

自動的等価性差分の抽出による SSA コンパイラ最適化の正しさの検証

Fang Ling^{†1} 佐々 政孝^{†1}

目的コードの効率を向上させる最適化はコンパイラの重要なフェーズである。しかし最近の進んだ最適化の多くは複雑なソフトウェアであるため、最適化に誤りがあることは稀ではなく、そのときその原因を突き止めることが難しい。本論文では、最適化前後の差分を自動的に抽出し、時相論理に基づいて変数等の等価性を評価することにより SSA 形式上のコンパイラ最適化の正しさを検証する手法を提案する。まず、変形箇所がプログラムの意味を保つために満たすべき性質を時相論理の CTL 式で記述しておく。次に、最適化前後の SSA 形式の中間言語を比較照合し、最適化による変形を抽出する。それらの結果に従ってモデルを生成し、すべての変形がその種の変形に応じた CTL 検査式を満たすかどうかをモデル検査により検査する。本手法により COINS コンパイラの最適化の誤りや曖昧な変形をいくつも発見した。本手法では、検証者は最適化アルゴリズムを知る必要がなく、テストコードを実行する必要もない点に特徴がある。

Verification of Compiler Optimization on SSA form by Finding Value Equality Difference

LING FANG^{†1} and MASATAKA SASSA^{†1}

Optimization is a very important phase of compilation. It can improve the performance of a program by a large factor. However, many recent optimizers are complex, which may compromise program correctness. It is essential that the compiler optimizer is implemented without changing the semantics of a program. Guaranteeing the correctness of optimization for realistic programs is still an open problem. In this paper, we propose a technique for validating the optimization transformation of the program by checking if the optimization transformations are equivalent transformations. First, we define the semantic equivalence of every kind of transformation and formalize them using CTL formulas. Then, we compare the intermediate programs in Static Single Assignment (SSA) form before and after optimization and extract the differences. After analysis and modelling, every transformation is checked against the corresponding CTL formula via model checking. We do not need to know the details of the optimization algorithms and execution of the test program. We applied this technique to the optimizer in the COmpiler INfraStructure (COINS) compiler. It worked with great efficiency, and found several bugs and ambiguous transformations.

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。最適化は目的プログラムの時間および空間効率を向上させるプログラム変換を行う。

コンパイラ最適化器は、入力プログラムの振舞いを変えてはいけない。これは最適化器に対する最低限の要求である。最近の進んだ最適化の多くは複雑なソ

フトウェアであるため、一般に、アルゴリズムの設計や実装の段階など様々な箇所、プログラムの意味を変えてしまうような誤りが混入しやすい。コンパイル段階でエラーになったり、実行結果が明らかに違ったりするバグもあるが、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。また、最適化の目的に反して冗長性を混入するバグもある。さらに、バグがどこで混入したものなのかが見極めにくい。

このような背景から、最適化器に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。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。

^{†1} 東京工業大学大学院情報理工学研究所

Graduate School of Information Science and Engineering, Tokyo Institute of Technology

- (1) 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て、その振舞いを変えことなく最適化できる。
- (2) テストプログラムを用いて、最適化器がそのテストで引き起こした中間表現の変化などを検証する。
- (3) テストプログラムを用いて、最適化によって変化したテストコードの実行を確かめる。

(1)には Lacey らの研究¹⁴⁾¹⁵⁾ や Lerner らの研究¹⁶⁾ などがある。これらの研究は証明できるドメイン言語を定義することによって、理論上の厳密性を保証するが、実際にあまりバグが混入しにくい簡単な最適化しか扱えず、実用性に欠ける点がある。

(2)には Necula ら¹⁸⁾ や佐原ら²¹⁾ の研究がある。Necula らの研究は最適化前後のプログラムにおいて、対応した call 文、jump 文及び return 文が等価であれば、プログラムの変換が等価であると仮定する。それらの対応点まで、記号的に抽象実行を行い、結果がすべて等価であれば、プログラムの変換も正しいとする。この検査手法は、高速で現実的であり、どんな最適化器にも使える。しかし、文献 18) に書いてあるように、この研究は抽象実行をしながら記号評価を行い、最適化前後のプログラムにおいて、等価であるべき対応点が等価であるかどうかを判定する。判定不能の場合、推測によるため、厳密なプログラムの意味の保存を保証できない。この手法は複雑な最適化に対しては精度が低いと思われる。

佐原らの研究²¹⁾ は最適化器の振舞いを検査の対象とし、最適化器を実行しながら、最適化のアルゴリズムと関係ある箇所にマークを付け、それらのマーク間の関係が時相論理式を満たすかどうかを検査する。この手法は最適化器を GluonJ³⁾ によって拡張する必要があるため、処理が煩雑で効率が良くない。最適化器と最適化器のアルゴリズムに依存するので、検証者が最適化のアルゴリズムを理解する必要があり、各フェーズに対して定式化が必要である。関係が複雑な場合は記述できない。この研究も厳密なプログラムの意味の保存を保証できない。

(3)の研究には、Jaramillo らの研究¹³⁾ などがある。この研究は最適化前後のプログラムを交互に実行して行き、対応する変数に対する値が一致しているかどうかを検証する手法である。しかし、理論上の裏づけが弱い。例えば、ある実行に対して、最適化前後の変数の値が一致したとしても、すべての実行に対して、この変数の値が一致することが保証できない。逆に、乱数を使うプログラムのような場合は、正しい変換によ

る最適化前後の変数の値は異なるかもしれない。また、トレースデータが大きすぎ、検証時間が長い、などの欠点がある。

これらのほかに型システムによって、型の検査による検証を行うものもある。それは意味的な検証とは大きな違いがあるため、最適化を厳密に検証したい時には役に立たない。

1.2 本研究の概要

本論文は前節の (2) の研究に属する。本手法は、最適化前後のプログラムの差異を自動的に抽出し、最適化による変形がプログラムの意味を変えない等価変形であるかどうかを検査する。値の等価性の判定は記号評価により、等価変形の検査は時相論理に基づく。中間コードや目的コードを実行する必要がなく、検証者は最適化に使われているアルゴリズムを理解しなくてよいという特徴がある。

本手法の流れは次のようになる。まず、変形箇所がプログラムの意味を保つために満たすべき性質を時相論理 CTL²⁷⁾ で記述しておく。次は最適化前後の中間言語プログラムを解析し、変数等の等価性の情報や定義 - 使用関係などを含めたプログラムの性質及び最適化による変形を解析し、モデルを生成する。最後にモデル検査を行い、最適化実行後に変形箇所がその変形に対応した検査式を用いて等価変形であったかどうかを調べ、間違った変形を報告する。

前節の (2) の研究と同様、本手法は与えられたプログラムに対する最適化器の正当性を検証する。

我々の手法による利点は以下の通りである。

- 静的単一代入形式の中間表現を用いて全部の変形を抽出し、検査するため、厳密性がある。バグを発見した際に原因の特定が容易である。
- 既存の最適化器に手を入れることなく、最適化器や実装のアルゴリズムに依存しないため、検証者が最適化アルゴリズムを理解しなくてよい、これは従来研究にはない特徴である。また、定式化しやすい。目的プログラムを実行する必要がない。
- SSA 形式上での最適化の各フェーズについて、条件付き定数伝播²⁸⁾ や質問伝播に基づく大域値番号付けと部分冗長性除去²⁶⁾ といった非常に複雑な最適化器を含み、精確に検証できた。これは従来研究では行えない。冗長性も発見できるのは本研究が初めてである。検証時間も現実的である。本手法を用いて、COINS コンパイラ⁵⁾ に実装されている SSA 最適化器に対して、本手法を適用して検査を行った。その結果、いくつかの誤りや曖昧な変形が発見できた。

本論文の構成は次の通りである．2節はプログラムの SSA 形式及び SSA 形式上での最適化について説明する．3節は検証に使う時相論理について述べる．提案手法の詳細は4節で，とくに本手法にとって重要な変数の等価性の評価は4.3節で述べる．本手法について実験した結果は5節で述べる．6節，7節はそれぞれ関連研究，考察と将来課題について述べる．8節ではまとめを述べる．

2. プログラムの静的単一代入形式 (SSA 形式) と SSA 形式上での最適化

この節では，図1に示した典型的な2つのプログラムを例として，プログラムの SSA 形式，SSA 形式上の最適化について述べる．

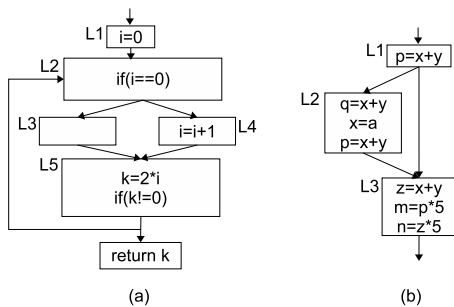


図1 プログラムの例
Fig.1 Example of the programs

2.1 SSA 形式 (静的単一代入形式)

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⁹⁾．SSA 形式では，変数の使用に到達する定義が一意に決定でき，プログラムの最適化に有利な形式といわれている．図1のプログラムを SSA 形式に変換すると，図2のようになる．

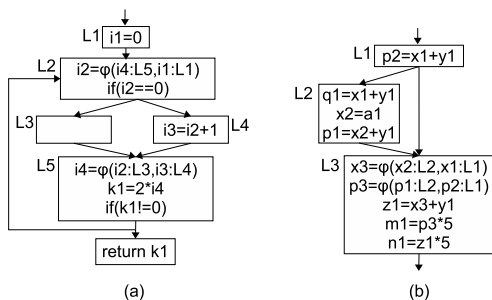


図2 図1のプログラムの SSA 形式
Fig.2 SSA form of the programs of figure 1

通常形式のプログラムを SSA 形式に変換するには，

- 変数の名前の付け換え
- ϕ 関数の挿入

を行う． ϕ 関数とは，元のプログラムで同じであった変数の定義が合流するところに挿入される仮想的関数である．図2(b)のL3にある $\phi(x2:L2, x1:L1)$ などが ϕ 関数である．この ϕ 関数は「L2 から来た場合は $x2$ を，L1 から来た場合には $x1$ を返す」関数である．

これを少し説明すると，図2(b)のL3において， x の値は，L2 から来た場合はL2内で定義された $x2$ の値，L1 から来た場合はL1内で有効な $x1$ となる．L3の入口に ϕ 関数による代入文 $x3 = \phi(x2:L2, x1:L1)$ を挿入することで，L3での x の値は $x3$ と決定することができる．

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 の異なる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，到達する v の値を区別する．

2.2 SSA 形式上でのプログラムの最適化

プログラムの最適化は，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．

プログラムの性質を解析する手法は色々あるが，制御フローグラフ上のデータフロー方程式を解くことにより行うのが一般的である．図3(b)はデータフロー方程式を解くことによって図2(b)のプログラムに対して，質問伝播に基づく大域値番号付けと部分冗長性除去 (PREQP)²⁶⁾ を行った例である．L3の $z1$ と $n1$ への代入が最適化されている．

しかし，条件分岐を考慮した定数伝播 (CSTP)²⁸⁾ のような，制御フローグラフ上のデータフロー方程式を解いたのでは解が得られず，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)⁸⁾ を行う最適化もある．図3(a)は図2(a)のプログラムに条件分岐を考慮した定数伝播を適用した例である．

最適化器は，最適化前後でプログラムが意味的に変わっていないことを保証しないといけない．プログラムが意味的に変わっていないことを，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，本当に最適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．

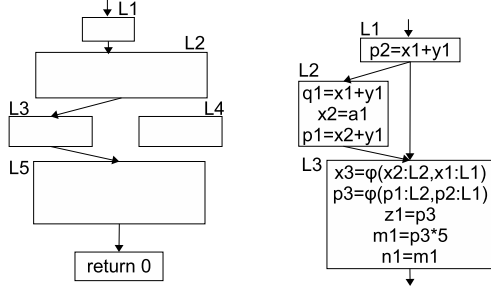


図 3 SSA 形式上の最適化の例
Fig. 3 Optimization on SSA form

さて, CSTP²⁸⁾ と PREQP²⁶⁾ は SSA 形式上で有力な最適化であるが, 非常に複雑で検証が必要である. 図 3(a) に示したように最適化の後 k は常に 0 であり, $L4$ は到達不能な基本ブロックになる. Lacey らや Lerner らは簡単な最適化しか検証できないため, CSTP の検証もできない. 佐原ら²¹⁾ の研究はできたと主張しているが, 循環論証と考えられ, 実際に検証しようとする, 検証器が無限循環に陥る.

SSA 形式は最適化には有利である. しかし, 図 2(b) の $L3 : z1 = x3 + y1$ が $L1 : p2 = x1 + y1$ と $L2 : p1 = x2 + y1$ とそれぞれ違う式になったため, 図 1(b) の中で, $L3 : z = x + y$ が $L1 : p = x + y$ と $L2 : p = x + y$ で計算され, 冗長な計算であるという情報が SSA 形式への変換の後に失われてしまった. PREQP はこういう冗長性を除去するアルゴリズムである. PREQP も非常に複雑で従来研究では検証できる研究がまだない.

3. 時相論理

この節では CTL 論理について説明する. Computational Tree Logic (CTL)²⁷⁾ は分岐時相論理 (branching-time temporal logic) の一つである. 経路限量子として, $A=(All)$, $E=(Exist)$ がある. 時相演算子として, $U=(Until)$, $X=(neXt)$, $G=(Globally)$, $F=(Future)$, $R=(Release)$ 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. Kripke 構造で無限に遷移する性質を分岐経路で記述することができる. プログラムの制御フローグラフ (CFG)^{1),17),23)} を Kripke 構造に対応させ, プログラムの抽象実行を CTL の無限経路に対応させると, CFG で書かれたプログラムの性質を CTL で記述することができるため, 本研究では CTL を採用した. この節では, CTL の構文や意味について述べる. 図 4(a) は Kripke 構造であり, 図 4(b) はその上を遷移する無限 CTL 木の例である. CTL 木は Kripke 構造

の開始状態から遷移関係に沿って無限に展開した木構造である.

本論文では, 有向辺の方向を時間順とみなす, 時間順で先に現れることを先行と呼ぶ.

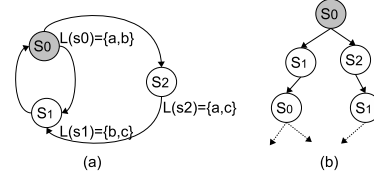


図 4 Kripke 構造 (a) と構造に対応した S_0 からの CTL 木 (b)
Fig. 4 Kripke structure (a) and its CTL (infinite) trees (b) for S_0

3.1 構文規則

CTL の構文規則は以下の通りである.

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

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⁾.

$$\begin{aligned} AX\phi &= \neg EX(\neg \phi) \\ EF\phi &= E[true U \phi] \\ AG\phi &= \neg EF(\neg \phi) \\ AF\phi &= A[true U \phi] \\ EG\phi &= \neg AF(\neg \phi) \\ A[\phi_1 R \phi_2] &= \neg E[\neg \phi_1 U \neg \phi_2] \\ E[\phi_1 R \phi_2] &= \neg A[\neg \phi_1 U \neg \phi_2] \end{aligned}$$

3.2 意味論

CTL の意味論は Kripke 構造によって与えられる. Kripke 構造 K は三つ組 (S, R, L) であり, S は状態の集合, $R \subseteq S \times S$ は遷移関係, $L : S \rightarrow 2^{Prop}$ は各状態にその状態において真となる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 $p = (s_0 \rightarrow s_1 \rightarrow s_2 \dots)$ 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 $\exists path(s = s_0 \rightarrow s_1 \rightarrow s_2 \dots)$:

$(s_i)_{i \geq 0} \models \psi$

$s \models A\psi$ iff $\forall path(s = s_0 \rightarrow s_1 \rightarrow s_2 \dots)$:

$(s_i)_{i \geq 0} \models \psi$

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 $\exists k \geq 0$:

$[s_k \models \phi_2 \wedge \forall i : [0 \leq i < k \Rightarrow s_i \models \phi_1]]$

Lacey ら¹⁵⁾ に従って \bar{A} と \bar{E} を導入する．これらは逆向きすなわち過去を表し， A と E と対称的な意味論を持つ．

3.3 循環論証の問題

論理によって証明する場合，詭弁論理を避けるように注意しないといけないことがある．例えば，変形が満たすべき条件のなかに，直接的もしくは間接的に，この変形が引き起こした性質が入ってはいけない．前提が結論の根拠となり，結論が前提の根拠となるものを循環論証 (*circular reasoning*) という．時相論理において循環論理を避けるには，条件になるものは証明の結論より CTL 木において時間順に先に成り立たないといけない．

4. 提案手法

この節は提案手法の概要及び提案手法の実現手順の詳細について述べる．

プログラムの意味を変えない変形を等価変形という．本論文では，最適化が行うプログラム変形が，等価変形であるかどうか，すなわちプログラムの意味を保っているかどうかを 3 節で述べた時相論理によって検査する手法を提案する．これは以下の手順で実現する．

- (1) 各種の変換に対して等価変換の意味論を記述する検査式を CTL 論理を用いて定式化する．
- (2) 最適化前後の中間コードの変形を抽出し，最適化前の中間コード及び最適化後の変形された中間コードに基づいてモデルを生成する．その際，変数の値が等しい合同集合の計算を行う．中間コードを実行する必要はない．
- (3) すべての変形について，その変形の種類に応じ

た検査式を満たすかどうかを検査する．満たさない場合，バグとして報告する．

図 5 は，提案する手法の概要を表す．各ステップの詳細は以降で順次説明していく．変形に対する検査式が変形の内容に対応するように，ステップ (2) の変形の抽出を説明しながら，ステップ (1) の変換に対応した CTL 式の定式化について説明する．

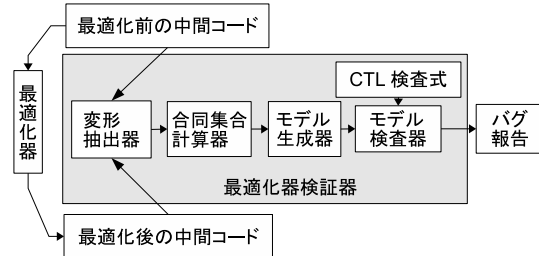


図 5 提案手法の概要

Fig. 5 Outline of the verifier

中間コードの命令文の BNF を表 1 に記す．記号の意味は標準的な C 言語の意味に準ずる． \bar{E} は一連の式を意味し， $[E]$ はポインタが指したメモリの中身の意味する． ϕ 関数は ϕ_1 のように，基本ブロック毎に区別する．

4.1 モデル

3.2 節に述べたように，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) である．本研究は時相論理を適用するため，中間コード及び最適化による変形を時相論理に適用したモデルを作成する必要がある．最適化前後の中間コードをそれぞれ π と π' と記す．モデルは $M_\pi = (S, R, L)$ ， $M_{\pi'} = (S', R', L')$ とする．本研究は最適化前後の中間コードの比較照合の結果を検証するため，最適化前の中間コードに基づいてモデル生成し，最適化後の中間コードの変形をこの中間コードのモデル上に反映する必要がある．SSA 形式上の最適化は，中間コードの基本ブロックのラベルを変えない，最適化前の変数の名前も変えない，追加の一時変数が最適化前の変数と重複しない，という特徴があるため，以下の手順で最適化の変化を反映したモデルを生成することができる．

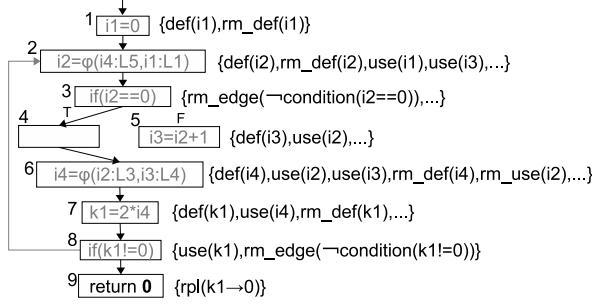
以降の節において，モデルは命令文をノードの単位とするが，説明しやすさのため，基本ブロックで表示する場合がある．図 6 は図 2(a) のプログラムから以下の手順によって生成されたモデルである．

- (1) π と π' に対して，制御フローモデル $M_\pi = (S, R, L)$ と $M_{\pi'} = (S', R', L')$ を生成する．

Instruction	i	::=	$t \leftarrow E \mid t \leftarrow [E] \mid [E_1] \leftarrow E_2 \mid t \leftarrow call(\bar{E}) \mid return(e) \mid read \mid write \mid$ $label(L) \mid jump(L) \mid E ? jump(L_1) : jump(L_2)$
Expressions	E	::=	$t \mid \&g \mid E_1 op E_2 \mid M$
Operators	op	::=	$+ \mid - \mid * \mid / \mid = \mid \neq \mid \leq \mid \geq \mid \phi \mid \& \mid >> \mid \dots$
Memory	M	::=	$*m \mid \&add \mid * \&add$

表 1 中間言語 LIR の構文

Table 1 The abstract syntax of the intermediate language LIR

図 6 モデル \tilde{M}
Fig. 6 Model \tilde{M}

- (2) $M_\pi = (S, R, L)$ と $M'_{\pi'} = (S', R', L')$ を比較し、最適化による変形を抽出する。抽出の手順は 4.2 節で述べる。
- (3) \tilde{M} を生成する。このモデルは $M_\pi = (S, R, L)$ に基づいて $M'_{\pi'} = (S', R', L')$ で変化した内容を反映したものである。変形の内容を原始命題で表示する。
- (4) \tilde{M} 上で合同集合を計算する。

まず、(1) であるが、図 2(a) のプログラムの CFG に基づいてモデルを生成する。そのモデルは CFG のノードを逐次に命令文単位に分割したものである。例を図 6 に示す。各ノードに命令文の定義-使用などの性質を原始命題として表現する。たとえば、命令文 5: $i3 = i2 + 1$ に原始命題 $def(i3), use(i2)$ を追加する。それから最適化の変化をモデル上で反映する。ノード 1, 2, 3, 4, 5, 6, 7, 8 の命令文は最適化によって削除された。削除された命令文のノードはそのまま残すが、 $rm_def(i1)$ のように削除されたことを原始命題として表現する。条件文 3: $if(i2 == 0)$ は $false$ になることがないため、分岐 $3 \rightarrow 5$ は削除される。削除された辺はモデルから取り除き、その辺に分岐する条件が $false$ であることを原始命題 $\neg condition(i2 == 0)$ として表現する。ノード 9 の命令文は変数 $k1$ を定数 0 に書き換えた、それを原始命題 $rpl(k1 \rightarrow 0)$ によって表現する。命令文の追加などほかの変形はこの例に現れないが、そういう場合も同じようにモデルにこの変

化を反映する。以下はモデルを生成する詳細を述べる。

4.1.1 制御フローモデル M_π と $M'_{\pi'}$

コード π に対する制御フローモデルは三つ組 $M_\pi = (Node_\pi, \rightarrow_\pi, L_\pi)$ として定義される。ここで $Node_\pi$ は各文ごとに分けられた命令文の集合であり、関係 \rightarrow_π 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。

$$\begin{aligned}
 n_1 \rightarrow_\pi n_2 \text{ iff} \\
 I_{n_1} = t \leftarrow E, t \leftarrow [E], [E_1] \leftarrow E_2, t \leftarrow call(\bar{E}), Label, skip \} \\
 \wedge n_2 = n_1 + 1 \\
 \vee (I_{n_1} = jump L \wedge L \wedge Label \wedge n_2 = L) \\
 \vee (I_{n_1} = if E? jump L_1 else L_2) \\
 \wedge (n_2 = L_1 \vee n_2 = L_2) \\
 \vee (I_{n_1} = read X \wedge n_2 = n_1) \\
 \vee (I_{n_1} = write X \wedge n_2 = n_1)
 \end{aligned}$$

表 2 遷移関係 \rightarrow_π の定義Table 2 Definition of transition \rightarrow_π

この最後の 2 つの式からわかるように、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。

表 3 は原始命題を定義し、空白の行によって上下の 2 部分に分けられる。上の半分は最適化前の性質である。後半の部分は最適化による変更の性質を表す。

4.1.2 最適化による変形を M_π に反映した \tilde{M}

説明しやすいように、たとえば $node \in rm_def$ はノードが最適化によって代入文が削除されたことを表す。ノードは命令文を単位とするため、命令文を指す。遷移関係の辺は制御フローグラフの有向辺を指す。生成されたモデル \tilde{M} は Kripke 構造の三つ組 $(\Omega, \varpi, \Upsilon)$ とする。最適化によるプログラムの変形には 7 種類ある。それは代入文の削除 (追加)、jump 文の削除 (追加)、変数 (式) の書換え、条件文が満たされないことによる辺の削除と式 E の移動による例外へ飛ぶ辺の変更である。それらの変形がモデル \tilde{M} を生成する前に抽出されるが、説明の都合で詳細は 4.2 節に述べる。以下に各変形をモデル $\tilde{M} = \{\Omega, \varpi, \Upsilon\}$ にどう反映するかを説明する。

原始命題 Υ :

モデル $M = (\Omega, \varpi, \Upsilon)$ における Υ の定義はプログラ

ムの性質及び最適化による変化に従って表3に示すようになる。これは空白の行の前後の二つの部分からなる。上部はプログラムの *def*, *use* など最適化前の性質であり、下部は最適化による変化の性質である。

$\Upsilon =$

- U { *use*(X): 変数 X は n で使用される }
- U { *def*(X): 変数 X は n で定義される }
- U { *trans*(E): 式 E は n で変更されない, E 中の変数は n で定義されない }

- U { *rm_def*(X): 変数 X は n で定義されるが, 最適化によって削除される }
- U { *rm_use*(X): 変数 X は n で使用されるが, 最適化によって削除される }
- U { *rpl_var*(X1 \rightarrow X2): n で変数 X1 が最適化によって変数 X2 に書換えられる }
- U { *rpl_exp*(E1 \rightarrow E2): n で式 E1 が最適化によって式 E2 に書換えられる }
- U { *rm_branch*(*condition*(e))(e): n で辺 e が *condition*(e) を満たされないため削除される }
- U { *ins_def*(X): n は X に対する定義文が挿入されたノード }
- U { *ins_jump*(L1 : *jump*(L2)): n は挿入された L1 から L2 に飛ぶ *jump* 文である. }
- U { *equal*(E1, E2): 式 (変数)E1, E2 は n で同じ合同集合に入る }
- U { *condition*(E): 条件式 E は n で成り立つ. }

表3 原始命題 Υ の定義

Table 3 Definition of primitive proposition Υ

状態集合 Ω :

最適化によって変化がないノードに対しては、そのままにする。

最適化によって書換えられたノードに対しては、ノードをそのままに残し、変更された部分を原始命題 *rpl*($v1 \rightarrow v2$) によって追加する。

最適化によって削除されたノード (代入文または *jump* 文) に対しては、ノードをそのままに残すが、削除されたことを原始命題 *rm_def*(v) または *rm_jump*(L) によって表示する。

最適化によって追加されたノード (代入文または *jump* 文) に対しては、その命令文のノードを新たに追加する。追加されたことを原始命題 *ins_def*(v) または *ins_jump*(L) によって表示する。

従って、状態集合は最適化前のすべてのノード S 以外に、最適化後によって追加、削除されたノードも含むことになる。

$\Omega = S \cup \Sigma node'$ (node' は追加、削除されたノード) になる。

遷移関係 ϖ :

モデル \tilde{M} の遷移関係 ϖ を表4に示す。

- $n_1 \rightarrow n_2$ iff
- $\vee (n_1 \rightarrow n_2 \in R \wedge n_1 \rightarrow n_2 \in R') \dots\dots\dots (1)$
 - $\vee (n_1 \rightarrow n_2 \in R \wedge n_1 \in rm_node \vee n_2 \in rm_node) \dots\dots\dots (2)$
 - $\vee (n_1 \rightarrow n_2 \in R \wedge n_1 \rightarrow n_2 \notin R' \wedge n_1 \rightarrow n'_1 \rightarrow \dots \rightarrow n'_n \rightarrow n_2 \in R') \dots\dots\dots (3)$
 - $\vee (n_1 \rightarrow n_2 \notin rm_branch) \dots\dots\dots (4)$

表4 遷移関係 ϖ の定義

Table 4 Definition of transition ϖ

図7によって説明する。図7(M_π) は最適化前のプログラムのモデルであり、図7(M'_π) は最適化後のプログラムのモデルであり、図7(\tilde{M}) は M_π に最適化による変化を反映したモデルである。「*ins*」で表示しているノード7,8は最適化により挿入されたものとし、「*rm*」で表示しているノード5は削除されたものとする。最適化の後、削除された分岐 $2 \rightarrow 4$ を灰色で表示する。

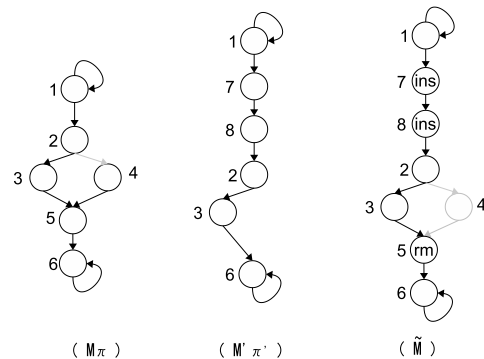


図7 遷移関係

Fig. 7 Transition relation of \tilde{M}

表4の式(1)は $n_1 \rightarrow n_2$ の遷移関係は図7の $2 \rightarrow 3$ のように、 M_π と M'_π で変わらないため、 \tilde{M} では維持することを意味する。式(2)は $n_1 \rightarrow n_2$ の遷移関係は M'_π で削除されたノード同士によるものである。削除されたノードは残るため、それらのノードへの遷移関係も維持する。例えば $3 \rightarrow 5$ または $5 \rightarrow 6$ はその例である。式(3)は挿入されたノード $n'_1 \dots n'_n$ が M_π の遷移関係 $n_1 \rightarrow n_2 \in R$ を分割し、新たにモデルに追加したことを意味する。例えば M_π の $1 \rightarrow 2$ を分割し、 \tilde{M} の $1 \rightarrow 7 \rightarrow 8 \rightarrow 2$ になったのはその例である。式(4)は M'_π で削除された分岐によって削除された遷移関係を取り除くことを意味する。例えば $2 \rightarrow 4$ または $4 \rightarrow 5$ はその例である。

4.1.3 検証における時間順について

3.3節で説明したように、時相論理を用いて検証する場合、検証に使われる条件は検証の結論より時間順

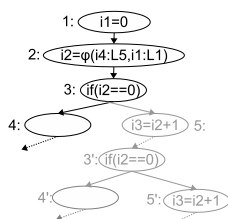


図 8 図 2(a) のプログラムから展開した CTL 木
Fig. 8 CTL tree of program in figure 2(a)

に先行していないといけない。

図 6 上で無限に遷移する CTL 木は図 8 である。最適化によって遷移しなくなった部分は灰色で表示する。3' はノード 3 を二度目に迎えることを意味する。

図 8 に示したように、CTL 木はプログラムの開始点から分岐しながら展開する。

モデル \tilde{M} は最適化によって、一時変数のノードの追加や、条件文が満たされないことによって分岐しない辺の削除など最適化前のプログラムを構造的に変更することがある。辺の削除については、初めて条件文のノードに到達するまでの CTL 木において、条件文が成り立つかどうかを検証する。例えば、図 2 の条件文 $if(i1 == 0)$ が満たされるかどうかを検査したいとき、図 8 において初めて現れたときのノード 3 を検査対象とし、時間順で先行して成立するのはノード 1 と 2 であるため、検査式にノード 1 と 2 に関する性質しか含んではいけない。従来研究 [21]) は検査式において、ノード 5 を含むため、検査対象がノード 3' になる。ノード 3' を検査するには、自分自身の命令文のノード 3 をすでに通ってからではないといけないため、循環論証である。従って、本研究はそれを避けるため、辺を削除できるかどうかの論証は、CTL 木構造上から検証対象となる辺を取り除いて、その辺を通ったことがないモデル上で行う。モデル \tilde{M} の初めて条件文のノードに到達するまでの CTL 木と最適化前のモデル M_π のその部分とは等価である。循環論証を避けるための辺の削除がモデル M_π の意味を変えるが、検証するには、意味的に等価である部分しか使わないため、 \tilde{M} を用いて検証するのが正しい。

4.2 最適化によるプログラムの変形の抽出及び変形に対する検査式の定式化

2.1 節に述べたように、ここで扱う SSA 形式は、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。SSA 形式上での最適化は最適化前の中間コードの変数の名前も制御フローグラフのラベル名も変えない。その特徴があるため、最適化前後の中間コードの命令文と辺の変形を精確に

抽出することができる。現実のプログラムを比較するのは非常にややこしいが、従来研究のように最適化器を生成したり、改造したりする必要がないため便利である。

前述のとおり、最適化による変形は命令文の変形と辺の変形を含めて全部で 7 種類ある。代入文の変形は削除と追加の 2 種類ある。命令文の一部 (変数または式) の書換えは 1 種類ある。辺の変形は、条件文の変形により分岐が削除されるものと、例外を投げる命令文の移動によって例外への遷移が変更されるものの 2 種類ある。危険辺除去と空の基本ブロック除去の 2 種類の最適化は辺の遷移関係を変えるが、辺の変形を検証する代わりに jump 文の変形によって検証できる。ただし、jump 文の削除と追加の条件が代入文と違うため、別途に 2 種類の定式化が必要となる。

ここで述べている命令文の変形は最適化の具体的な 1 つのフェーズに対応しているとは限らず、すべてのフェーズで起こりうるものであることに留意されたい。そのため、最適化のアルゴリズムや実装のアルゴリズムとは関係がない。従って、最適化ごとに多くの式を個別に記述する必要がなく、定式化が簡単である。新しい最適化器を追加したり最適化や実装のアルゴリズムが変わったりしても適用できる。検証者が最適化アルゴリズムを理解しなくても定式化できる点は大きな特長である。

さて、最適化の変形が満たすべき条件を 3 節で述べた時相論理式で記述しておく。これが CTL 検査式による定式化である。表 5 はそれら 7 つの検査式^{*1}である。以下は変形と変形が満たすべき CTL 式の定式化について表の順番に説明する。各式の詳細な意味は付録 A.1 を参照されたい。

代入文を除去する検査式 rm_def :

変数 v が定義した後使用されないか、最適化により使用が削除された、かつ、使用が最適化によって追加されていない場合である。たとえば図 9 の $x = \dots$ の x は使用されないため、削除できる。

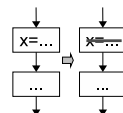


図 9 代入文の削除
Fig. 9 Statement deletion

*1 検査式の中の自由変数を具体的な変形の変数や式などに束縛する必要がある。

変形の名	変形の内容	変形に対する検査式
<i>rm_def</i>	代入文 $v = x$ を除去する	$\neg(E \text{ trans}(v) U ((\text{use}(v) \vee \neg \text{rm_use}(v)) \vee \text{ins_use}(v)))$
<i>rm_jump</i>	jump 文 $L1 : \text{jump}(L0, L2)$ を除去する	$\neg \overline{E} (\text{true} U (\text{use}(L1) \wedge \neg \text{rpl}(L1 \rightarrow L2))) \wedge \neg E (\text{true} U (\text{use}(L1) \wedge \neg \text{rpl}(L1 \rightarrow L0)))$
<i>rpl</i>	変数や式 $v1$ を $v2$ に書換える	$\overline{A} ((\text{trans}(v1) \wedge \text{trans}(v2)) U \text{equal}(v1, v2))$
<i>ins_def</i>	代入文 $v = e$ を追加する	$\neg((\overline{E} \text{ trans}(v) U \text{def}(v)) \wedge (E \text{ trans}(v) U \text{use}(v)))$
<i>ins_jump</i>	jump 文 $L1 : \text{jump}(L0, L2)$ を追加する	$\neg \overline{E} (\text{true} U (\text{use}(L2) \wedge \neg \text{rpl}(L2 \rightarrow L1))) \wedge \neg E (\text{true} U (\text{use}(L0) \wedge \neg \text{rpl}(L0 \rightarrow L1)))$
<i>rm_branch</i>	条件 $\text{condition}(e)$ が満たされないことよ って辺を削除	$\neg \overline{A} (\text{true} U \text{condition}(e))$
<i>rpl_edge</i>	式 e の移動によって例外へ飛ぶ辺の変更	false

表 5 各種の変形に対応した CTL 検査式
Table 5 CTL Check formulas of optimization transformations

jump 文を除去する検査式 *rm_jump* :

基本ブロック間の遷移は最後の jump 文による。基本ブロックが空になって除去される時、その基本ブロックの最後にある jump 文を削除することになる。図 10 を例として説明する。jump 文の変形は先行ノードと後続ノードが関わるため、 $L1 : \text{jump}(L0, L2)$ によって、 $L0 \rightarrow L1 \rightarrow L2$ の接続の関係を表す。ラベル $L1$ はノード $L0$ において飛び先として、ノード $L2$ において ϕ 関数の引数として「使用」される。jump 文 $L1 : \text{jump} L2$ を削除するとき、先行ノードと後続ノードにおいてラベル $L1$ の使用を削除し、直接繋ぐように $L0 : \text{jump} L2$, $L2 : \phi(x0:L0, x1:L3)$ に変換しないと行けない。

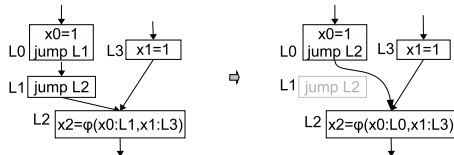


図 10 jump 文の削除
Fig. 10 jump statement deletion

変数や式を書換える条件式 *rpl* :

命令文 s で変数 (式) $v1$ が $v2$ に書換えられる条件は、 $v1$ と $v2$ が同じ合同集合¹⁷⁾ に属するようになった後、値が変わらないままで書換え箇所に通じることである。合同集合とはどんな実行経路でも同じ値になる変数 (式) の集合である。原始命題 $\text{equal}(v1, v2)$ が $v1$ と $v2$ が同じ合同集合に属することを意味する。説明の都合で合同の詳細は 4.3 節で述べる。図 11 では $x = 1$ の時点で合同集合 $\{x, 1\}$ が得られる。 $y = x$ が $y = 1$ によって書き換えられた時点までの経路で x が変わらないため、変換は正しい。

代入文追加の条件式 *ins_def* :

代入文 $v = e$ は一時変数が必要となるとき追加され

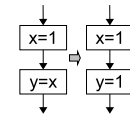


図 11 命令文の書き換え
Fig. 11 Statement replacement

る。代入文の $v = e$ の挿入は最適化前の v の定義使用連鎖¹⁷⁾ を壊してはいけない。つまり、挿入した代入文 $v = e$ から v が再定義がないまま、順向きで v の使用に、逆向きで v の定義文に辿りつく経路があってはならない。図 12 は代入文 $t = a + b$ を追加した例である。

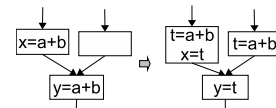


図 12 代入文の追加
Fig. 12 Definition statement insertion

jump 文追加の条件式 *ins_jump* :

図 13 の $L3 : \text{jump} L2$ の追加は危険辺を除去する意味である。 $L1$ と $L2$ の間に $L3$ を挟むように、 $L1$ での $L2$ への参照及び $L2$ での $L1$ への参照を $L3$ に変更し、 $L3$ の最後を $\text{jump} L2$ とする。

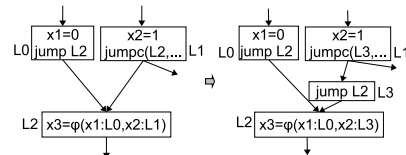


図 13 jump 文の追加
Fig. 13 Jump statement insertion

条件文による辺の削除の条件式 *rm_branch* :

図 14 の $L0$ から $L1$ への分岐は、どんな実行でも条件が満たされないため、その分岐を削除する。表 5 に示した式の意味は、逆向きですべての経路において、削除された辺に分岐する条件文が常に満たされない。

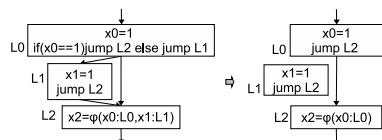


図 14 辺の削除
Fig. 14 Branch deletion

文の移動による例外分岐の移動 *rpl_edge* :

図 15 のように、例外を投げる命令文 x/y の移動が正しいかどうか y の値が 0 になるかどうかによるため、検査せずに「ALARM」として検証者に報告し、検証者が判断する。

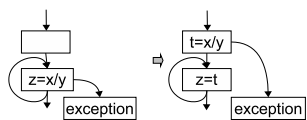


図 15 文の移動による例外分岐の移動:
Fig. 15 Movement of edge

これらの検査仕様は予め定義しておく。変形が抽出されたら、個々の具体的な変形に対して、以上の CTL 検査式に現れた自由変数を具体的な変形の変数や式に束縛する必要がある。例えば図 2 の例で $i0 = 0$ の削除を検査するには代入文を削除する検査式に束縛すると以下のようになり、その式を用いてモデル検査する。

$$\neg(E \text{ trans}(i1)U(\text{use}(i1) \wedge \neg \text{rm_use}(i1) \vee \text{ins_use}(i1)))$$

4.3 合同集合の計算

最適化前後の中間コードを比較するには、最適化前後の変数の等価性、すなわち合同 (congruence)²⁾¹⁷⁾¹⁹⁾ を求める必要がある。合同はどんな実行でも同じ値になる対象 (変数、定数また式) を指す。合同集合 (congruence set) は合同である対象の集合である。変数の等価性を見つけるアルゴリズムは Kildall ら¹⁰⁾、Alpern ら²⁾ 及び Rütthing ら¹⁹⁾ の研究など色々ある。Kildall らの手法は SSA 形式に関わらず、データフロー解析によって、すべて等価である変数を見つけるが、計算量が $O(2^n)$ であるため、使いにくい。Alpern らの手法は計算量が $O(n \log(n))$ で効率がよいが、違う演算による計算を違う変数とみなすため、精度が低い。Rütthing らの手法は折衷的な方法であり、計算量が $O(n^4 \log(n))$ である。

我々は独自のやり方を用いている。この手法は Alpern らや Rütthing らの手法と同じように SSA 形式に限定するが、複雑なデータフロー解析を必要としないで、計算量が $O(n)$ である。SSA Graph を導入して変形することによって、合同集合の範囲を拡張し、条件付き定数伝播²⁸⁾ や質問伝播に基づく大域値番号付けと部分冗長性除去²⁶⁾ といった、非常に複雑な最適化を行った場合の変数の等価性も求められる。

本手法は最小合同集合からはじめ、「悲観的」に合同集合を求める。計算は以下の手順による。説明は図 16 の $a2, b2$ を例とする。図 17 は図 16 のプログラムから展開した CTL 木構造である。モデルのノードは本当は命令文単位であるが、表示しやすさのため、基本ブロック単位で表示している。

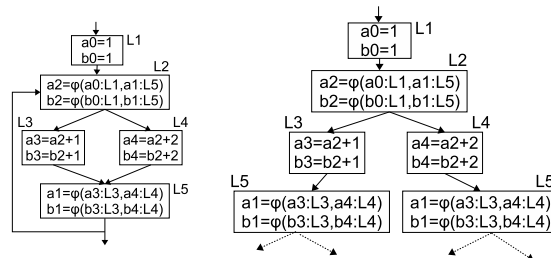


図 16 変数の等価性の例
Fig. 16 Congruence Set

図 17 図 16 の CTL 展開木
Fig. 17 CTL extended by figure 16

各代入文に新しい合同集合を生成し、独自の名前をつける。定数メンバーがある場合、合同集合の値をその定数とする。

- 定数や変数による代入文に対して、左辺と右辺をメンバーとする合同集合を生成し、合同集合に名前を付ける。定数による代入の場合、合同集合に値を与える。ここでは命令文 $a0 = 1$ の合同集合

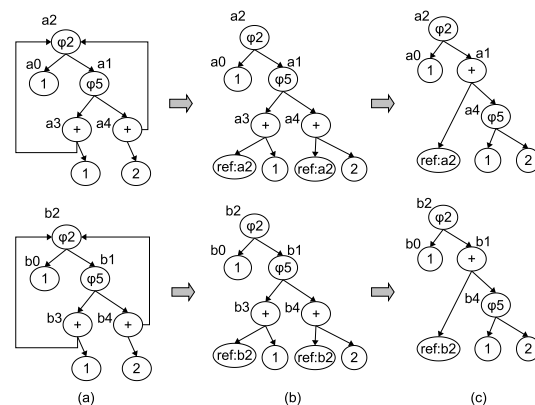


図 18 SSA グラフ及び変形
Fig. 18 SSA graph and transformation

$\{a0, 1\}$, $b0 = 1$ の合同集合 $\{b0, 1\}$ が求められる。

- 右辺が二項式, 例えば $a2, b2$ の場合, 以下の手順で処理する。

- 図 18(a) のように SSA グラフ²⁾¹⁷⁾ を生成する。 ϕ 関数にその ϕ 関数が属した基本ブロック番号を付ける。定数畳み込みできるものは畳み込みを行う。木のノードは *name*, *data*, *value* から構成される。*name* はノードの名前である。図 18 では各ノードの側に表示される。*data* はノードが格納した内容である。非終端ノードの場合は演算子で, 終端ノードの場合はノードの名前と同じとする。図 18 ではノードの中に書かれている。*value* はノードが定数の値がある場合与えられる値である。
- 図 18(b) のようにループ構造を木構造に表現する。ループで戻り辺に指されているノード $a2$ のコピーを生成し, 元のノードを指すように *data* に *ref* : $a2$ のように格納する。この木構造は頂点ノードにある変数 $a2$ がプログラムにおける計算の振る舞いを表現するため, 計算木と呼ぶ。
- 計算木を整形する。計算木をある基準で整形すると, 見た目が違うが等価計算である計算の計算木が同じようになる。 ϕ 関数を下げるアルゴリズムには Rütthing ら¹⁹⁾ の手法を採用した。 ϕ ノードの各子ノードが同じ子を持つ場合, 結合律を適用する。その後, アルファベット順にノードをソートする。図 18 の (b) の計算木を処理すると, 図 18 の (c) になる。
- 計算木を文字列に戻す。整形された計算木のノードを前順で巡回し, ノードの *data* を文字列として並べる。図 17(c) において $a2, b2$ の計算木にこの処理をすると, それぞれ

$$\phi_2(1, +(ref : a2, \phi_5(1, 2)))_{a2}$$

$$\phi_2(1, +(ref : b2, \phi_5(1, 2)))_{b2}$$

になる。 $a2$ と *ref* : $a2$ 及び $b2$ と *ref* : $b2$ は SSA グラフの中の参照元と参照先である。それらを式の中に現れた順番に $\alpha_1, \alpha_2, \dots$ に書換えると $a2$ と $b2$ の計算木はともに $\phi_2(1, +(\alpha_1, \phi_5(1, 2)))_{\alpha_1}$ になる。これにより, $a2$ と $b2$ は合同であると判断できる。 α はループを表すノードである。小さい α はその直前の括弧で括った式を代表する。大きい α は小さい α が代表している式をもう一回展開することを意味する。

- 合同集合を時間順にマージする。この処理は時相

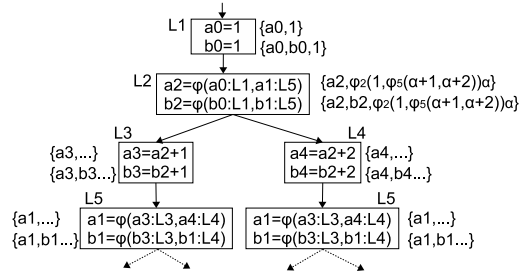


図 19 図 17 の CTL 展開木の各時点の合同集合
Fig. 19 Example of congruence set of figure 17

論理の観点から言うと, 各時点で成り立つ合同集合を計算する。CTL 木の開始ノードから, すべてのノードについて以下の処理を行う。時間順で先のノード A の合同集合が後のノード B の命令文の右辺を含む場合, A のメンバーを B の合同集合に加える。直観的に, ノード A で成り立つ等価性が途中で壊れていない限り, 時間順に B に流れてきても等価性が成り立つ。B の時点で, B の右辺がその合同集合に入っていれば, B の左辺もこの合同集合に合流できる。例えば, 図 19 の $a0 = 1$ の合同 $\{a0, 1\}$ に対して, $b0 = 1$ の右辺 1 が $\{a0, 1\}$ に入っているため, $\{a0, 1\}$ を $\{b0, 1\}$ に加え $\{a0, b0, 1\}$ とする。図 19 は合同集合を求めた例である。すべての変数について, 一回計算したら終了する。

図 2(b) のプログラムにおいて, $z1$ と $p3$ は Alpern ら²⁾ の手法では演算が違うため合同でないと判断されるが, 本研究では, 以上の手順で示したように処理すると, $p3$ と $z1$ が合同であることがわかる。

4.4 モデル検査及びバグの報告

ここまでで用意できたモデル及び検査式をモデル検査器に入力し, モデル検査する。すべての変形をその変形に応じた検査式を用いて検査する。検査式が満たされない場合, バグとして関数名, 行番号, 命令文, 変形の内容及び検査式が報告される。

バグは 3 種類に分類される。一つ目は意味的なエラーであり「Fault」として報告される。2つ目はある条件が成り立てば意味的なエラーになるもので「Possible semantic error」として報告される。3つ目は意味的なエラーを起こさないが, 最適化の意図に反して冗長性が混入した場合である。それは「Redundancy alert」として報告される。

5. 実 験

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。COINS コンパイラ⁵⁾のバックエンド上では、低水準中間表現 LIR⁶⁾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²²⁾が充実している。

データは COINS 1.4.4.2(最新版)において、SPEC2000 benchmarks に対する実験を行った結果から取得した。検査器のコードは約 6500 行である(モデル検査器を含まない)。実験環境は CPU: 1.98GHz SPARC64 V, OS: SunOS 5.10, JavaVM: 1.5.0 15, Heap Size: 512Mbyte, Memory: 10GByte である。

5.1 最適化器への適用

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。

- 無用命令除去 (DCE)
- 条件分岐を考慮した定数畳み込みと定数伝播 (CSTP)
- コピー伝播 (CPYP)
- 共通部分式除去 (CSE)
- ループ不変式移動 (HLI)
- 条件分岐を考慮した定数伝播 (CSTP)
- 空の基本ブロック除去 (EBE)
- 危険辺の除去 (ESPLT)
- 質問伝播に基づく大域値番号付けと部分冗長性除去 (PREQP)

COINS SSA 最適化モジュールには、条件分岐を考慮した定数伝播²⁸⁾や質問伝播に基づく大域値番号付けと部分冗長性除去²⁶⁾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、バグがある可能性がある。これらの最適化器を本手法により検証できた。

本手法はアルゴリズムに依存しないため、最適化の各フェーズ毎でも、全フェーズを適用した後も検証は可能である。

条件分岐を考慮した定数伝播 (CSTP) について：

条件分岐を考慮した定数伝播²⁸⁾は SSA 形式上での強力な定数伝播アルゴリズムである。その正しさの検証は非常に有用である。2.2 節の図 2(a) のプログラムは、条件分岐を考慮した定数伝播を行った後、図 3(a) になる。抽象実行 (abstract interpretation)⁸⁾によって、図 2(a) のグラフの入り口から順に辿ると、 $k1$ は常に 0 となり、ブロック $L4$ は到達不能であることなどが分かる。最適化後は $i0$ や $i1$, $k1$ 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、 $L4$ 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。

条件分岐を考慮した定数伝播はデータフロー方程式では扱えない。データフロー方程式は、 μ 計算²⁷⁾と同等の計算力があるといわれている。Lacey らの手法は μ 計算より計算力の低い CTL によるためこの問題に対応できない。佐原ら²¹⁾は条件付き定数伝播を証明したと主張しているが、3.3 節と図 8 に示したように、 $if(i2 == 0)$ と $i3 = i2 + 1$ が互いに検証の条件になっているため、循環論証だと思われる。検証できているように見えるが、実際はできていなく、実装を動かしてみると無限循環に陥る。

質問伝播に基づく大域値番号付けと部分冗長性除去 (PREQP) について：

2.1 節に述べたように、SSA 形式は変数の使用に到達する定義が一意に決定でき、プログラムの最適化に有利な形式である。しかし、2.2 節の図 1(b) のプログラムを SSA 形式に変換すると図 2(b) になる。 $z1 = x3 + y1$ は $p1 = x2 + y1$ と $p2 = x1 + y1$ によって計算された結果を冗長に再計算しているが、SSA 変換により、冗長だという情報が失われた。この冗長性を除去するアルゴリズムとして、質問伝播に基づく大域値番号付けと部分冗長性除去 (PREQP)²²⁾がある。図 2(b) のプログラムに PREQP を適用すると、図 3(b) になる。PREQP のアルゴリズムは複雑で正しいという確証を得るのは難しい。それに対して本研究は 4 節で述べた方法により、 $z1$ と $p3$ が合同であり、 $z1$ は $p3$ を代入すれば正しいことが簡単に認識できる。

5.2 未知の誤りや曖昧な変形の発見

以下は今の段階で発見したいくつかの不具合の例である。図 20 の (1) - (5) が本研究によって新しく発見されたバグであり、(6) が従来研究によって発見された既知のバグである。4.4 節に述べたように、バグは 3 種類に分けられる。

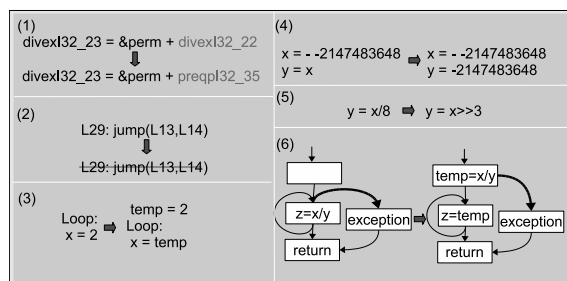


図 20 SPEC2000 のバグ
Fig. 20 Bug about SPEC2000

5.2.1 意味的なエラーを引き起こすバグ

この種のバグは場合によって致命的なエラーになる。こういうバグが見つかったら「Fault」として報告し、修正しないとイケない。この種の変換が PREQP と EBE のフェーズに 2 つ見つかった。

その 1(PREQP のバグ): 図 20(1) のバグは CSE と PREQP を 181.mcf にかけてと再現できる。実行可能なコードが生成されるが、正しい結果が得られない。

```
FAULT : in sort_basket.c
transformation : var_rpl(divexI32_22 → preqpI32_35)
at node 63 : divexI32_23 = &perm + divexI32_22
CTL formula :
 $\overline{A}(\text{trans}(\text{divexI32\_22}) \wedge \text{trans}(\text{preqpI32\_35})) \vee$ 
 $\text{equal}(\text{divexI32\_22}, \text{preqpI32\_35})$ 
```

この報告から、変数 *divexI32_22* を *preqpI32_35* に書き換えるのが不正であると指摘している。その情報に従って分析した結果、*preqpI32_44* が *preqpI32_35* の計算に必要な変数であるが、その定義がどこにもないことがわかった。

ほかに 188.ammp と 255.vortex に PREQP をかけると同様の報告がされるが、定義のない変数が使われないため、PREQP だけではベンチマークが通るが、さらに OSR⁷⁾ をかけると、CFG ノードを SSA グラフのノードに変換する手続きの中で、定義のない変数を変換しようとして、*null* 参照になるため、例外が投げられる。

その 2(EBE のバグ): 図 20(2) のバグは 253.perlbnmk に PREQP と CSTP をかけた後さらに EBE をかけると例外が投げられるバグである。

```
FAULT : in md5.c
transformation : rm_jump L29 : jump(L13, L14)
at node 191 : L29 : jump L14
CTL formula :
 $\overline{E} \text{ true } \vee (\text{use}(L29) \wedge \neg \text{rpl}(L29 \rightarrow L14)) \wedge$ 
 $\neg \text{E true } \vee (\text{use}(L29) \wedge \neg \text{rpl}(L29 \rightarrow L13))$ 
```

この報告から、変数 *L29* の削除がバグの原因であることがわかる、その情報に従って分析した結果、*L29* が $D.9.20 = \phi(D.9.19 : L29)$ に参照されているため、 $(\text{use}(L29) \wedge \neg \text{rpl}(L29 \rightarrow L13))$ を満たさない。削除すると *null* 参照になるため例外が投げられたことがわかる。

5.2.2 意味的なエラーを起こすかもしれないバグ

この種のバグは場合によって致命的なエラーになる可能性がある。こういうバグを「Possible semantics error」として報告し、バグであるかどうかの判断を検証者に任せる。この種の変換が 3 つ見つかった。

その 4(各フェーズ共通のバグ): 図 20(4) に示した変換について、 $- - 2147483648$ が符号なしで -2147483648 が符号付の場合、同じビットパターンであるが、符号なしの変数 *x* を符号付の変数 *y* に代入するのは実装依存のため、誤りの可能性がある。

その 5(各フェーズ共通のバグ): 図 20(5) に示した変換について、*x/8* が最適化によって、 $x \gg 3$ になった。*x* が負数のとき C 言語では、 \gg は実装依存であり、算術シフトの場合は大丈夫だが、論理シフトの場合は違った結果になる。

その 6(HLI のバグ): 図 20(6) に示した変換について、*x/y* が最適化によって、ループの外に追い出される。*y* が 0 になると例外を投げる箇所が変わる。このバグは佐原らの研究で見つかった既知な 254.gap のバグである。*y* が 0 にならなければこの変形はバグではない。

5.2.3 冗長性を導入するバグ

図 20(3) の変換は HLI のフェーズに現れる。この変換は意味的に間違っていないが、最適化の意図と逆に $x = 2$ を $x = \text{temp}$ に書き換え、 $\text{temp} = 2$ をループの外へ追い出すので冗長な代入を生じさせる。この種のバグを「Redundancy alert」として報告する。164.zip について、こういう冗長性を修正した所、実行時間が冗長性があるときより 27%も向上できた。

5.3 検査効率

ここで実験結果についての検査効率を表 6 に示す。A は検査器なしのコンパイル時間である、B は $-O2$ の各フェーズ毎に検査器をかけた場合のコンパイル時間である。C は $-O2$ の全フェーズを終わった後検査した場合のコンパイル時間である。D は検査した変換箇所数である、E は認識できない箇所数で誤りではないが、FAULT として報告した箇所数である。

最適化器としては COINS のオプション $-O2^{*1}$ から OSR⁷⁾ を取り除いたすべての最適化を含めた。フェーズ毎に全部の変形について検査器をかけた場合、平均してコンパイル時間が 18.66 倍になる。これは従来研究 18)21) より劣っているが、それは表 6 の D 欄に示したように検証箇所が多いからである。また、PREQP

*1 $-O2 : -\text{coins} : \text{hirOpt} = \text{cf}, \text{ssa} - \text{opt} = \text{prun}/\text{divex}/\text{cse}/\text{cstp}/\text{hli}/\text{osr}/\text{hli}/\text{cstp}/\text{cpyp}/\text{preqp}/\text{cstp}/\text{rpe}/\text{dce}/\text{srd3}, \text{loopinversion}$

	A	B	C	$\frac{B}{A}$	$\frac{C}{A}$	D	E	$\frac{E}{D}$
171.swim	15	173	46	11.53	3.07	2208	0	0
172.mgrid	30	359	80	11.97	2.67	3029	2	0.0007
179.art	22	418	94	19.00	4.27	1389	2	0.0014
188.ammp	330	10131	1078	30.70	3.27	3029	2	0.0007
175.vpr	269	16978	2390	63.12	8.88	12019	15	0.0021
181.mcf	62	334	116	5.39	1.87	1244	2	0.0016
197.parser	322	4669	1039	14.50	3.23	9416	26	0.0028
255.vortex	963	14660	6244	15.22	6.48	40724	36	0.0009
256.bzip2	152	1729	397	11.38	2.61	2997	9	0.0030
300.twolf	985	9335	2527	9.48	2.56	40254	3	0.0001
	sum(A)	sum(B)	sum(C)	$\frac{\text{sum(B)}}{\text{sum(A)}}$	$\frac{\text{sum(C)}}{\text{sum(A)}}$	sum(D)	sum(E)	$\frac{\text{sum(E)}}{\text{sum(D)}}$
sum	3150	58786	14011	18.66	4.45	116309	63	0.0005

表 6 検証なしと検証ありによるコンパイル時間 (単位: 秒) 及び検証数 (単位: 個)

Table 6 Comparison of times with and without verification (unit: second) and the number of transformations verified

や CSTP を含め、検証の精度が大幅に向上したことを考慮されたい。また、実装の改善によって検証時間を大幅に改善できると予測される。-O2 の全フェーズを終わった後検査した場合は、平均してコンパイル時間が 4.45 倍になるが、途中のバグが後のフェーズで消えてしまう可能性が十分高いため、推奨はしない。

6. 関連研究

最適化の正しさの検証に関する研究は多くなされている。1.2 節で、我々は従来研究を 3 種類に分類した。Necula らの研究¹⁸⁾ は本研究と同じカテゴリである。佐原らの研究²¹⁾ も本研究と同じカテゴリで、同じ研究グループによるもので COINS の SSA 最適化器を検証する。この節では、本手法に関連深いこの 2 つの従来研究について議論する。

Necula らは、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¹⁸⁾。プログラムを基本ブロックを単位として、変数に対して各基本ブロックで行った計算を記号評価する。各対応点 $\{PC_s, PC_t, E\}$ の結果が等価であるかどうかを検査する。 $\{PC_s, PC_t\}$ は最適化前後のプログラムにおいて、call 文、return 文及び jump(jumpc) 文の等価であるべき箇所 (notation) のペアである。E はその対応関係 (equivalence criterion) である。SSA 形式でないものも扱える。この研究は、処理時間がコンパイル時間の 8 倍しか遅くないため、実用的である。しかし、この研究では厳密なプログラムの意味の保存は保証されない。推測が失敗する場合があります、複雑な最適化については等価性の分析が難しくなり、精度が低いと思われる。本研究は Necula らの手法と同じカテゴリに属し、等価性によって最適化前後のプログラムを比較するが、以下の相違点がある。

検査箇所: Necula らの手法は最適化前後の call 文、return 文及び jump(jumpc) 文だけを対象するが、本

研究は call 文、return 文及び jump(jumpc) 文を含めてすべての変形を対象とするため、精度が高い。

合同の判断基準: Necula らの手法はプログラムの構造を解析するのを避けるため、Kildall ら¹⁰⁾ に近似した手法を採用した。それは命令文単位ではなく、基本ブロック単位で変数に対して記号評価を行う。そのため、基本ブロックを超える変形に対して、精度が低く、判断できない場合、検証器が Fault Alarm として報告される。本研究では独自の手法によって、 $O(n)$ の計算量で Kildall ら¹⁰⁾ よりやや劣った精度ではあるが基本ブロックを超える変形に対しても等価性が得られる。

検査の手法: 本研究は時相論理に基づいてモデル検査によって検証するため、複雑なデータフロー解析などを必要としなく、理論上の根拠が強いと思われる。

一方、佐原らの研究²¹⁾ は最適化器の振舞いを検査の対象とし、最適化しながら最適化のアルゴリズムと関わる箇所にマークをつけ、それらのマークの関係を証明して検証する。しかし、検証器が最適化のアルゴリズムを理解し、アルゴリズムの対応する箇所を分析し、アスペクト指向システム GluonJ³⁾ により最適化器の呼出しにアドバイスを書き入れる必要がある。普通の検証器はコンパイラ最適化器のアルゴリズムなどを理解するのは大変苦労すると思われる。また、この方法は最適化アルゴリズムに依存するため、すべてのアルゴリズムに対して多数の定式化が必要になる。質問伝播に基づく大域値番号付けと部分冗長性除去 (PREQP) や SSA 形式の Lazy code motion¹²⁾ のような関係が複雑な最適化については記述できない。佐原らは文献 20) にコピー伝播、共通部分式除去と無用命令除去の扱いについて記述しているが、3 種の最適化について、マークの種類と検査式の種類がそれぞれ 16 で合わせて 32 もある。最後の問題として、この研究はプログラムの制御フローグラフ CFG 上で変形

と関わる箇所の関係を検証する．そのため、CFG を CTL 木で展開するとそれらの関係は 3.3 節に述べたような循環論証に陥りやすい．それに対して、本研究はアルゴリズムに依存しないため、検証者が最適化器を理解する必要がない、最適化器を改造しなくてもよい、定式化が非常に簡単である、などの長所がある．また、変形した箇所間の関係を記述するのではなく、変形の意味論のみを時相論理上で記述するため、循環論証がなく、精度が高い．

7. 考察と今後の課題

この節では、本手法に関していくつかの考察と将来課題を述べる．

7.1 本研究の新規性

本研究は以下の新規性がある．

- 最適化アルゴリズムに立ち入らず最適化による変形を時相論理に基づいてモデル検査するだけでよい．この方式は本手法が初めてである．検証は変形だけを対象とし、余計な検証が不要で、精度が高い．
- SSA グラフを合同計算に導入した．SSA グラフを木構造によって表現した後、標準的な形に整え、記号評価を行うのはオリジナルな手法である．この手法により合同集合の範囲が拡大した．
- 明らかな冗長性を検出できる．

7.2 SSA 形式の制限について

本研究は厳密性があり、完全な意味を検証できるが、従来研究 [21]) と同じように SSA 形式に限られる．しかし、SSA 最適化は COINS や gcc^[11]、最近の多くのコンパイラで採用されている、精度の良い合同の計算が実用的な時間で計算できる、などから合理的な選択と考えられる．

7.3 型変換に関わる変形について

型と型の変換も合同集合の対象とすると、検証の速度が大幅に落ちると考えられるため、定数の型を全部 double に統一した． $x = 0$ と $x = 0.0$ を区別しないといけない時、バグであるのに、報告されないため、誤差が生じる可能性がある．現実には稀であるため、精度に殆ど影響しないと考えられる．実験したベンチマークのプログラムではそういうバグの誤差が 1 回も生じなかった．

逆に、型変換に関わった変形の検証について認識できないため、等価変形であるが、バグとして報告されるものがある．表 6 の D 欄に示したのは各ベンチマークにおいて最適化 $-O2$ (OSR を除く) をかけた場合、変形した箇所の数であり、E 欄はそのうちで、型変

換によって認識できなかった数である．そういう変形は *FAULT* の報告に *at node 133: divexF64_12 = (double)divexF32_16* のように報告されるため、手動で判定することが容易である．

7.4 今後の課題

今後の課題としては、主に次の 3 つが挙げられる．

- 本手法で記述した最適化の変形箇所の検査式は、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、厳密な証明を与えるべきである．
- OSR も SSA 形式上で行われる最適化のフェーズであるが、現時点では適用外である．SSA グラフの変形の等価性を解析することによって検証したい．
- 実装を改善することによって検証の効率を大幅に向上できると思われる．当面、等価性の評価は計算木を完全に比較することによるため、合同集合の計算が非常に時間とリソースを費やしている．部分的に比較しても等価性を判定できる場合があるため、それを用いれば実行速度が大幅に速くなると予測される．

8. ま と め

我々は、時相論理 CTL に基づいてモデル検査を利用し、最適化による変形がプログラムの意味を保存する等価変形であったかどうかを検査する手法を提案した．最適化による変形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 で記述しておき、最適化前後の中間言語プログラムを比較分析し、モデル検査を行うことで最適化による変形が性質を満たすことを調べる．本手法では、検証者は最適化や実装のアルゴリズムを知らなくてよいと、定式化しやすい、使いやすい、また、厳密性がある、テストプログラムを実行する必要がない、などの利点がある．また、効率も現実的である．

本手法により初めて複雑な最適化器 CSTP と PREQP を検証できた．また、冗長性を報告できるのも本研究の特長である．

この手法により、様々な既存の最適化器の検査が行えた．提案手法を実装し実験したところ、COINS コンパイラ最適化器の最新版において、SSA 最適化器の未知の不具合をいくつか発見することができた．

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) Alpern B., Wegman M. N., and Zadeck F. K.: Detecting equality of variables in programs, *In Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages*, pp.1–11, 1988 .
 - 3) Chiba, S., Nishizawa, M. and Kumahara, N.: GluonJ Home Page, <http://www.csg.is.titech.ac.jp/projects/gluonj/> .
 - 4) Clarke, E. M. Jr., Grumberg, O. and Reled, D. A.: *Model Checking*. The MIT Press, 1999 .
 - 5) COINS Project: COINS Home Page. <http://www.coins-project.org>
 - 6) COINS Project: COINS プロジェクト LIR 仕様書 <http://www.coins-project.org/spec/lir.pdf>
 - 7) Cooper, K.D., Simpson, L.T., and Vick, C. A.: Operator strength reduction. *ACM Trans. Program. Lang. Syst.*, Vol.23, No.5, pp.603–625, 2001 .
 - 8) Cousot, P. and Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 238–252, Los Angeles, California, 1977. ACM Press, New York.
 - 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, pp.451–490, 1991 .
 - 10) G.A.Kildall.: A unified approach to global program optimization, *1st ACM Symposium on Principles of Programming Language*, pp.194–206, October 1973.
 - 11) gcc Home Page. <http://www.gnu.org/software/gcc/gcc.html>.
 - 12) 今橋孝典, 伊藤陽, 佐々政孝: 静的単一代入形式上で通常形式部分冗長除去を実現する汎用的手法, 情報処理学会論文誌: プログラミング, Vol. 49, No. SIG 1 (PRO 35), pp. 84-95 (Jan. 2008).
 - 13) Jaramillo, C., Gupta, R. and Soffa, M. L.: Debugging and testing optimizers through comparison checking, *Electronic Notes in Theoretical Computer Sciences*, Vol. 65, No. 2, pp.1–17, 2002 .
 - 14) Lacey, D., Jones, N. D., Van Wyk, E. and Frederiksen, C. C.: Proving correctness of compiler optimizations by temporal logic. *Proceedings of Symposium on Principles of Programming Languages*, pp. 283–294, 2002.
 - 15) 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 .
 - 16) 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*, pp.220–231, 2003.
 - 17) 中田育男: コンパイラの構成と最適化. 朝倉書店, 1999.
 - 18) Necula, G. C.: Translation validation for an optimizing compiler, *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pp. 83–94, 2000.
 - 19) Rüthing, O., Knoop, J. and Steffen, B.: Detecting equalities of variables in program: combining efficiency with precision. *In Proc. 6th Int. Static Analysis Symposium (SAS'99), Lecture Notes in Computer Science 1694*, Vol.1694, pp. 232–247, 1999.
 - 20) 佐原聡一郎: 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, 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 .
 - 21) 佐原聡一郎, 佐々政孝: 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, コンピュータソフトウェア, Vol. 25, No. 1, pp. 151–166, Jan. 2008.
 - 22) 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. 2007. <http://www.is.titech.ac.jp/~sassa/coins-www-ssa/japanese/ssa-external-japanese.pdf>.
 - 23) 佐々 政孝: プログラミング言語処理系. 岩波書店, 1989 .
 - 24) Schmidt, D.A.: Data flow analysis is model checking of abstract interpretations. *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM New York, NY, USA, ISBN:0-89791-979-3, pp. 38–48, 1998 .
 - 25) SPEC Home Page. <http://www.spec.org/> .
 - 26) Takimoto M. and Harada K.: Efficient question propagation, in 22) .
 - 27) Van Leeuwen, J. (Ed.): Handbook of Theoretical Computer Science Vol. B, Formal Models and Semantics, Elsevier Science Publishers B. V., 1990. (広瀬ほか訳, 丸善)
 - 28) Wegman, M.N. and Zadeck, F.K.: Constant propagation with conditional branches, *ACM Trans. Program. Lang. Syst.*, Vol.13, No.2, pp.181–210, 1991 .

付 録

A.1 各種の変形に対応した CTL 検査式

図 21-26 は各種の変形に対応した CTL 検査式である．下の括弧と数字は各部分式を代表する．以下は各部分式の意味を説明する．

$$\text{rm_def} : \neg(E \text{ trans}(v) \cup (\underbrace{(\text{use}(v) \wedge \neg \text{rm_use}(v))}_{1} \vee \underbrace{\text{ins_use}(v)}_{2}))$$

図 21 代入文を削除するときの CTL 式

Fig. 21 Formula for removal of definition $v = e$

図 21 は代入文を除去するときの変形に対応した CTL 検査式である．

- 1: 変数 v が使用される, かつその使用が最適化によって削除されていない.
- 2: 1 または最適化によって挿入されていない.
- 3: 変数 v が定義された後, 再定義されないまま 2 に辿りつく経路がない.

$$\text{rm_jump}(L1:\text{jump}(L0,L2)) : \neg \overline{E} (\underbrace{\text{true} \cup (\text{use}(L1) \wedge \neg \text{rpl}(L1 \rightarrow L2))}_{3} \wedge \underbrace{\neg E (\text{true} \cup (\text{use}(L1) \wedge \neg \text{rpl}(L1 \rightarrow L0)))}_{2})$$

図 22 $L0 \rightarrow L1 \rightarrow L2$ の間から jump 文 $L1 : \text{jump } L2$ を削除するときの CTL 式

Fig. 22 Formula for removal of jump statement $L1 : \text{jump } L2$ from $L0 \rightarrow L1 \rightarrow L2$

図 22 は jump 文を除去するときの変形に対応した CTL 検査式である．

- 1: 基本ブロックのラベル $L1$ が使用されるが, 最適化により $L2$ に書換えられていない.
- 2: 基本ブロックのラベル $L1$ が使用されるが, 最適化により $L0$ に書換えられていない.
- 3: 逆向きで 1 に辿りつける経路がない.
- 4: 順向きで 2 に辿りつける経路がない.
- 5: 3 と 4 は同時に満たさないとはいけない.

$$\text{rpl}(v1 \rightarrow v2) : \overline{A} (\underbrace{(\text{trans}(v1) \wedge \text{trans}(v2))}_{2} \cup \underbrace{\text{equal}(v1, v2)}_{1})$$

図 23 変数 $v1$ を変数 $v2$ によって書換えた場合の CTL 式

Fig. 23 Formula for replacement of variable $v1 \rightarrow v2$

図 23 は変数 $v1$ を $v2$ によって書換えたときの変形に対応した CTL 検査式である．

- 1: 変数 $v1$ と変数 $v2$ が同じ合同集合に属する.
- 2: 変数 $v1$ と変数 $v2$ が経路上で再定義されない.
- 3: 2 の性質を保ったまま逆向きの経路において 1 に辿りつく.

$$\text{ins_def}(v=e) : \underbrace{\neg((\overleftarrow{E} \text{trans}(v) \cup \text{def}(v)) \wedge (\text{E} \text{trans}(v) \cup \text{use}(v)))}_{3}$$

図 24 式 $v = e$ を挿入するときの CTL 式
Fig. 24 Formula for insertion of statement $v = e$

図 24 は代入文を追加するときの変形に対応した CTL 検査式である。

- 1: 変数 v が再定義されないまま, 逆向きで v の定義に辿りつく。
- 2: 変数 v が再定義されないまま, 順向きで v の使用に辿りつく。
- 3: 1 かつ 2 の条件を満たさない。つまり, 最適化前の v の定義-使用連鎖が壊れない。

$$\text{ins_jump}(L1:\text{jump}(L0,L2)) : \underbrace{\neg \overleftarrow{E}(\text{true} \cup (\text{use}(L2) \wedge \neg \text{rpl}(L2 \rightarrow L1))) \wedge \neg \text{E}(\text{true} \cup (\text{use}(L0) \wedge \neg \text{rpl}(L0 \rightarrow L1)))}_{5}$$

図 25 $L0 \rightarrow L2$ の間に jump 文 $L1 : \text{jump } L2$ を挿入するときの CTL 式
Fig. 25 Formula for insertion of jump statement $L1 : \text{jump } L2$ to $L0 \rightarrow L2$

図 25 は jump 文を追加するときの変形に対応した CTL 検査式である。

- 1: 基本ブロックのラベル $L2$ が使用されるが, 最適化により $L1$ に書換えられていない。
- 2: 基本ブロックのラベル $L0$ が使用されるが, 最適化により $L1$ に書換えられていない。
- 3: 逆向きで 1 に辿りつける経路がない。
- 4: 順向きで 2 に辿りつける経路がない。
- 5: 3 と 4 は同時に満たさないといけない。

$$\text{rm_branch} : \neg \overleftarrow{A} (\text{true} \cup \underbrace{\text{condition}(e)}_{2})$$

図 26 条件文による分岐の削除に対応した CTL 式
Fig. 26 Formula for removal according to the deletion of branch

図 26 は条件文 $\text{condition}(e)$ の分岐を削除するときの変形に対応した CTL 検査式である。

- 1: 対象分岐へ飛ぶための条件文 e が満たされる。
- 2: 逆向きですべての経路において 1 に辿りつくことはない。

命令文の移動により例外への遷移が変わった場合, ALARM として報告する。条件式は *false*。