, No.SIG10(PRO33), 2007, pp. 76–89.

[8] Richard Warburton and Sara Kalvala : From Specification to Optimisation:An Architecture for Optimisation of Java Bytecode, CC 2009, LNCS 5501, 2009, pp 17–31