時相論理を用いたコンパイラ最適化器の実行の正しさの検査

佐原 聡一郎　佐々 政孝

コンパイラの最適化がプログラムの意味を変えず正しく動作することは非常に重要である．しかし，最適化には複雑なものも多く，コンパイラの最適化を正しく実装することは困難である．本論文では，プログラムが正しく最適化されたかどうかを時相論理を用いて最適化後に検査する手法を提案する．最適化によるプログラムの変形箇所が，プログラムの意味を変えないために満たすべき性質を時相論理で記述し，論理式で記述した性質を満たすかどうかを最適化後のプログラムをモデル検査することで検証する．この手法には，既存の複雑な最適化器にも適用でき，現実的な時間の範囲内での検査が可能であるといった利点がある．提案手法を実装し検査を実行した結果，COINS コンパイラの最適化器の未知のバグを発見することができた．

It is very important that the compiler optimization works correctly without changing the semantics of a program. However, because there are a lot of complex optimizations, it is difficult to implement them correctly in general. In this paper, we propose a technique for validating whether the optimization transformation into the program have been done correctly or not by using temporal logic after the execution of the optimizer. We describe the properties that points of optimization transformation have to satisfy by using temporal logic, and we check whether these points satisfy the formulas by model checking of the program. This technique can be applied to complex optimizers that already exist, and it can be executed in the realistic time. We implemented and executed this technique, and we found an unknown bug of an optimizer in the COINS compiler.

1 はじめに

1.1 背景

コンパイラのプログラム最適化は重要な技術であり，近年盛んに研究されている．ところが，最適化は複雑であるものが多く，アルゴリズムの設計や実装の段階など様々な箇所で，プログラムの意味を変えてしまうようなバグが混入しやすい．さらに，コンパイラ最適化のバグは，次の理由から発見が困難であり，一般に取り除くのが難しいという問題もある．

- 最適化が正常に終了したように見えても，最適化された目的プログラムは意図しない動作をするかもしれない．これは目的プログラムを実行するまで気付かない上，実行しても気づかないかもしれない．
- 目的プログラムの意味が変わってしまうバグを発見しても，最適化のどの箇所が誤った変形をしたのか，見分けるのが難しい．

このような背景から，最適化器のバグがないことを保証するための技術は非常に重要である．最適化器に完全にバグがないという保証をするのが無理でも，少なくとも最適化コンパイルされたプログラムの意味が変わっていないことを保証しなくてはならない．

コンパイラ最適化器の信頼性を向上させる既存の研究として，次のようなものがある．

- 最適化器そのものが正しいことを検証する．検証された最適化器は，任意のプログラムについ

Validating Correctness of Compiler Optimizer Execution by Using Temporal Logic

Soichiro Sahara Masataka Sassa，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.

コンピュータソフトウェア, Vol.16, No.5 (1999), pp.78–83.
[一般論文] xxxx 年 yy 月 zz 日受付.

て，その振舞いを変えることなく最適化できる．

- 最適化器の実行後に，プログラムの意味が変わらないような変形であったことを検査して確かめる．検査をしたプログラムは正しく最適化されたと判断できる．

前者の研究には，Lacey らの研究[12] や Lerner らの研究[13] [14] などがある．しかし，いずれも複雑な最適化を扱うことはできないようである．後者の研究には，Rinard らの研究[17] や Necula の研究[16] などがある．Rinard らの研究は，プログラム変形の正当性を厳密に示せるが，実際に適用する際にどの程度実用的かは明らかではない．Necula の研究は対照的に，非常に実用的ではあるが厳密な正当性の保証はない．

1.2 概要

本論文では，最適化器が行うプログラム変形が，プログラムの振舞いを変えていないかどうかを最適化実行後に検査する手法を提案する．最適化によるプログラムの変形箇所それぞれが，プログラムの意味を変えないために満たすべき性質を時相論理 CTL-FV [12] で記述しておき，最適化実行後にモデル検査を行うことで性質を満たすことを調べる．全ての検査に成功すれば，最適化は正しく実行され，プログラムの意味は変わっていないと判断できる．

我々の手法による利点は以下の通りである．

- 既存の最適化器に適用できる．
- Lacey らが扱える最適化よりも広い範囲の最適化を検査できる．
- バグを発見した際に原因の特定が容易である．
- 現実的な時間の範囲内で検査することができる．

本手法の適用性を確認するために，我々は COINS コンパイラ [6] に実装されているいくつかの最適化器に対して，本手法を適用し検査を行った．その結果，ループ不変式移動を行う最適化器の未知のバグを発見することができた．

提案手法の詳細は 4 節で，手法の有用性の考察は 6 節で述べる．

2 プログラムの最適化

プログラムの最適化とは，コンパイラが，プログラムの実行速度やサイズなどの性質を改善する目的で行うプログラムの変形のことを指す．最適化は，プログラムの性質を**解析**し，その結果に基づき**変形**するという方法が一般的である．

2.1 最適化の正しさ

最適化に最低限求められるのは，最適化前後でプログラムの振舞いを変えないことである．例えば，関数の戻り値が最適化前後で変わってしまったら，これはプログラムの振舞いが変わってしまう恐れがある．プログラムの振舞いが変わらないことを，**プログラムの意味が保存される**という．プログラムの意味が保存されないような最適化は，正しくない最適化である．

最適化の正しさとしては他に，その最適化による変形が確かにプログラムの効率を向上させているという性質を満たすということが挙げられる．しかし，

- 最適化を個別に行うより組み合わせた方がよい場合がある．
- プロファイル情報がない限り「保守的に」最適としか言えない．

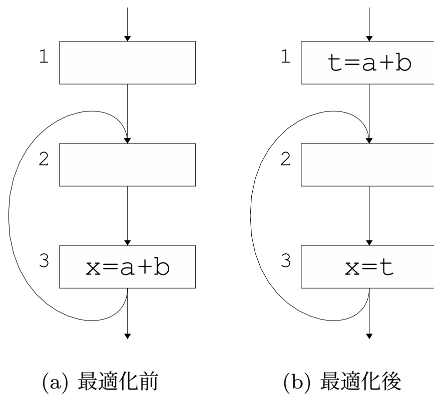
など，本当に効率が向上しているかどうかは難しい問題で，一般に示すことができない性質である．だが，プログラムの意味の保存の観点から見るとこれは必要条件ではなく，本論文では考慮しない．

以下，最適化が正しく実行されたとは，少なくとも最適化されたプログラムの意味が保存されたことをいう．

2.2 最適化の例とその正しさ

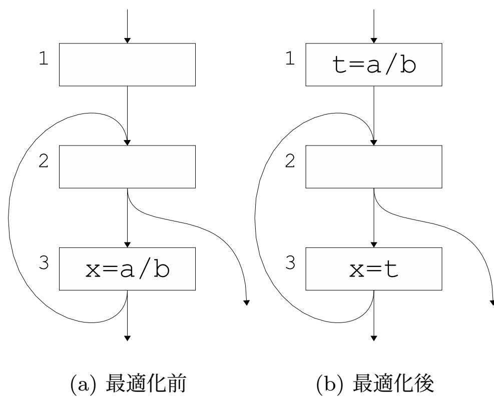
2.2.1 ループ不変式移動

ループ内で，常に値の変わらないような式を**ループ不変式**という．ループ不変式は例えば，式のオペランドが全てループ外で定義された変数または定数であるような式である．ループ不変式はループの外で計算しても値が変わらないので，ループに入る前で計算しておいて実行時の計算回数を減らすという最適化が考えられる．この最適化を**ループ不変式移動**という．



(a) 最適化前 (b) 最適化後

図 1 ループ不変式移動の例



(a) 最適化前 (b) 最適化後

図 2 ループ不変式移動の誤った最適化の例

図 1 はループ不変式移動の最適化の例である．図 1(a) で，a＋b はループ内で値の変わらない式であるので，図 1(b) のように a＋b をループの前で計算し，一時変数 t に代入しておく．ループ内での元の a＋b の使用は，一時変数 t に置き換えられる．このように式の計算位置を元の位置より前に移動することを，**式を巻き上げる**という．

2.2.2 ループ不変式移動の正しさ

図 1 のループ不変式移動の例では，最適化による変形は次の 2 箇所である．

- a＋b を巻き上げて，ループの前に t＝a＋b を挿入した．
- a＋b の元の使用を一時変数 t で置き換えた．

それぞれを**挿入点**，**置換点**と呼ぶ．

以下，ループ不変式移動がプログラムの意味を変えないために，それぞれの変形が満たすべき性質を説明する．

巻き上げた式の挿入 t＝a＋b を挿入してもプログラムの意味が変わらないためには，次の条件を満たす必要がある．

- 挿入した t＝a＋b の文で定義される t の値が，置換点以外で使用されることはない．

もし t の値が置換点以外で使用される場合，その場所での t の元々の値は a＋b の値以外であったはずなので，プログラムの意味が変わってしまうことになる．

巻き上げる式が a/b のように例外を発生する可能性のある式である場合は，もう少し注意が必要である．図 2 は，ループ不変式 a/b を移動した例だが，この最適化は誤りである．図 2(a) で，ノード 3 を通らない実行経路が考えられるので，例えば「b の値は 0 であるが，ノード 3 を通らずにループを抜けるため例外は発生しない」という実行が考えられる．しかし，図 2(b) のように a/b を巻き上げてしまうと，a/b は必ず計算されることになり，元々は起こらなかった例外が起こるようになってしまう．

文献 [1] などによると，このような例外を発生する可能性のある式を巻き上げて挿入する際には，次の条件も満たす必要がある．

- 元々計算のなかった実行パスに計算を挿入することがない†1

式の使用の置換 a＋b の使用を t で置換してもプログラムの意味が変わらないためには，次の条件を満たす必要がある．

- 置換点での a＋b と t の値が等しい．

言い換えると，挿入点と置換点の間に，t や a，b を定義する文がなければ，プログラムの意味は変わらない．

3 時相論理

本手法の検査には，時相論理 CTL-FV によるモデル検査を用いる．この節では，まず検査対象であるプログラムのモデルについて述べ，その後 CTL-FV の構文や意味について述べる．

†1 厳密には，例外を捕捉するシグナルハンドラの呼ばれる回数が変わるので，これは正しくないが，最適化でそこまで考慮するのは稀である．

3.1 プログラムのモデル

CTL-FV によるモデル検査を行うためには，プログラムを状態遷移モデルとして形式的に表現する必要がある．プログラムの最適化は制御フローグラフ上で行われるので，本手法で用いる状態遷移モデルも制御フローグラフ [1] [2] [15] [19] を基にするのが自然である．

定義 1 (制御フローモデル) 1 つの文をノードとする制御フローグラフ $G = (N, E)$ を考える．ただし，N はノードの集合，E はノード間の有向辺の集合とする．

原始命題全体の集合を AP とし，ノード $n \in N$ に対し，そこで成り立つ原始命題の集合 $L(n)$ を与える写像を $L\colon N \to 2^{AP}$ とする．三つ組 $M = (N, E, L)$ を**制御フローモデル**ということにする．これは Kripke 構造であり，N が**状態**の集合，$E \subseteq N \times N$ は状態間の**遷移**に相当する．状態 n と n' の間に遷移がある，すなわち $(n, n') \in E$ であることを $n \to n'$ とも表記する．以後，モデルとはこの制御フローモデルを指すこととする．

定義 2 (制御フローモデル上の経路) 制御フローモデル $M = (N, E, L)$ において，状態の無限列 $n_0, n_1, n_2, \ldots$ が，任意の $i \geq 0$ について $n_i \to n_{i+1}$ であるとき，この無限列を**無限経路**という．また，状態の有限列 $n_0, n_1, \ldots, n_m$ が，任意の $i\,(0 \leq i < m)$ について $n_i \to n_{i+1}$ かつ $\forall n \in N, \neg(n_m \to n)$ であるとき，この有限列を**有限経路**という．無限経路と有限経路を合わせて**経路 (パス)** という．

$n_1 \to n_2$ であるとき，この遷移関係の逆を**逆向きの遷移**といい，$n_2 \to^{\circ} n_1$ と表す．この逆向きの遷移に対して定義される経路を**逆向きの経路**という．

制御フローモデル $M = (N, E, L)$ における $L(n)$ の定義を表 1 に示す†2．

†2 正確な定義は，対象のプログラミング言語による．COINS の中間表現 LIR における定義は，文献 [7], [18] を参照されたい．

†3 ノード 1 に付くのは $node(1)$ である．

$L(n) =$

	$\{\ node(\mathtt{N})$	$\mid\ n = \mathtt{N}$ †3$\}$
$\cup$	$\{\ block(\mathtt{B})$	$\mid\ n$ は基本ブロック B の文 $\}$
$\cup$	$\{\ use(\mathtt{X})$	$\mid$ 変数 X は n で使用される $\}$
$\cup$	$\{\ def(\mathtt{X})$	$\mid$ 変数 X は n で定義される $\}$
$\cup$	$\{\ comp(\mathtt{E})$	$\mid$ 式 E は n で計算される $\}$
$\cup$	$\{\ trans(\mathtt{E})$	$\mid$ 式 E は n で変更されない，つまり E 中の変数は n で定義されない $\}$
$\cup$	$\{\ mark(\mathtt{M})$	$\mid\ n$ にはマーク M が付いている $\}$

表 1 $L(n)$ の定義

3.2 CTL-FV

本手法で用いる時相論理は，Lacey らが提案した **CTL-FV** である [12]．これは，分岐時間時相論理の一種である CTL [4] [5] を基にした論理で，次のような特徴がある．

- 逆向きの経路に関する限量子 $\overleftarrow{E}$，$\overleftarrow{A}$ を扱える．
- 原始命題を自由変数を引数にとる述語に一般化している．

直観的には，$\overleftarrow{E}$，$\overleftarrow{A}$ は通常の向きの経路限量子 E，A をそのまま逆向きの経路限量子としたものである．

我々が数ある時相論理の中から CTL-FV を採用した理由は次の 3 つである．

- 実行経路に沿った論理であり直観的に理解しやすい．
- 逆向きの経路も自然に扱うことができ，制御フローやデータフローの性質の記述に適している．
- 効率よくモデル検査を行うことができる．

CTL-FV の構文 CTL-FV の構文規則を表 2 に示す．ϕ は状態に関する式 (**状態式**) を導出する非終端記号，ψ は経路に関する式 (**経路式**) を導出する非終端記号であり，α は自由変数を引数とする述語である．

また，構文規則には現れないが，よく用いられる結合子を省略形として表 3 のように定義する．

CTL-FV の意味論 モデル M 上の状態 n で状態式 ϕ が成立することを，$M, n \models \phi$ と表す．また，経路 p で経路式 ψ が成立することを，$M, p \models \psi$ と表す．どちらも，M が自明であるときには省略して単

$\phi ::= true$	$\phi ::= E\ \psi$	$\psi ::= X\ \phi$
$\phi ::= false$	$\phi ::= A\ \psi$	$\psi ::= \phi\ U\ \phi$
$\phi ::= \alpha$	$\phi ::= \overleftarrow{E}\ \psi$	$\psi ::= \phi\ W\ \phi$
$\phi ::= \neg\phi$	$\phi ::= \overleftarrow{A}\ \psi$	
$\phi ::= \phi \wedge \phi$		

表 2 CTL-FV の構文規則

$\phi_1 \vee \phi_2$	$\equiv$	$\neg(\neg\phi_1 \wedge \neg\phi_2)$
$\phi_1 \rightarrow \phi_2$	$\equiv$	$\neg\phi_1 \vee \phi_2$
$EF\ \phi$	$\equiv$	$E\,(true\ U\ \phi)$
$AF\ \phi$	$\equiv$	$A\,(true\ U\ \phi)$
$\overleftarrow{E}F\ \phi$	$\equiv$	$\overleftarrow{E}\,(true\ U\ \phi)$
$\overleftarrow{A}F\ \phi$	$\equiv$	$\overleftarrow{A}\,(true\ U\ \phi)$

表 3 CTL-FV の糖衣構文

状態式

$n \models true$	iff	true
$n \models false$	iff	false
$n \models \alpha$	iff	$\alpha \in L(n)$
$n \models \neg\phi$	iff	not $n \models \phi$
$n \models \phi_1 \wedge \phi_2$	iff	$n \models \phi_1$ and $n \models \phi_2$
$n \models E\psi$	iff	$\exists p = n \rightarrow n_1 \ldots,\ p \models \psi$
$n \models A\psi$	iff	$\forall p = n \rightarrow n_1 \ldots,\ p \models \psi$
$n \models \overleftarrow{E}\psi$	iff	$\exists p = n \rightarrow^{\circ} n_1 \ldots,\ p \models \psi$
$n \models \overleftarrow{A}\psi$	iff	$\forall p = n \rightarrow^{\circ} n_1 \ldots,\ p \models \psi$

経路式 $(p = n_0 \rightarrow' n_1 \ldots,\ \rightarrow'$ is $\rightarrow$ or $\rightarrow^{\circ})$

$p \models X\phi$	iff	n_1 exists and $n_1 \models \phi$
$p \models \phi_1 U \phi_2$	iff	$\exists i \geq 0\ [\ n_i \models \phi_2$ and $\forall j\ [\ 0 \leq j < i$ implies $n_j \models \phi_1\ ]]$
$p \models \phi_1 W \phi_2$	iff	$(p \models \phi_1 U \phi_2)$ or $(\forall k \geq 0\ [\ n_k \models \phi_1$ and n_{k+i} exists $])$

表 4 CTL-FV の意味定義

に $n \models \phi$，$p \models \psi$ と表す．

CTL-FV の制御フローモデル上での意味の定義を表 4 に示す[†4]．

†4 厳密には，式中の自由変数をプログラム中のシンボルで全て束縛した式について意味は定義される．

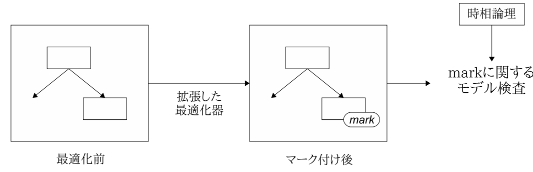


図 3 提案手法の概要

4 提案手法

4.1 提案手法の概要

本論文では，1.2 節でも述べた通り，最適化器が行うプログラム変形が，プログラムの意味を変えていないかどうかを最適化実行後に検査する手法を提案する．

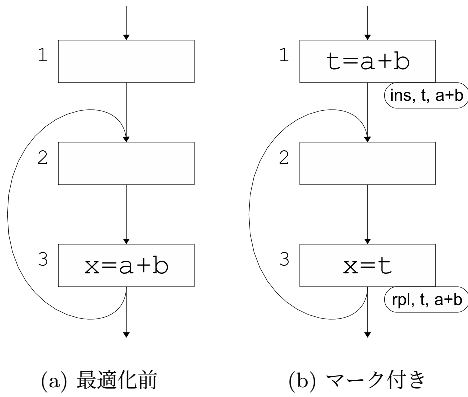
図 3 は，提案する手法の概要を表す．提案手法の検査は，次のように行う．

- プログラムの意味を変えないために変形箇所ごとに満たすべき性質を，時相論理 CTL-FV であらかじめ記述しておく．
- 全ての変形箇所が記述した式を満たすかどうか，最適化実行後にモデル検査により調べる．
- 全ての検査が成功すれば，最適化は正しく実行されたと判断する．検査に失敗した場合，その変形は誤りであり最適化器のバグであると判断する．

最適化実行後に変形箇所の検査を行えるように，変形箇所には，最適化中に変形に応じたマークを付ける．マーク付けは，最適化器を拡張することで行えるようにする．また，この拡張はアスペクト指向プログラミングを用いることで，最適化器自身のソースコードをほとんど改変することなく容易に行うことができる．

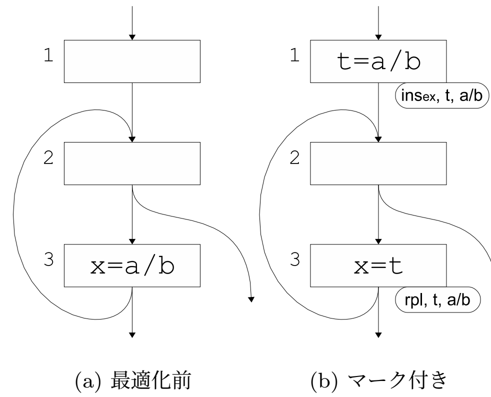
本手法は，最適化器そのものの正しさを検証するものではなく，最適化器の実行結果が正しかったかどうかを検査する手法である．

また，本手法で検査するのはあくまで変形箇所が CTL-FV 式を満たすかどうかであり，プログラムの意味が変わらないことを厳密な意味で検証するものではない．しかし，Lacey らが文献[12] で提案したプログラムの意味の保存を証明する手法は，本手法にお



(a) 最適化前　(b) マーク付き

図 4 ループ不変式移動の例 (マーク付き)



(a) 最適化前　(b) マーク付き

図 5 誤ったループ不変式移動の例 (マーク付き)

いても利用可能であると考えられる．プログラムの意味の保存の証明は今後の課題である．

4.2 提案手法の実現手順

提案手法は，次の 5 つのステップからなる．

■ 事前準備

1. 最適化の正しさの条件を検査仕様として最適化ごとに記述しておく．
2. 最適化中に変形を行う箇所にマークを付けられるように，既存の最適化器を拡張しておく．

■ 最適化実行前

3. プログラムをモデル化する．

■ 最適化実行中

4. 最適化によるプログラムの変形箇所にマークを付ける．

■ 最適化実行後

5. モデル検査を行う．

各ステップの詳細は以降で順次説明していく．説明の際，具体例として 2.2 節で説明したループ不変式移動の最適化を例に用いる．ループ不変式移動による変形箇所は，挿入点と置換点の 2 つであった．図 4(b) はそれぞれの変形箇所にマークを付けた最適化後のフローグラフである．挿入点には $(\mathtt{ins}, \mathtt{t}, \mathtt{a}+\mathtt{b})$，置換点には $(\mathtt{rpl}, \mathtt{t}, \mathtt{a}+\mathtt{b})$ というマークを付けて，どのような変形がどこで行われたかを分かるようにしている．

ステップ 1: **検査仕様の記述** 本手法ではまず，最適化による変形箇所がプログラムの意味を保存するために満たすべき検査仕様を，CTL-FV を用いてあらかじめ記述しておく．この記述を**検査仕様**，または単に**仕様**ということにする．変形の仕方や変形の満たすべき性質は当然最適化ごとに異なるので，仕様は最適化ごとに個別に記述する必要がある．

4.1 節でも述べたように，最適化による変形箇所にはそれぞれマークが付けられるので，仕様はマークの種類ごとに記述するのが自然である．よって，マークを m，それに対応する CTL-FV 式を ϕ とすれば，仕様はこれらの組 $\langle m, \phi\rangle$ と表される．

以下，図 4 のループ不変式移動に対して，検査仕様の記述を考えていく．

巻き上げた式を挿入した箇所 2.2 節で述べたように，$\mathtt{t} = \mathtt{a} + \mathtt{b}$ を挿入した点での正しさは次の通りであった．

- 挿入した $\mathtt{t} = \mathtt{a} + \mathtt{b}$ の文で定義される $\mathtt{t}$ の値が，置換点以外で使用されることはない．

これは言い換えると，

- 挿入点からの経路で「$\mathtt{t}$ が再定義されずに，置換点以外で使用されるような経路」は存在しない．

である．一時変数 $\mathtt{t}$ を t，式 $\mathtt{a} + \mathtt{b}$ を e という自由変数で表し，置換点にはマーク $(\mathtt{rpl}, t, e)$ が付いていることに注意すると，これを表す CTL-FV 式は，

$$\neg E\,(\neg def(t)\ U\ (use(t) \wedge \neg mark(\mathtt{rpl}, t, e))) \quad (1)$$

である．よって，式 (1) を ϕ_1 とすれば，満たすべき仕様は $\langle(\mathtt{ins}, t, e), \phi_1\rangle$ となる．

$\mathtt{a/b}$ のように例外を発生する可能性のある式を巻き

上げて挿入する場合は，2.2 節で述べたように，

- 元々a/b の計算のなかった実行パスに a/b の計算を挿入することがない．

という条件も必要である．例外を発生することのない式の巻き上げと区別するために，除算のような式を巻き上げて挿入した箇所には，図 5(b) のようにマーク $(\mathtt{ins}_{\mathrm{ex}}, t, e)$ を付けることにする．上記の条件を言い換えると，

- このマークの付いた挿入点からの全ての経路では，置換点に到達する．

である．a/b を e という自由変数で表すと，これを表す CTL-FV 式は，

$$A\,(\mathit{true}\ W\ \mathit{mark}(\mathtt{rpl}, t, e)) \qquad (2)$$

である[†5]．よって，式 (1) を ϕ_1，式 (2) を ϕ_2 とすれば，満たすべき仕様は $\langle(\mathtt{ins}_{\mathrm{ex}}, t, e), \phi_1 \wedge \phi_2\rangle$ となる．

式の使用を置換した箇所 2.2 節で述べたように，a + b の使用を t で置換した点での正しさは次の通りであった．

- 置換点での a + b と t の値が等しい．

これは言い換えると，

- 置換点から始まる全ての逆向きの経路では，a + b と t の値が変わることなく，挿入点に必ず到達する．

である．挿入点のときとは異り，a/b のように例外を発生する可能性のある式の置換についても，これと同じ条件で十分である．

一時変数 t を t，式 a + b を e という自由変数で表すと，挿入点はマーク $(\mathtt{ins}, t, e)$ かマーク $(\mathtt{ins}_{\mathrm{ex}}, t, e)$ のいずれかが付いているノードであるので，これを表す CTL-FV 式は，

$$\overleftarrow{A}\ ((\neg \mathit{def}(t) \wedge \mathit{trans}(e))\ W\ (\mathit{mark}(\mathtt{ins}, t, e) \vee \mathit{mark}(\mathtt{ins}_{\mathrm{ex}}, \mathtt{t}, \mathtt{e}))) \qquad (3)$$

である[†5]．よって，式 (3) を ϕ_3 とすれば，満たすべき仕様は $\langle(\mathtt{rpl}, t, e), \phi_3\rangle$ となる．

†5 U 演算子ではなく W 演算子を使う理由は，無限ループなどの終端に至らない経路を考慮しないためである．

ステップ 2: 既存の最適化器の拡張 本手法では，最適化実行時の変形箇所へのマーク付けを実現するために，最適化器のソースコード中でプログラムの変形を行う部分のコードの箇所に，マークを付けるためのコードを挿入しておく必要がある．これは最適化器ごとに人の手で行わなくてはならないが，一般に変形を行う部分のコードはあまり多くない．

このコードの挿入は，アスペクト指向プログラミングを利用することで，既存の最適化器のソースコードの直接の変更を最小限に抑えることができる．ソースの改変を行ったのは，コード挿入箇所を指定するためにやむなく中身が空のメソッドコールを挿入したのみである．我々は，Java のアスペクト指向システム GluonJ [3] を利用して実装した．

ステップ 3: プログラムのモデル化 本手法では，3.1 節で述べたように，制御フローグラフを制御フローモデルとしてモデル化する．このモデルは，コンパイラの内部表現の制御フローグラフをそのまま利用することも考えられ，制御フローグラフ上で直接モデル検査することも考えられる．しかし，この方法は次のような問題がある．

- 文を削除する無用命令除去のような最適化で，グラフのノードごと削除してしまった場合，最適化後のフローグラフだけではどこを削除したのか特定できない．削除した箇所が特定できないと，変形箇所の検査を行うことができない．

文を削除する際に，ノードを削除せず「skip」「nop」など何もしないことを表す文に置き換える実装の最適化器もあるだろうが，一般にはノードごと削除する最適化器が多いと思われるので，この問題に対処する必要がある．

この問題に対する最も単純かつ確かな解決方法は，最適化前に制御フローグラフのコピーを取り，それをモデルとして扱うことである．最適化器が元のプログラムに変形を加えるたびに，コピーのモデルにも同様の変形を加える．ただし，ノードを削除する変形に対しては，文を「skip」とすることで対応させる．

我々の実装も，この方法でノードの削除の問題に対応した．

ステップ 4: 変形箇所のマーク付け 最適化中に，図 4(b) のように変形に応じたマークを付けていく．これは，最適化器を拡張したことにより自動で行われる．

ステップ 5: モデル検査 最適化の実行の終了後には，検査対象のモデルと検査仕様の集合が得られている．これらを用いて，プログラムの変形箇所が意味を保存するための性質を満たしているかどうか検査する．

仕様 $\langle m, \phi \rangle$ は，

- マーク m の付いたノードでは ϕ が成立するべき．

という意味なので，これは次の式と同じ意味である†6．

$$\forall n \in N,\ n \models mark(m) \rightarrow \phi \tag{4}$$

モデル検査は，仕様を式 (4) の形の 1 つの CTL-FV 式に変換して行う．

ところで，自由変数を含む CTL-FV 式では，そのままではモデル検査することができないので，実際にプログラムに表れるシンボルで CTL-FV 式中の自由変数を束縛し，自由変数を含まない式とする必要がある．この束縛は，最適化中に付けたマークを利用することで行う．

図 4(b) の例では，2 つのマークを付けた．マーク $(\mathtt{ins}, \mathtt{t}, \mathtt{a}+\mathtt{b})$ に対応する CTL-FV 式は式 (1) である．実際に付けたマークと式 (1) の対応を取ると，$\mathtt{t}$ と t，$\mathtt{a}+\mathtt{b}$ と e が対応しているので，これで式 (1) を束縛する．すると式 (5) が得られる．

$$\neg E\,(\neg def(\mathtt{t})\ U\ (use(\mathtt{t}) \wedge \neg mark(\mathtt{rpl}, \mathtt{t}, \mathtt{a}+\mathtt{b}))) \tag{5}$$

同様にして，$(\mathtt{rpl}, \mathtt{t}, \mathtt{a}+\mathtt{b})$ に対応する式 (3) を束縛すると，式 (6) が得られる．

$$\overleftarrow{A}\ ((\neg def(\mathtt{t}) \wedge trans(\mathtt{a}+\mathtt{b}))\ W\ (mark(\mathtt{ins}, \mathtt{t}, \mathtt{a}+\mathtt{b}) \vee mark(\mathtt{ins_{ex}}, \mathtt{t}, \mathtt{a}+\mathtt{b}))) \tag{6}$$

これら 2 つの式と式 (4) から，最終的にモデル検査で用いる論理式は次の 2 つとなる．

$$\models \quad mark(\mathtt{ins}, \mathtt{t}, \mathtt{a}+\mathtt{b}) \rightarrow \phi'_1 \tag{7}$$

$$\models \quad mark(\mathtt{rpl}, \mathtt{t}, \mathtt{a}+\mathtt{b}) \rightarrow \phi'_3 \tag{8}$$

ただし，ϕ'_1 は式 (5)，ϕ'_3 は式 (6) である．

文献[10] によると，CTL-FV 式により最適化器を作成する場合は，自由変数の束縛の組み合わせの数がモデル検査にかかる時間に大きく影響するとあるが，本手法では実際に集めたマークを元に束縛を行うので，組み合わせの数を非常に少く抑えることが可能である．

モデル検査はどのように行ってもよいが，我々は，文献[4] による古典的な CTL モデル検査に基づいたアルゴリズムを用いて実装した．

5 実際の最適化器への適用

COINS コンパイラ[6] のバックエンド上では，中間表現 LIR[7] を対象とした多くの最適化が実装されており，特に SSA 形式上での最適化が充実している[20]．

我々は，LIR を対象とする次の最適化器について本手法を適用し検査を行った．

■ SSA 形式上
- ループ不変式移動
- 条件分岐を考慮した定数伝播
- コピー伝播
- 共通部分式除去
- 無用命令除去

■ 通常形式上
- Lazy Code Motion†7 [11]

以下，SSA 形式上でのループ不変式移動と条件分岐を考慮した定数伝播への適用について詳細を述べる．

5.1 SSA 形式

SSA 形式とは，変数の定義がプログラムの字面上唯一となるようにしたプログラムの表現形式である[2] [9] [15]．SSA 形式は，プログラムの最適化に有利

†6 n を省略して，$\models mark(m) \rightarrow \phi$ としても同じ意味である．

†7 COINS 標準モジュールには含まれていない．

な形式といわれている．

SSA 形式は，次の有用な性質を持つ．

- 各変数の使用には，唯一の定義が到達する．
- 制御フローグラフ上のノード n で，変数 v の異なった定義が合流するとき，ϕ **関数**を n の先頭に挿入することで，それらの到達する v の値を区別することができる．

5.2 ループ不変式移動

COINS には，SSA 形式上でのループ不変式移動が実装されている．この最適化は，2.2 節や 4.2 節で説明した通常形式上でのループ不変式移動と基本的には同じであり，

- ループ不変式を巻き上げてループの前に挿入する．
- ループ不変式の使用を一時変数で置換する．

という 2 つの変形からなる．ただし，プログラムの意味を保存するためにこれらの変形箇所が満たすべき性質は，4.2 節とは多少異なり，SSA 形式の特徴を利用したものとなる．

巻き上げた式を挿入した箇所 SSA 形式では，変数の二重定義は許されないので，$t = e$ を挿入する際には，式 (1) に代わって t が他のノードで定義されていないことが正しさの条件となる．$t = e$ を挿入したノードを n とすると，挿入点のノード n で成り立つべき CTL-FV 式は，

$$\neg EF\ (\neg node(n) \wedge def(t)) \wedge \neg \overleftarrow{E}F\ (\neg node(n) \wedge def(t)) \quad (9)$$

となる．これはつまり，n 以外に t を定義するノードがない，ということを表す式である．

また，e が `a/b` のように例外を発生する可能性のある式である場合は，通常形式の場合と同様に，式 (2) も正しさの条件として必要となる．

式の使用を置換した箇所 式 e を一時変数 t で置換した箇所は，通常形式と同様，式 (3) で十分である．しかし，SSA 形式では t は挿入点以外では定義されないことが式 (9) により保証されるので，式 (3) 中の $\neg def(t)$ はなくてもよく，置換点のノードで成り立つべき CTL-FV 式は，

$$\overleftarrow{A}\ (trans(e)\ W\ (mark(\texttt{ins}, t, e) \vee mark(\texttt{ins}_{\texttt{ex}}, \texttt{t}, \texttt{e}))) \quad (10)$$

としてもよい．

以上のように，本手法は通常形式上の最適化でも SSA 形式上の最適化でも，性質の違いに注意するだけで，同じように扱うことができる．

5.3 条件分岐を考慮した定数伝播

COINS には，条件分岐を考慮した定数伝播 [23] という SSA 形式上での強力な定数伝播アルゴリズムが実装されている．

図 6 は，条件分岐を考慮した定数伝播を行ったプログラムの例である．図 6(a) のグラフを入り口から順に辿ると，`t1` は常に `0` となり，ブロック `B4` は到達不能であることなどが分かる．この最適化は，データフロー方程式による解析ではなく，このように制御フローグラフを辿り，値がどうなるかを調べる**抽象実行 (Abstract Interpretation)** という枠組みの中で行われる最適化である．

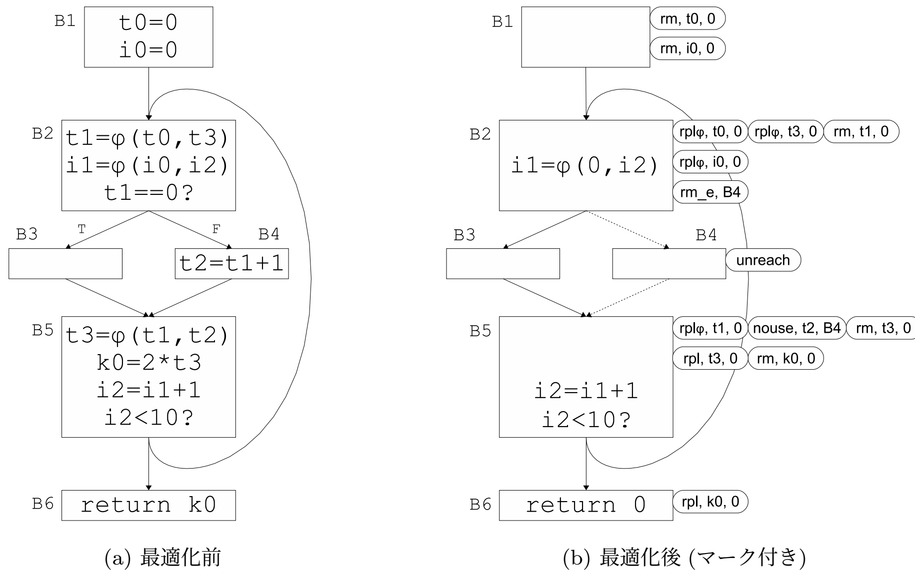
最適化後は図 6(b) のようになる．`t0` や `t1`，`k0` など，値が常に定数となることが分かる変数の使用が定数に置き換えられ，その変数への代入文が削除される．また，`B4` のような到達不能ブロックの文やそこへの分岐が削除される．

Lacey らの手法では，この条件分岐を考慮した定数伝播の最適化を行うことはできないと思われるが，本手法では，この最適化が正しく実行されたかどうかを検査することができる．その理由は 5.3.2 節で述べる．

5.3.1 条件分岐を考慮した定数伝播の正しさ

条件分岐を考慮した定数伝播の変形箇所をまとめると，次の 5 種類となる．

1. 変数 x への代入文で x の値が定数 c になるので，この代入文を削除した箇所．マーク $(\texttt{rm}, x, c)$ を付ける．
2. 変数 x の使用を定数 c で置換した箇所．マーク $(\texttt{rpl}, x, c)$ を付ける．
3. ブロック b への分岐を削除した箇所．マーク



(a) 最適化前　　(b) 最適化後 (マーク付き)

図 6　条件分岐を考慮した定数伝播の例

$(\texttt{rm_e}, b)$ を付ける．

4. 到達不能ブロック中のノードの文を削除した箇所．マーク (unreach) を付ける．
5. ϕ 関数中で，到達不能ブロック p に関連付けられた引数を除去した箇所．マーク $(\texttt{nouse}, x, p)$ を付ける．

以下で，これらの変形の正しさについて簡単に述べ，記述した仕様を示す．

定数代入となる代入文を削除した箇所 代入文 $x = e$ の右辺 e が定数 c と評価できれば，x の値は c となり以後の全ての x の使用は c に置き換えることができるので，この代入文は削除することができる．

評価の方法は，右辺 e が ϕ 関数かどうかにより異なる．

- e が ϕ 関数の場合，e で使用されている全ての変数が，定数値 c を取る変数であるか，到達不能ノードから制御が流れてきた際に評価される変数である場合，e は c と評価される．
- e が ϕ 関数でない場合，e で使用されている変数は全て定数値をとり[†8]，その式の計算結果が c となる場合，e は c と評価される．

なお，x が定数値 c を取る変数ならば，この文のあるノードにはマーク $(\texttt{rpl}, x, c)$ が付いており，また，ϕ 関数中の変数 x が到達不能ブロック p からの変数である場合には，この文のあるノードにはマーク $(\texttt{nouse}, x, p)$ が付いていることに注意する．

この評価方法は，関数 $eval$ として次のように定義される．ただし，$\top$ は値が未定であることを表す記号，$\bot$ は値が不定であることを表す記号とする [23]．また，c, c_1, c_2 は定数を，x, y などは変数を，op は算術演算を表す．

$$eval(c) = c$$

$$eval(x) = \begin{cases} \top & \text{marked } (\texttt{nouse}, x) \\ c & \text{marked } (\texttt{rpl}, x, c) \\ \bot & \text{otherwise} \end{cases}$$

$$eval(x\,op\,y) = \begin{cases} c & eval(x) = c_1 \\ & \wedge\, eval(y) = c_2 \\ & \wedge\, c_1\,op\,c_2 = c \\ \bot & \text{otherwise} \end{cases}$$

$$eval(\phi(x_1, ..)) = \begin{cases} c & \forall x_i, eval(x_i) = c \\ & \vee\, eval(x_i) = \top \\ \bot & \text{otherwise} \end{cases}$$

†8　$0 \times x$ のような場合の x は，この限りではない．

$eval(e) = c$ ならば，この文 $x = e$ は削除することができ，ノードにはマーク $(\mathtt{rm}, x, c)$ が付けられる．このノードで検査する式は，

$$eval(e) = c \tag{11}$$

となる．この式は，厳密には 3.2 節で定義した CTL-FV ではないが，マーク付けの際にノードの文とその時点で付いているマークから検証できるものであり，モデル検査による検証を必要としない．よって，このマークに関する検証は，マーク付けの際に $eval(e)$ を計算することにより行うことにする．

変数の使用を定数で置換した箇所 変数 x の値が定数 c となるようなノードでの x の使用は，c に置換できる．あるノード n で x の値が c になるのは，先行パスに $x = c$ と等価な文のノード m があり，かつ m が n を支配しているときである．$x = c$ に相当する文のあるノード m では，この文は削除されてマーク $(\mathtt{rm}, x, c)$ が付いているので，これを利用する．到達不能ブロックを除去することも考えると，n の全ての先行パスには，n を支配するノード m か到達不能ブロックが存在すればよいので，このノード n で検査する式は，

$$\overleftarrow{A}\,(true\ W\ (mark(\mathtt{rm}, x, c) \vee mark(\mathtt{unreach}))) \tag{12}$$

となる．

ϕ 関数の引数 x を定数 c に置換する際は，もう少し注意が必要である．x が先行ブロック p から制御が流れてきたときに評価される引数だとすると，p の最後の文は $x = c$ に相当する文のあるノード m に支配されている必要があるが，p 以外の先行ブロックの最後の文は m に支配されていなくてもよい．例えば図 6 では，ブロック B2 の $\mathtt{t1} = \phi(\mathtt{t0}, \mathtt{t3})$ で，t3 を 0 に置換している．t3 は，B5 から制御が流れてきたときには値 0 として定義されていなくてはならないが，B1 からきたときには未定義でも構わない．以上から，ノード n のあるブロックを b とし，b の中には n の前に別の ϕ 関数のノードがあるかもしれないことを考慮すると，

- n からの全ての逆向きのパスでは，ブロック b のノード (これは ϕ 関数のノード) がいくつか続いた後に別のブロック (これは b の先行ブロック) のノードになる．そのブロックが p であるならば，そのブロックの最後のノードでは式 (12) が成り立たねばならない．

という条件が成り立てば，x の c による置換は正しいといえる．

ϕ 関数の引数 x の置換には特別にマーク $(\mathtt{rpl}_\phi, x, c)$ を付けることにすると，マーク $(\mathtt{rpl}_\phi, x, c)$ を付けたノード n で検査する式は，

$$\overleftarrow{A}\,(block(b)\ U\ (block(p) \to \phi)) \tag{13}$$

となる．ただし，ϕ は式 (12) である．

分岐を削除した箇所 条件分岐文の条件式 e が，真または偽と評価できれば，コンパイル時に分岐を決定することができ，必要ない分岐を削除することができる．これも，定数代入文を削除する箇所と同様に，$eval$ 関数と同様の方法によりマーク付けの際に検査を行えばよく，モデル検査は必要ない．

到達不能ブロックの文を削除した箇所 ブロック b への辺が存在しないとき，b は到達不能ブロックである．b 中のノードには (unreach) というマークを付ける．b への分岐が削除された場合，b は到達不能ブロックとなる．到達不能ブロックの中のノードは，

- ブロックの先頭ならば，一つ前のノードは分岐を削除した文のあったノード (マーク $(\mathtt{rm_e}, b)$ が付いている) であったはずである．
- ブロックの先頭でなければ，一つ前のノードは文を削除したノードである．文を削除したノードは，到達不能ノード (マーク (unreach) が付いている) であったか，もしくは定数代入文として削除したノード (マーク $(\mathtt{rm}, x, c)$ が付いている) であったはずである．

のいずれかの条件を満たす．これより，マーク (unreach) を付けたノードで検査する式は，

$$\overleftarrow{A}X\ (mark(\mathtt{rm_e}, b) \vee mark(\mathtt{unreach}) \vee mark(\mathtt{rm}, x, c)) \tag{14}$$

となる．

ϕ 関数中の到達不能ノードからの引数を除去した箇所 ϕ 関数の引数は全て，いずれかの先行ブロックに関連付けられている．ϕ 関数のある引数 x が先行ブロック p に関連付けられているとすると，p が到達不能ブロックであるか，p の最後のノードがこの ϕ 関数のあるブロック b への分岐を削除した箇所であれば，この引数は除去できる．この ϕ 関数のあるブロックを b とすると，マーク $(\mathtt{nouse}, x)$ を付けたノードで検査する式は，式 (13) のときと同様の考察により，

$$\overleftarrow{A}\ (block(b)\ U\ (block(p) \rightarrow (mark(\mathtt{unreach}) \vee mark(\mathtt{rm_e}, b))) \quad (15)$$

となる．

5.3.2 本手法で検査可能である理由

条件分岐を考慮した定数伝播は，制御フローグラフ上のデータフロー方程式を解くことで行える最適化ではなく，プログラムの抽象実行に基づいた最適化である．フローグラフのデータフロー方程式を解く方法では，全ての辺は通るかもしれないという前提があるため，`t1` などが任意の実行で定数となることが解析できない．よって，`B4` が到達不能であるということも解析することができない．

データフロー方程式は，μ 計算と同等の計算力があるといわれている [21]．μ 計算より計算力の低い CTL-FV モデル検査をプログラムの解析に用いる Lacey らの手法ではこの最適化を行うことができない．

一方本手法では，最適化器が図 6(b) のように変形した結果が本当に正しいか検査するだけであり，これは CTL-FV によるモデル検査で可能である．つまり，ノード `B4` は到達不能で変数 `t1` は常に `0` という解析結果が与えられているので，それが正しいかどうかを検査する式なら 5.3.1 節で示したように記述可能である．

6 実験と考察

この節では，提案手法を COINS の最適化器に適用して行った実験を基に，本手法の有用性について考察する．

6.1 未知のバグの発見

我々は，COINS に実装されている SSA 形式上でのループ不変式移動の最適化器を本手法により検査したことで，未知のバグを発見した．これは，図 5 のように例外を発生する可能性のある式の巻き上げを誤って行ってしまうバグであった．

このバグは，SPEC CPU2000 ベンチマーク [22] の 254.gap というプログラムをコンパイル・検査している際に発見した．このプログラムは COINS の SSA 最適化を行っても，正常に実行される目的プログラムが生成されていた．実際，目的コードを実行しても `0` による除算は起きず，今までバグは発見されていなかったが，本手法を用いたところ，このような潜在的なバグを発見することができた．これは，本手法の特筆すべき点である．

このバグは次のように発見された．

1. 検査を実行したところ，マーク $(\mathtt{ins_{ex}}, t, e)$ に関する仕様のモデル検査で，反例が検出された．この仕様は，式が巻き上げ可能であることを検査する仕様である．
2. 最適化器のソースコードの対応箇所を見てみると，例外を発生する可能性のある式の考慮がなされていなかった．

本手法において，モデル検査で反例が検出されるということは，検査した仕様に関連する変形箇所に誤りがあったということを示す．つまり，本手法には最適化のどの変形にバグがあるかが反例から直接分かり，バグの発見の際の原因の特定が容易であるという特徴がある．

6.2 最適化器の拡張箇所の数

本手法では，最適化によるプログラムの変形箇所にマークを付けられるように最適化器を拡張する必要があった．一般に，最適化による変形箇所は少ないので，この拡張はさほど手間にはならない．また，アスペクト指向プログラミングを利用することで，実際に最適化器のソースコードを書き換える必要のある箇所は非常に少なくなる．

†9 本研究では完全な実装には至っていないが，これ以上拡張箇所も改変箇所も増えることはない．

最適化器	行数	拡張	改変
ループ不変式移動	357	2	0
条件分岐を考慮した定数伝播	1143	5	2
コピー伝播	143	3	2
共通部分式除去	498	7	1
無用命令除去[†9]	480	3	0
Lazy Code Motion	1259	2	0

表 5 最適化器ごとの拡張箇所の数

CPU	Intel Pentium 4 CPU 2.80GHz
OS	Linux 2.6.17-13msmp
JavaVM	1.5.0_08
Heap Size	256Mbyte
Stack Size	2048Kbyte
COINS	1.4.1
Benchmark	SPEC CPU 2000 version 1.2

表 6 実験環境

表 5 に各最適化器ごとの拡張箇所の数を示す．それぞれの列は次の意味である．

- 行数 ··· 最適化器のソースコードの行数
- 拡張 ··· 最適化器の拡張箇所の数
- 改変 ··· 最適化器のソースコードを書き換えた行数

表 5 によると，最適化器の行数に対して，マークを付けるための拡張箇所が少なかったことが分かる．また，拡張箇所はプログラムの変形を行う箇所となるが，これらは特徴的であることが多く[†10]，発見するのは最適化に対する一般的な知識のあった著者にとっては容易であった．

しかし，検証者に最適化の知識があまりない場合などは，拡張箇所を発見するのは容易ではないと思われる．拡張箇所を自動的に，もしくは簡単に発見する手法は今後の課題である．

6.3 検査仕様の記述コスト

本手法では，各最適化器のマークの種類ごとに検査仕様を記述する必要があった．ここで，検査仕様の記述にどの程度の労力がかかるか考察する．

記述量 検査仕様はマークの種類ごとに記述するが，マークの種類の数はすなわち表 5 の拡張箇所の数に当たる．既に述べたように，拡張箇所の数は少ない．また，検査仕様はほとんどが 1〜3 行で記述できる程度で，多くても 5 行程度であった．以上から，仕様の記述量は少ないといえる．

記述の容易性 本手法の検査仕様を記述するためには，最適化器ごとのアルゴリズムや特徴をきちんと理解しておく必要があるので，誰でも簡単に記述できるというものではない．最適化器の製作者でなければ，その最適化がどのような動作をするのかを，コンパイラの仕様書や最適化研究の論文，もしくはソースコードなどから読み取る必要がある．

このように，検査仕様の記述は誰でも容易に行えるものではないが，記述量が少なくて済むことを考えると，最適化に比較的詳しく，時相論理に慣れていれば，それほど難しくはないと思われる．実際，表 5 の 6 つの最適化器の検査仕様は，最適化器の正確な動作をソースコードから読み取り理解する時間を含めて 3 週間ほどで記述した．仕様書だけでなくソースコードも読む必要があったのは，仕様書と実装の間にいくつかのずれがあったためである．

6.4 検査効率の実験

4.1 節で述べたように，本手法は各最適化を実行するたびに検査を行う手法である．よって，検査時間が現実的な時間内で行われる必要がある．この節では，本研究で検査を実装した最適化について，検査にどの程度の時間がかかったかを計測した実験について述べる．実験を行った環境は表 6 の通りである．

実験は，検査を行わない場合のコンパイル時間と，検査を行う場合のコンパイル時間をそれぞれ計測し，比較した．計測した項目は次の 4 つである．

A 最適化器実行にかかった時間の合計

B 本手法の検査付き最適化器の実行にかかった時間の合計

C 検査なしの場合の総コンパイル時間

D 本手法の検査付き総コンパイル時間

最適化は，オプション O2 を指定する際に行われる SSA 最適化を用いた (オプション O2 で行われる SSA 最適化は，共通部分式除去，条件分岐を考慮した定数伝播，質問伝播大域値番号付けと部分冗長性除去，演算の強さの軽減と判定の置き換え，コピー伝播，ルー

†10 例えば，フローグラフを操作するメソッド呼び出しなどである．

	A	B	$\frac{B}{A}$	C	D	$\frac{D}{C}$
175.vpr	6.85	43.56	6.35	325.79	487.49	1.49
181.mcf	1.78	8.99	5.05	47.86	98.45	2.05
186.crafty	14.21	380.29	26.74	303.51	834.97	2.75
197.parser	5.92	29.49	4.97	364.93	504.66	1.38
254.gap	27.17	400.42	14.73	1541.74	2303.60	1.49
255.vortex	21.32	112.57	5.27	1175.71	1675.16	1.42
256.bzip2	1.25	11.74	9.34	108.85	149.58	1.37
300.twolf	25.03	475.61	19.00	459.80	1234.32	2.68
171.swim	0.52	12.46	23.70	10.21	27.59	2.70
172.mgrid	0.81	17.10	21.02	19.12	42.81	2.23
177.mesa	29.53	540.70	18.30	1654.83	2622.32	1.58
179.art	0.53	3.66	6.89	30.43	43.41	1.42
183.equake	1.00	23.69	23.68	50.63	88.18	1.74
188.ammp	9.88	140.19	14.18	281.46	554.08	1.96
	sum(A)	sum(B)	$\frac{\text{sum(B)}}{\text{sum(A)}}$	sum(C)	sum(D)	$\frac{\text{sum(D)}}{\text{sum(C)}}$
sum	145.86	2200.52	15.08	6374.93	10666.66	1.67

表 7 検査の有無によるコンパイル時間の比較 (単位：秒)

プ不変式移動，無用命令除去，である)†11．

計測結果を表 7 に示す．これによると，最適化の実行時間の総和は 15.08 倍の増加，総コンパイル時間の総和は 1.67 倍の増加であった†12．これは決して高速とはいえないが，検査によるコンパイル時間の増加は，最大でも 177.mesa で 27 分が 43 分になる程度であったことも考慮すると，十分現実的な時間であるといえる．

また，本手法の検査は次のように高速化が可能であると思われる．本手法では，4.2 節で説明したように，検査仕様の CTL-FV 式の自由変数を束縛してモデル検査する．これは，1 つの検査仕様から，付けたマークの個数だけ検査式が生成されることを意味する．1 つの仕様から生成された式は，それぞれを個別に検査するよりも，データフロー方程式のビットベクトル法 [15] [2] のように，まとめて検査することが可能であると考えられる．データフロー方程式の計算がビットベクトル法により大幅に高速化したように，本手法の検査も大幅な高速化が期待できる．

以上から，本手法は現実的な時間内で検査を行え，十分実用的であるといえる．

7 関連研究

最適化の正しさに関する研究は多く為されている．この節では，本手法に関連の深いいくつかの研究と本手法の比較を述べる．

Lacey らは，時相論理 CTL-FV を用いて最適化のデータフロー解析を記述し，モデル検査により最適化を実行する手法を提案した [12]．また，この手法により実現した最適化がプログラムの意味を保存するものであることを，いくつかの最適化について，簡単な方法で証明できることを示した．証明された最適化器は，常に正しく実行されることが保証され，バグのない最適化器といえる．しかし，データフロー方程式のみで記述できる，比較的簡単な最適化しか実現できないようである．例えば，5.3 節で説明した条件分岐を考慮した定数伝播は，本手法では扱えるが，Lacey らの手法では扱うことができない．

Lerner らは，時相論理を基にした独自のドメイン記述言語により最適化を記述し実行するシステムを提案した [13] [14]．これは，Lacey らの研究に似ているが，定理証明器を用いて最適化の正しさの証明をほぼ自動的に行えるという特徴がある．Lerner らのシステムも，Lacey らの手法と同様，複雑な最適化は扱えないようである．

†11 Lazy Code Motion はまだ試用版であったため，含めないこととした．

†12 B − A = D − C とならないのは，GluonJ による織り込みの時間や JIT コンパイルのタイミング，GC の実行時間などの影響が考えられる．

Necula は，最適化前後のプログラムの意味が等しいことを，記号的に推測し評価することで検査する手法を提案した[16]．この検査手法は，原理的には最適化に依存しないので，最適化ごとに検査仕様を作成する本手法より実現コストが低いといえる．また，検査時間の効率も高いようである．しかし，この手法では推測や評価に失敗することがあり，厳密なプログラムの意味の保存は保証されないという欠点もある．本手法でも，厳密なプログラムの意味の保存は保証されないが，厳密な証明を与えることもおそらく可能であると思われる．また，彼らの手法では，バグを発見した際の原因の特定は本手法ほど容易ではないと思われる．

Rinard らは，最適化前後のプログラムの意味が等しくなるための変数の値の条件を割り出し，最適化の実行後に，プログラムの意味が保存されているかどうか証明する手法を提案した[17]．この手法は厳密なプログラムの意味の保存を保証できるようだが，具体的なアルゴリズムの記載がなく，どの程度実用的な手法なのか不明である．

8 まとめ

我々は，CTL-FV のモデル検査を利用して，最適化器の実行がプログラムの意味を保存するものであったかどうか検査する手法を提案した．提案した手法は，最適化によるプログラムの変形箇所それぞれが，プログラムの意味を変えないために満たすべき性質を CTL-FV で記述しておき，最適化実行後にモデル検査を行うことで性質を満たすことを調べるというものであった．

この手法により，様々な既存の最適化器の検査が行えた．また，提案手法を実装し実験したところ，最適化器の未知のバグを発見することができた．検査にかかる時間もコンパイル時間と比較して現実的な時間内であった．

9 今後の課題

今後の課題としては，主に次の三つが挙げられる．

厳密な証明 本手法で記述した最適化の変形箇所の仕様は，それ単独で最適化の正しさを保証するものではない．多くのものは，直観的に明らかであったり，他の文献ですでに証明がなされていたりするが，厳密には，対象プログラムの意味論を正確に定義し，その上で「仕様を満たすならば正しい」という証明を与えるべきである．

これは，Lacey らは手で行い，Lerner らは定理証明器を用いて行ったものである．本手法でも同様に行えるものと期待される．

より複雑な最適化器への適用 COINS SSA 最適化モジュールには，帰納変数の演算の強さの軽減と判定の置き換え[8] や質問伝播に基づく大域値番号付けと部分冗長除去[20] といった，非常に複雑な最適化が実装されている．これらの最適化器には，バグがあるのではないかという疑いが以前からある．これらの最適化器の検証を本手法により行いたい．

検査の高速化 本論文の主題はモデル検査の高速化ではなかったので，細かいアルゴリズムの高速化までは踏み込まなかった．しかし，6.4 節で述べたような手法で，大幅な検査時間の短縮が望めると思われる．

拡張箇所の自動発見 6.2 節でも触れたが，検証者に最適化の知識がない場合などは，最適化器中の拡張箇所を発見するのは困難であると考えられる．拡張箇所を自動的に発見する，もしくは容易に発見する手法については，今後の課題である．

謝辞

本研究の一部は，科学研究費補助金，大川情報通信基金の補助を受けた．

参 考 文 献

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D.: *Compilers: Principles, Techniques, and Tools 2nd ed.*, Addison-Wesley Longman Publishing Co., Inc., 2006.

[2] Appel, A. W. and Palsberg, J.: *Modern Compiler Implementation in Java 2nd ed.*, Cambridge University Press, 2002.

[3] Chiba, S., Nishizawa, M., and Kumahara, N.: GluonJ Home Page. `http://www.csg.is.titech.ac.jp/projects/gluonj/`.

[4] Clarke, E. M., Emerson, E. A., and Sistla, A. P.: Automatic verification of finite-state concurrent systems using temporal logic specifications, *ACM Trans. Prog. Lang. Syst.*, Vol. 8, No. 2(1986),

pp. 244–263.

[5] Clarke, E. M., Grumberg, O., and Peled, D. A.: *Model Checking*, The MIT Press, 1999.

[6] COINS Project: COINS Home Page. http://www.coins-project.org/.

[7] COINS Project: COINS プロジェクト LIR 仕様書, 2002. http://www.coins-project.org/spec/lir.pdf.

[8] Cooper, K. D., Simpson, L. T., and Vick, C. A.: Operator strength reduction, *ACM Trans. Prog. Lang. Syst.*, Vol. 23, No. 5(2001), pp. 603–625.

[9] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K.: Efficiently computing static single assignment form and the control dependence graph, *ACM Trans. Prog. Lang. Syst.*, Vol. 13, No. 4(1991), pp. 451–490.

[10] 方玲, 佐々政孝: 双方向 CTL による Java 最適化器の生成, 日本ソフトウェア科学会第 *23* 回大会（*2006* 年度）論文集 *1B-1*, 2006.

[11] Knoop, J., Rüthing, O., and Steffen, B.: Optimal code motion: theory and practice, *ACM Trans. Prog. Lang. Syst.*, Vol. 16, No. 4(1994), pp. 1117–1155.

[12] Lacey, D., Jones, N. D., Wyk, E. V., and Frederiksen, C. C.: Compiler Optimization Correctness by Temporal Logic, *Higher Order Symbol. Comput.*, Vol. 17, No. 3(2004), pp. 173–206.

[13] Lerner, S., Millstein, T., and Chambers, C.: Automatically proving the correctness of compiler optimizations, *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, ACM Press, 2003, pp. 220–231.

[14] Lerner, S., Millstein, T., Rice, E., and Chambers, C.: Automated soundness proofs for dataflow analyses and transformations via local rules, *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM Press, 2005, pp. 364–377.

[15] 中田育男: コンパイラの構成と最適化, 朝倉書店, 1999.

[16] Necula, G. C.: Translation validation for an optimizing compiler, *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, ACM Press, 2000, pp. 83–94.

[17] Rinard, M. and Marinov, D.: Credible Compilation with Pointers, *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, Jul 1999.

[18] 佐原聡一郎: 時相論理を用いたコンパイラ最適化器の検証, 修士論文, 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, 2007.

[19] 佐々政孝: プログラミング言語処理系, 岩波書店, 1989.

[20] 佐々研究室: 静的単一代入形式最適化システム外部仕様書, 2006. http://www.is.titech.ac.jp/~sassa/coins-www-ssa/japanese/ssa-external-japanese.pdf.

[21] Schmidt, D. A.: Data flow analysis is model checking of abstract interpretations, *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM Press, 1998, pp. 38–48.

[22] Corporation, S. P. E.: SPEC Home Page. http://www.spec.org/.

[23] Wegman, M. N. and Zadeck, F. K.: Constant propagation with conditional branches, *ACM Trans. Prog. Lang. Syst.*, Vol. 13, No. 2(1991), pp. 181–210.