

# 時相論理を用いた コンパイラ最適化器の検証

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

佐原 聡一郎  
(05M37151)

平成 18 年度 修士論文

指導教官 佐々 政孝 教授

2007 年 2 月 28 日

## 概要

コンパイラは、ソースプログラムをその振舞いを変えずに正しくコンパイルしなくてはならない。ソースプログラムがいくら正しくても、コンパイラに誤りがあると目的コードの正しさは全く保証されない。

しかし、コンパイラは複雑なソフトウェアであり、正しく実装することは困難である。特に最適化フェーズは複雑なアルゴリズムによるものも多く、バグが混入しやすい。コンパイラの最適化フェーズの正しさを保証するのは重要な技術である。

最適化器の正しさを保証する既存の研究はいくつかある。Lacey らは、時相論理 CTL-FV を用いた記述から直接最適化を行う枠組みを提案し、その上で最適化の正しさを証明する手法を提案した。Lerner らは、時相論理による記述に似た独自の言語を用いて最適化するシステムを提案した。Lerner らのシステムは Lacey らの手法に似ているが、正しさの証明を定理証明器を用いてほぼ自動化したところに特徴がある。しかし、いずれの手法も、現実のコンパイラに使われている最適化を簡略化したものしか扱えないこと、および現実到手書きで書かれている最適化に適用できないといった制限がある。

本研究では、Lacey ら、Lerner らのような、最適化を記述から直接実行する手法とは異なった観点に立ち、実際に使われている最適化器の正しさを検証する手法を提案する。

提案する手法は、最適化によるプログラムの変形箇所がそれぞれ正しいものであったかどうかを、最適化後のプログラムを検証して確かめる立場をとる。変形箇所が満たすべき性質をあらかじめ時相論理 CTL-FV で記述し、プログラムの最適化後に論理式が成り立っていることをモデル検査により検証する。CTL-FV 式を満たせば変形が正しいことを Lacey らの方法などで別途証明しておけば、全ての変形箇所が式を満たすかどうかを検査することで、最適化が正しく行われたかどうかを判断することができる。

この検証は、時相論理によるモデル検査によって行う。それぞれの変形箇所が満たすべき性質を検査仕様として時相論理式で記述し、モデル検査を実施する。全ての変形箇所が式を満たせば、最適化は正しく行われたと判断する。

提案する手法は、既存の手書きの最適化器に適用でき、現実に使われている最適化の能力を損うことなく正しさを保証することができるといった利点がある。また、アスペクト指向の考え方を適用することで、既存の最適化器のソースの変更を最小限に抑えながら検査することが可能である。

本手法の適用性を確認するため、本手法を COINS コンパイラに実装されている SSA 最適化モジュールと、通常形式上での Lazy Code Motion に提案手法を適用し実行した。その結果、ループ不変式移動の未知のバグを発見することができた。このバグは、多くのテストプログラムをコンパイル・実行しても発見されていなかった潜在的なものであり、このようなバグを発見できるのは本手法の特筆すべき点のひとつである。

# 目次

第 I 部	序論	1
第 1 章	はじめに	2
1.1	背景 . . . . .	2
1.2	提案手法の概要 . . . . .	3
1.3	論文の構成 . . . . .	4
第 II 部	準備	5
第 2 章	プログラムの最適化	6
2.1	最適化と正しさ . . . . .	6
2.2	制御フローグラフ . . . . .	7
2.3	データフロー解析 . . . . .	10
2.3.1	利用可能な式の解析 . . . . .	10
2.3.2	生存変数解析 . . . . .	11
2.4	静的単一代入形式 . . . . .	11
2.4.1	SSA 形式上での最適化 . . . . .	12
第 3 章	時相論理とモデル検査	14
3.1	状態遷移システム . . . . .	14
3.2	時相論理 . . . . .	15
3.2.1	線形時間時相論理 — LTL . . . . .	16
3.2.2	計算木論理 — CTL . . . . .	18
3.2.3	CTL* . . . . .	20
3.3	モデル検査 . . . . .	21

3.3.1	モデル検査の計算量 . . . . .	22
3.3.2	$\mu$ 計算への変換 . . . . .	23
<b>第 III 部</b>	<b>提案手法</b>	<b>24</b>
<b>第 4 章</b>	<b>提案手法で採用する時相論理</b>	<b>25</b>
4.1	プログラムのモデル化 . . . . .	25
4.1.1	制御フローモデル . . . . .	26
4.1.2	原始命題の種類 . . . . .	26
4.2	CTL の採用 . . . . .	27
4.3	双方向経路を扱う CTL . . . . .	28
4.4	自由変数の導入 . . . . .	28
<b>第 5 章</b>	<b>提案手法</b>	<b>31</b>
5.1	概要 . . . . .	31
5.2	Lazy Code Motion . . . . .	32
5.3	ステップ 1 – 検査仕様の記述 . . . . .	33
5.3.1	仕様の構文 . . . . .	34
5.3.2	仕様の意味 . . . . .	34
5.3.3	Lazy Code Motion の検査仕様の記述 . . . . .	34
5.4	ステップ 2 – 既存の最適化器の拡張 . . . . .	37
5.4.1	アスペクト指向プログラミングの利用 . . . . .	38
5.5	ステップ 3 – モデルの生成 . . . . .	38
5.6	ステップ 4 – 変形箇所のマーク付け . . . . .	39
5.6.1	マーク付けの際の注意点 . . . . .	40
5.6.2	解析箇所のマーク付け . . . . .	40
5.7	ステップ 5 - CTL 式の生成 . . . . .	41
5.7.1	Lazy Code Motion での例 . . . . .	41
5.8	ステップ 6 – モデル検査 . . . . .	43
5.8.1	アルゴリズムの概要 . . . . .	44
5.8.2	各演算子ごとのアルゴリズム . . . . .	45

第 IV 部	実装	52
第 6 章	コンパイラインフラストラクチャ COINS と本手法との関係	53
6.1	概要	53
6.2	構成	53
6.3	COINS バックエンドの構造	54
6.4	LIR におけるモデルの定義	56
6.4.1	モデルの状態	58
6.4.2	状態間の遷移関係	58
6.4.3	原始命題の意味	58
第 7 章	提案手法の適用	61
7.1	ループ不変式移動への適用	61
7.1.1	ループ不変式移動の正しさ	62
7.1.2	効率向上のための工夫	64
7.2	条件分岐を考慮した定数伝播への適用	66
7.2.1	条件分岐を考慮した定数伝播の正しさ	67
7.3	他の最適化器への適用	70
7.4	アスペクト指向プログラミングによる最適化器の拡張	70
7.4.1	アスペクト指向プログラミングシステム GluonJ	71
7.4.2	ループ不変式移動を行う最適化器への適用例	71
7.4.3	ポイントカット指定が困難な場合	72
第 8 章	実験と考察	75
8.1	未知のバグの発見	75
8.2	最適化器の拡張箇所の個数	76
8.3	検査仕様の記述コスト	77
8.4	検査効率の実験	77
第 V 部	結論	80
第 9 章	関連研究	81
9.1	Lacey らの研究	81
9.1.1	本手法と Lacey らの手法の比較	81

9.2	Lerner らの研究 . . . . .	82
9.3	Necula の研究 . . . . .	83
9.4	Rinard らの研究 . . . . .	84
第 10 章	まとめ	85
第 11 章	今後の課題	86
参考文献		88
付録 A	COINS SSA 最適化モジュールの検査仕様の記述	91
A.1	コピー伝播 . . . . .	91
A.2	共通部分式除去 . . . . .	93
A.3	無用命令除去 . . . . .	96

# 目次

2.1	プログラムの制御フローグラフの例 . . . . .	8
2.2	SSA 形式への変換の例 . . . . .	12
3.1	LTL 式の例 . . . . .	18
3.2	CTL 式の例 . . . . .	20
5.1	提案手法の概要 . . . . .	31
5.2	部分冗長除去の例 . . . . .	33
5.3	最適化器の拡張の例 . . . . .	37
5.4	制御フローグラフの変更にモデルを同期させる例 . . . . .	39
5.5	誤ったマーク付けの例 . . . . .	40
5.6	複数箇所を最適化する例 . . . . .	43
5.7	CTL 式の構文木の例 . . . . .	44
6.1	COINS の構成図 . . . . .	54
6.2	Module の構成 . . . . .	55
6.3	FlowGraph の構成 . . . . .	56
6.4	instrList の構成 . . . . .	57
7.1	SSA 形式上でのループ不変式移動の例 . . . . .	62
7.2	ループ不変式移動の誤った最適化の例 . . . . .	63
7.3	条件分岐を考慮した定数伝播の例 . . . . .	66
7.4	ループ不変式移動を行う最適化器の擬似コード . . . . .	72
7.5	ループ不変式移動を行う最適化器の擬似コード . . . . .	73
7.6	空のメソッドの挿入の例 . . . . .	74

# 表目次

3.1	LTL の構文規則 . . . . .	16
3.2	LTL の意味定義 . . . . .	17
3.3	LTL の演繹体系 . . . . .	17
3.4	CTL の構文規則 . . . . .	18
3.5	CTL の意味定義 . . . . .	19
3.6	CTL の演繹体系 . . . . .	20
3.7	CTL*の構文規則 . . . . .	21
3.8	CTL*の意味定義 . . . . .	22
3.9	CTL*の演繹体系 . . . . .	22
4.1	$L(n)$ の直観的な意味の定義 . . . . .	26
4.2	CTL <sub>bd</sub> の構文規則 . . . . .	29
4.3	CTL <sub>bd</sub> の意味定義 . . . . .	29
5.1	仕様の構文規則 . . . . .	35
8.1	最適化器ごとの拡張箇所の個数 . . . . .	76
8.2	実験環境 . . . . .	78
8.3	検査の有無によるコンパイル時間の比較 . . . . .	79

# 第 I 部

## 序論

# 第 1 章

## はじめに

本論文では，既存の最適化器の正しさを時相論理を用いて検証する手法を提案する．

### 1.1 背景

コンパイラは，ソースプログラムを，その振舞いを変えずに正しくコンパイルしなくてはならない．ソースプログラムがいくら正しくても，コンパイラに誤りがあると，目的プログラムの正しさは全く保証されない．

しかし，コンパイラは複雑なソフトウェアであるため，正しく実装することは困難である．正しく実装するのが困難である理由のひとつに，次のようなバグの発見の難しさが挙げられる．

- コンパイルが正常に終了しても，生成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．これは通常，目的プログラムを実行するまで気付かない．
- 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，コンパイルのどのフェーズによるものか見分けるのが難しい．

特に，コンパイラの重要な技術のひとつである最適化は，複雑なアルゴリズムのものも多く，最適化の正しい実装は非常に困難であることが多い．最適化を正しく実装することは重要な技術である．

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．

- 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，その振舞いを変えずに最適化できる．

- 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 [18, 19] や Lerner らの研究 [20, 21] などがある．Lacey らは，時相論理 CTL-FV を用いた条件付き書き換え規則により最適化を行う枠組みを提案し，その枠組み上でいくつかの最適化の正しさを証明した．Lerner らは，時相論理による記述に似た独自の言語を用いて最適化するシステムを提案した．Lerner らの提案したシステムは Lacey らの枠組みに似ているが，Lerner らのシステムでは，正しさの証明を定理証明器を用いてほぼ自動化したところに特徴がある．どちらの手法でも，実装された最適化器は常に正しく動作することが保証されるが，現実のコンパイラに使われている最適化を簡略化したものしか扱えないこと，および現実には手書きで書かれている最適化に適用できないといった制限がある．

後者の研究には，Rinard らの研究 [26] や Nacula の研究 [24] 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．Nacula 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 提案手法の概要

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ずに正しく最適化できたかどうかを最適化実行後に検査する手法を提案する．

提案する手法では，最適化によるプログラムの変形箇所それぞれが正しいものであったかどうかを，最適化後のプログラムを検証して確かめる立場をとる．変形箇所が満たすべき性質をあらかじめ時相論理 CTL-FV [18] で記述し，プログラムの最適化後に論理式が成り立っていることをモデル検査により検証する．CTL-FV 式を満たせば変形が正しいことを別途証明しておけば<sup>\*1</sup>，全ての変形箇所が式を満たすか検査することで，最適化が正しく行われたかどうかを判断することができる．

提案手法による利点は以下の点である．

- 既存の手書きの最適化器に適用できる．
- 最適化後に検査を行うことで，Lacey ら，Lerner らより広い範囲の最適化を検査できる．

---

<sup>\*1</sup> この証明は例えば Lacey らの方法 [18] で可能である．

- アスペクト指向の考え方を適用し、既存の最適化器のソースの変更を最小限に抑えながら検査できる。
- 正しくない最適化に提案手法を用いることで、最適化器のバグを発見できる。バグの箇所の特定も容易である。
- 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、我々は COINS コンパイラ [8] に実装されている SSA 最適化モジュール [29] と、通常形式上での Lazy Code Motion [16, 17] に提案手法を適用し検査を実行した。その結果、ループ不変式移動の未知のバグを発見することができた。このバグは、SPEC CPU2000 [32] のベンチマークプログラムの一つである 256.gap をコンパイル・検査して発見できたものだが、このプログラムのベンチマーク実行では正常な実行が行われており、今まで存在が発見されていなかった潜在的なバグであった。このようなバグを発見できるのは本手法の特筆すべき点の一つである。

### 1.3 論文の構成

本論文の構成を以下に示す。

#### 第 II 部 準備

- 2 章 : プログラムの最適化に関する説明
- 3 章 : 時相論理とモデル検査に関する説明

#### 第 III 部 提案手法

- 4 章 : 提案手法で用いる時相論理の説明
- 5 章 : 提案手法の説明

#### 第 IV 部 実装

- 6 章 : COINS の説明
- 7 章 : 提案手法の COINS への適用例
- 8 章 : 実験と考察

#### 第 V 部 結論

- 9 章 : 関連研究
- 10 章 : まとめ
- 11 章 : 今後の課題

第 II 部

準備

## 第 2 章

# プログラムの最適化

本章では、プログラムの最適化について述べる。

### 2.1 最適化と正しさ

プログラムを最適化するとは、プログラムを「より良い」ものに変形することである。「より良い」とは、「より実行が速い」、「よりサイズが小さい」など目的によって異なるが、多くの場合、実行の速さに焦点を当てることが多い。最適化は普通、プログラムを解析し、その結果に基づき変形するといった手順を取る。

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、最適化前後で関数の戻り値が変わってしまったら、これは正しい最適化とは言えない。最適化前後でプログラムの振舞いが変わらないことを、プログラムの観測的意味が保存される、または単に意味が保存されるという。

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。だが、意味の保存の観点から見ると、必要条件ではないので、最適化の正しさについての議論の場合は考慮しなくてもよい。

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観測的意味が保存された場合のことを言う。

## 2.2 制御フローグラフ

制御フローグラフとは、プログラムの実行の流れを抽象化して表したグラフである [1, 3, 23, 28]。制御フローグラフは、基本ブロックをノードとし<sup>\*1</sup>、基本ブロック間の制御の流れの関係を有向辺で表した有向グラフである。

### 定義 2.1 (基本ブロック)

プログラムの最初の文、無条件あるいは条件分岐の行き先の文、無条件あるいは条件分岐の直後の文をリーダーという。リーダーから始まり、次のリーダーの一つ手前まで、あるいはプログラムの最後までの一連の文を基本ブロック、または単にブロックという。基本ブロック内の文は、先頭から終端までこの順で実行されることが保証される。

### 定義 2.2 (制御フローグラフ)

ブロック  $B_1$  の最後の文の実行の直後に、ブロック  $B_2$  の最初の文を実行する可能性があるとき、 $B_1$  と  $B_2$  の間には辺があるといい、 $B_1 \rightarrow B_2$  と表す。

$N$  をプログラムのブロックの全集合、 $E \subseteq N \times N$  を辺の全集合とすると、2 つ組  $G = (N, E)$  をプログラムの制御フローグラフ、または単にフローグラフという。 $N$  はフローグラフのノード、 $E$  はフローグラフの有向辺であるともいう。制御フローグラフは、基本ブロックをグラフのノードとせず、一つの文をグラフのノードとすることもある。

プログラムの最初の文を含むノードを、フローグラフの入口ノードといい、プログラムの最後の文を含むノードを、出口ノードという。

### 定義 2.3 (先行ノード、後続ノード)

フローグラフ  $G = (N, E)$  について、 $(n_1, n_2) \in E$ 、すなわち  $n_1 \rightarrow n_2$  であるとき、 $n_1$  は  $n_2$  の先行ノードであるという。また、 $n_2$  を  $n_1$  の後続ノードであるという。ノード  $n$  の先行ノードの集合を  $pred(n)$ 、後続ノードの集合を  $succ(n)$  と表す。

### 定義 2.4 (パス)

フローグラフのノードの列  $n_0, n_1, n_2, \dots, n_m (m \geq 0)$  について、 $\forall i, 0 \leq i < m; n_i \rightarrow n_{i+1}$  が成り立っているとき、この列を  $n_0$  から  $n_m$  へのパスという。この定義によると、長さが 1 のノードの列もパスとなる。

---

\*1 一つの文をノードとすることもある。

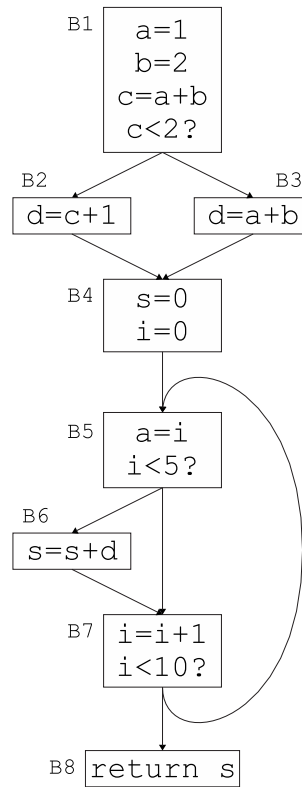
```

a = 1          (B1)
b = 2          (B1)
c = a + b     (B1)
if (c < 2) {   (B1)
    d = c + 1  (B2)
}
else {
    d = a + b  (B3)
}

s = 0          (B4)
i = 0          (B4)
do {
    a = i       (B5)
    if (i < 5) { (B5)
        s = s + d (B6)
    }
    i = i + 1   (B7)
} while (i < 10) (B7)

return s      (B8)

```



(a) プログラム

(b) 制御フローグラフ

図 2.1 プログラムの制御フローグラフの例

**定義 2.5 (先行パス, 後続パス)**

フローグラフの入口ノードから, ノード  $n$  に至るパスを,  $n$  の先行パスという. 逆に,  $n$  から出口ノードに至るパスを,  $n$  の後続パスという.

図 2.1 にプログラムの制御フローグラフの例を示す. B1~B8 は基本ブロックであり, このフローグラフのノードである\*2. 矢印が辺を表している. 入口ノードは B1 であり, 出口ノードは B8 である. このグラフの矢印を辿ったノードの列がパスである, 例えば, B5 からのパスは「B5, B7, B8」「B5, B7, B5, B6, B7, B8」などがある.

制御フローグラフは, コンパイル時に静的に決定しうる情報のみから構築される, プログラムの実行の「保守的な抽象化」であるともいえる. ここでの「保守的」とは, 任意のプログラムの実行経路は, そのプログラムの制御フローグラフのパスとして存在する, と

\*2 当然, 一つの文をノードとするフローグラフも考えられるわけだが, ここでは図は記載しない.

という意味である。

以下，制御フローグラフに関するいくつかの性質について述べる。

#### 定義 2.6 (支配)

フローグラフの入口ノードからノード  $n_2$  に至る全てのパスが必ずノード  $n_1$  を通るとき，ノード  $n_1$  はノード  $n_2$  を支配するといい， $n_1 \text{ dom } n_2$  と表す．この定義によると， $n \text{ dom } n$ ，つまり全てのノードは自分自身を支配することになる。

特に， $n_1 \text{ dom } n_2$  かつ  $n_1 \neq n_2$  のとき， $n_1$  は  $n_2$  を厳密に支配するという。

#### 定義 2.7 (帰辺)

フローグラフの辺  $n_2 \rightarrow n_1$  は， $n_1 \text{ dom } n_2$  が成り立つとき帰辺であるという．図 2.1(b) の例では， $B7 \rightarrow B5$  が帰辺となる。

#### 定義 2.8 (自然ループ)

フローグラフの辺  $b \rightarrow h$  が帰辺であるとする．この帰辺  $b \rightarrow h$  に関する自然ループとは， $h$  と， $h$  を通らずに  $b$  に到達できるような全てのノード ( $b$  を含む) を合わせたものである．つまり， $\text{loop}(h, b) = \{h\} \cup \{n \mid n \text{ から } h \text{ を通らずに } b \text{ に到達するパスがある}\}$  と表せる。

自然ループ  $\text{loop}(h, b)$  において，ノード  $h$  をこのループのヘッダという．また，ノード  $n \in \text{loop}(h, b)$  が  $n' \notin \text{loop}(h, b)$  なる後続ノードを持つとき，ノード  $n$  を出口ノードという。

#### 定義 2.9 (後支配)

フローグラフのノード  $n_1$  から出口ノードに至る全てのパスが必ずノード  $n_2$  を通るとき，ノード  $n_2$  はノード  $n_1$  を後支配するといい， $n_2 \text{ pdom } n_1$  と表す．この定義によると， $n \text{ pdom } n$ ，つまり全てのノードは自分自身を後支配することになる。

特に， $n_2 \text{ pdom } n_1$  かつ  $n_2 \neq n_1$  のとき， $n_2$  は  $n_1$  を厳密に後支配するという。

#### 定義 2.10 (制御依存)

ノード  $n_1$  からノード  $n_2$  への空でないパスがあり， $n_2$  は  $n_1$  を厳密に後支配しないが， $n_2$  はそのパスの  $n_1$  より後のすべてのノードを後支配するとき， $n_2$  は  $n_1$  に制御依存するという。

$n_2$  が  $n_1$  に制御依存するとき， $n_1$  の分岐の決定によって  $n_2$  が実行されたりされなかったりする。

## 2.3 データフロー解析

データフロー解析とは、制御フローグラフ上でデータの流れ（変数への代入や式の計算と使用の関係など）の解析を行うもので、データフロー方程式というもので定式化できる [1, 3, 23, 28]。以下、データフロー解析の典型的な例を紹介する。なお、簡単のため、フローグラフのノードは一つの文とする。

### 2.3.1 利用可能な式の解析

変数  $x$  を定義する文とは、 $x$  へ値を設定する可能性のある文のことである。変数を定義する文の典型的な例は代入文である。

式  $e$ <sup>\*3</sup> が、ある文に来るまでの間に必ず評価され、その評価からその文までの間に  $e$  の値が変更されていない<sup>\*4</sup> とき、この式はこの文で利用可能であるという。つまり、どのようなパスを通過してこの文に来たときも必ず  $e$  が計算されていて、かつ  $e$  の値が変わっていないということである。

利用可能な式が分かると、同じ式の再計算を防ぐ最適化が行える可能性がある。図 2.1(b) のブロック B3 では、すでにブロック B1 で計算した  $a + b$  を再び計算しているが、これは冗長な計算であり、 $a + b$  を  $c$  に置き換えて再計算を防ぐことができる。このような最適化を共通部分式除去という。

文  $n$  で利用可能な式の集合  $AVAIL(n)$  は次のデータフロー方程式を解くことにより求めることができる。ただし、 $kill(n)$  は文  $n$  で定義される変数の集合、 $gen(n)$  は  $n$  で新たに利用可能となる式の集合とする。

$$AVAIL(n) = \bigcap_{p \in pred(n)} ((AVAIL(p) \wedge \neg kill(p)) \vee gen(p)) \quad (2.1)$$

データフロー方程式は一般に、初期解を全集合または空集合とし、集合の最大または最小不動点解に至るまで繰り返し計算する反復解法によって解かれる。式 (2.1) の場合、 $AVAIL$  を全集合とし、最大不動点解に至るまで繰り返し計算する。

式 (2.1) を図 2.1(b) に対して解くと、例えば  $a + b \in AVAIL(\text{B3 の文})$  であることが分かる。この結果を利用すると、上で述べたような共通部分式除去を行うことができる。

---

\*3 例えば  $a + b$  などは式である。

\*4 式  $a + b$  の場合、 $a$  も  $b$  も定義されていないということ。

### 2.3.2 生存変数解析

ある文で定義された変数  $x$  の値が、その後どこかで使用される可能性があるとき、 $x$  は生きてるといい、そうでないとき、 $x$  は死んでいるという。ある文の直後で生きている変数を生存変数という。

生存変数でない変数を定義している代入文は不要な文であり、除去することができる<sup>\*5</sup>。このような最適化を無用命令除去という。

文  $n$  の直後での生存変数の集合  $LIVE(n)$  は次のデータフロー方程式を解くことにより求めることができる。ただし、 $use(n)$  は文  $n$  で使用されている変数の集合である。

$$LIVE(n) = \bigcup_{s \in succ(n)} (use(s) \vee (LIVE(s) \wedge \neg kill(s))) \quad (2.2)$$

式 (2.2) は、 $LIVE$  を空集合とし、最小不動点解に至るまで繰り返し計算することで解かれる。

式 (2.2) を図 2.1(b) に対して解くと、例えば  $a \notin LIVE(B5 \text{ の文})$  であることが分かる。つまり B5 で定義された  $a$  は死んでいることになる。この結果を利用すると、B5 での  $a$  への代入文を除去する無用命令除去を行うことができる。

## 2.4 静的単一代入形式

静的単一代入形式 (Static Single Assignment Form, 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, 11, 23, 3]。SSA 形式は、プログラムの最適化に有利な形式といわれている。

図 2.2 は、プログラムの SSA 形式への変換の例である。図 2.2(a) のプログラムは、図 2.1 と同じものである。これを SSA 形式に変換すると、図 2.2(b) のようになる。

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
- $\phi$  関数の挿入

を行う。 $\phi$  関数とは、元のプログラムで同じ変数の定義が合流するところに挿入される仮想の関数である。図 2.2(b) の B4 にある  $\phi(d0, d1)$  などが  $\phi$  関数である。この  $\phi$  関数は、

---

<sup>\*5</sup> 正確には、この代入文に副作用がない、という条件も必要である。

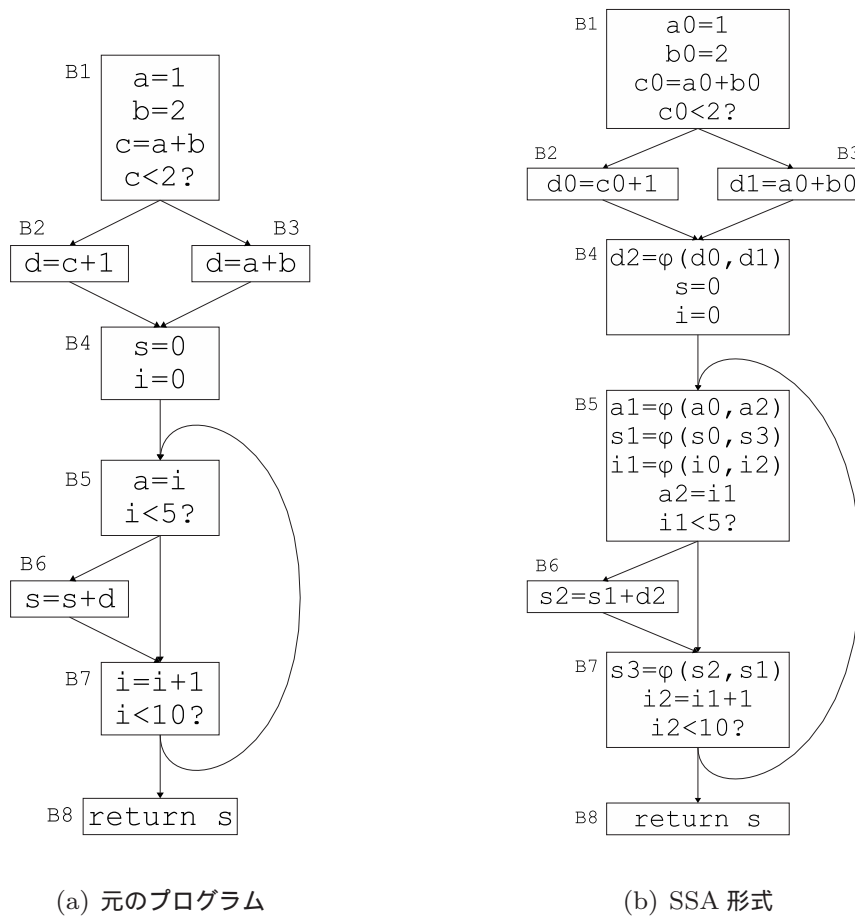


図 2.2 SSA 形式への変換の例

「B2 から来た場合は d0 を，B3 から来た場合には d1 を返す」関数である。

これを少し説明すると，図 2.2(a) の B4 において，d の値は，B2 から来た場合は B2 内で定義された d，B3 から来た場合は B3 内で定義された d の値となる．SSA 変換により，B2 で定義される d が d0，B3 で定義される d1 になるとすると，B4 での d の使用は d0，d1 のどちらになるのか決定できなくなる．これは，B4 の入口に  $\phi$  関数による代入文  $d2 = \phi(d0, d1)$  を挿入することで．B4 での d の値は d2 と一意に決定することができる．

### 2.4.1 SSA 形式上での最適化

SSA 形式のプログラムでは，変数の名前が同じならばそれらは全て同じ値となることが保証されている．この性質を利用して，通常形式上での最適化と同様の最適化を次のように簡単に行うことができる．

- 図 2.2(b) を見ると, B3 での  $a_0 + b_0$  は, B1 ですでに利用可能となっていることが分かる. よって, 共通部分式  $a_0 + b_0$  の除去を行うことができる.
- 図 2.2(b) を見ると, 変数  $a_1$  はプログラム上で使用されていないのが分かる. よって,  $a_1$  への代入文は不要な文として削除することができる. すると,  $a_2$  も使用されなくなり,  $a_2$  への代入文を削除できる.

このように, 通常形式上では式 (2.1) や式 (2.2) を解いて行っていた最適化が, SSA 形式では簡単に行える.

SSA 形式上で最適化などを行った後は, コンパイルを続けるために通常形式に変換する必要がある. 理由は,  $\phi$  関数を直接実行できるアーキテクチャがないためである. SSA 形式から通常形式への変換は, いくつかのアルゴリズムが提案されている [4, 33].

## 第 3 章

# 時相論理とモデル検査

プログラムの実行は、刻々と状態を変えていくシステムとしてモデル化できる。このようなシステムを状態遷移システムという。状態遷移システム上で「ある性質を持った状態にいつか到達する」といった性質を記述するには、命題論理では十分でなく、様相論理の一種である時相論理を用いるのが適している。状態遷移システムで、ある時相論理式が成り立つかどうかを調べるのには、モデル検査という手法がある。

この章では、時相論理とモデル検査について詳しく述べる。

### 3.1 状態遷移システム

プログラムの実行は、命令 1 ステップの実行を行うたびに、機械の内部状態が変わっていくシステムと見なすことができる。機械の内部状態それぞれを 1 つの状態と考えると、これは刻々と状態を変えていくシステムであるといえる。

また、2.2 節で説明した制御フローグラフで、ブロックから辺を辿って別のブロックに行くことは、プログラムを抽象的に実行していることに当たるが、これも状態から状態へ遷移していくシステムであるといえる。

これらのように、刻々と状態を遷移していくようなシステムを状態遷移システムという。状態数が有限の状態遷移システムは、定義 3.1 のように記号的に定義することができる<sup>\*1</sup>。

定義 3.1 (状態遷移システム)

状態遷移システムは、2 つ組  $T = (S, R)$  であり、

---

<sup>\*1</sup> 状態数が無限のものは、何らかの抽象化をして状態数を有限に抑える必要がある。

- $S$  は空でない状態の集合 .
- $R \subseteq S \times S$  は状態間の遷移関係 .

である .  $(s, s') \in R$  であるとき , 状態  $s$  から状態  $s'$  に遷移可能であり , これを  $s R s'$  や ,  $s \rightarrow s'$  と書く .

定義 3.2 (状態遷移システムの経路)

状態遷移システム  $T = (S, R)$  において , 状態の無限列  $s_0, s_1, s_2, \dots$  が , 任意の  $i \geq 0$  について  $s_i \rightarrow s_{i+1}$  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  $s_0, s_1, \dots, s_m$  が , 任意の  $i (0 \leq i < m)$  について  $s_i \rightarrow s_{i+1}$  かつ  $\forall s \in S, \neg(s_m \rightarrow s)$  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 . 経路の長さは列の長さであり , 1 以上である .

状態の列  $p = s_0, s_1, \dots, s_i, \dots$  が経路であり ,  $s_i$  が確かに存在するとき ,  $s_i$  から始まる最長部分列  $s_i, s_{i+1}, \dots$  も経路であり ,  $p_i$  と表す .

状態遷移システムの各状態  $s \in S$  に , その状態で成り立つ原始命題をラベル付けしたものは , Kripke 構造というモデルとなる .

定義 3.3 (Kripke 構造)

原始命題全体の集合を  $AP$  とする . 状態遷移システム  $T = (S, R)$  の状態  $s$  で成り立つ原始命題の集合  $L(s)$  を与える写像を  $L: S \rightarrow 2^{AP}$  とすると\*2 , 三つ組  $M = (S, R, L)$  は Kripke 構造である .

## 3.2 時相論理

時相論理は様相論理の一種であり , 状態遷移システム上で成り立つ性質を記述するのに適した論理である . 時相論理には様々な種類が存在するが , 基本的に Kripke 構造上において意味を定義される .

この節では本研究に関連するいくつかの時相論理を説明する .

---

\*2 つまり  $\alpha \in L(s)$  ならば , 原始命題  $\alpha$  は状態  $s$  で成り立つ .

### 3.2.1 線形時間時相論理 — LTL

線形時間時相論理 (Linear-time Temporal Logic, LTL) [25] とは, 状態遷移システムの単一の経路に関する性質を記述する論理である.

状態遷移システムの経路を時系列だと考えれば, 単一の経路は (分岐がないという意味で) 線形時間であり, 線形時間を扱う論理を特に線形時間論理という. LTL は線形時間論理である.

#### LTL の構文

LTL の構文は表 3.1 の通りである. ただし, *proposition* は原始命題を表す.

<i>LTL-formula</i>	::=	<i>proposition</i>
		$\neg$ <i>LTL-formula</i>
		<i>LTL-formula</i> $\wedge$ <i>LTL-formula</i>
		$X$ <i>LTL-formula</i>
		$G$ <i>LTL-formula</i>
		<i>LTL-formula</i> $U$ <i>LTL-formula</i>

表 3.1 LTL の構文規則

ここで,  $X, G, U$  は時相演算子と呼ばれるものであり, それぞれ, *neXt*, *Globally*, *Until* という英単語に由来している.

#### LTL の意味論

Kripke 構造  $M = (S, R, L)$  の経路  $p = s_0 \rightarrow s_1 \rightarrow \dots$  で LTL 式  $\phi$  が成り立つことを,  $M, p \models \phi$ , または  $M$  を省略して  $p \models \phi$  と表す. 時相演算子の直観的な意味は次の通りである.

- $p \models X\phi$  : 経路  $p$  の先頭の次の状態で  $\phi$  が成立する.
- $p \models G\phi$  : 経路  $p$  上の全ての状態で  $\phi$  が成立する.
- $p \models \phi_1 U \phi_2$  : 経路  $p$  上で,  $\phi_2$  が成立するまで  $\phi_1$  が成立し続ける.

LTL の正確な意味の定義は, 表 3.2 の通りである.

$p \models \text{proposition}$	iff	$\text{proposition} \in L(s_0)$
$p \models \neg\phi$	iff	not $p \models \phi$
$p \models \phi_1 \wedge \phi_2$	iff	$p \models \phi_1$ and $p \models \phi_2$
$p \models X\phi$	iff	$p_1 \models \phi$
$p \models G\phi$	iff	for any $i \geq 0$ , $p_i \models \phi$ and $p_{i+1}$ exists
$p \models \phi_1 U \phi_2$	iff	$p_i \models \phi_2$ for some $i \geq 0$ , and $p_j \models \phi_1$ for any $j$ such that $0 \leq j < i$

表 3.2 LTL の意味定義

### LTL の演繹体系

LTL では、構文規則に表れる演算子以外に、よく使われる演算子が表 3.3 のように他の演算子から定義される。

$\phi_1 \vee \phi_2$	$\equiv$	$\neg(\neg\phi_1 \wedge \neg\phi_2)$
$\phi_1 \rightarrow \phi_2$	$\equiv$	$\neg\phi_1 \vee \phi_2$
$F\phi$	$\equiv$	$\text{true} U \phi$
$\phi_1 W \phi_2$	$\equiv$	$(\phi_1 U \phi_2) \vee (G\phi_1)$

表 3.3 LTL の演繹体系

$F, W$  も時相演算子であり、それぞれ、Future、Weak until という英単語に由来している。それぞれ、直観的には、

- $p \models F\phi$  : 経路  $p$  上で、いつか  $\phi$  が成立する。
- $p \models \phi_1 W \phi_2$  : 経路  $p$  上で、 $\phi_2$  が成立するまで  $\phi_1$  が成立し続ける。  
 $U$  と違い、 $\phi_2$  がずっと成立しなくてもよい。

となる。

### LTL 式の例

図 3.1 は、LTL 式とそれを満たす経路の例である。



図 3.1 LTL 式の例

### 3.2.2 計算木論理 — CTL

計算木論理 (Computational Tree Logic, CTL) [6] とは, LTL の  $X, U$  などの時相演算子に加えて, 経路に対する限量子  $E, A$  を導入した論理で, 複数の経路に関する性質を記述できる論理である.

複数の経路を分岐した時系列と考えれば, CTL は分岐時間を扱う論理であるといえる. このように, 分岐時間を扱う論理を, 線形時間論理に対して分岐時間論理という.

#### CTL の構文

CTL の構文は表 3.4 の通りである. *state-formula* は, 状態に関する式で状態式という. *path-formula* は, 経路に関する式で経路式という.

<i>CTL-formula</i>	::=	<i>state-formula</i>
<i>state-formula</i>	::=	<i>proposition</i>
		$\neg$ <i>state-formula</i>
		<i>state-formula</i> $\wedge$ <i>state-formula</i>
		$E$ <i>path-formula</i>
		$A$ <i>path-formula</i>
<i>path-formula</i>	::=	$X$ <i>state-formula</i>
		$G$ <i>state-formula</i>
		<i>state-formula</i> $U$ <i>state-formula</i>
		<i>state-formula</i> $W$ <i>state-formula</i>

表 3.4 CTL の構文規則

$E, A$  は経路限量子であり, それぞれ, All, Exists という英単語に由来している. CTL の特徴として, 経路限量子と時相演算子を組で用いる構文しか許されておらず, いくつか

ある分岐時間論理の中でも特に記述力が低い。

LTL の構文規則と違い, CTL の構文規則には  $W$  演算子が明示的に入っている。もし, LTL と同様に  $W$  を  $U$  と  $G$  から定義してしまうと,  $A(\phi_1 W \phi_2) = A((\phi_1 U \phi_2) \vee (G\phi_2))$  のような式が書けることになってしまう。これは, 経路限量子と時相演算子が組となるという CTL の構文規則に従っていない。

### CTL の意味論

Kripke 構造  $M$  の状態  $s$  で状態式  $\phi$  が成り立つことを,  $M, s \models \phi$  と表す。同様に, 経路  $p$  で経路式  $\psi$  が成り立つことを,  $M, p \models \psi$  と表す<sup>\*3</sup>。

CTL の時相演算子も, 直観的には LTL と同様の意味となる。CTL の正確な意味の定義は, 表 3.5 の通りである。ただし,  $s$  は状態,  $p$  は経路とする。

<b>state formula</b>	
$s \models \text{proposition}$	iff $\text{proposition} \in L(s)$
$s \models \neg\phi$	iff not $s \models \phi$
$s \models \phi_1 \wedge \phi_2$	iff $s \models \phi_1$ and $s \models \phi_2$
$s \models E\psi$	iff $p \models \psi$ for some path $p = s \rightarrow s_1 \rightarrow \dots$
$s \models A\psi$	iff $p \models \psi$ for any path $p = s \rightarrow s_1 \rightarrow \dots$
<b>path formula</b> ( $p = s_0 \rightarrow s_1 \rightarrow \dots$ )	
$p \models X\phi$	iff $s_1$ exists and $s_1 \models \phi$
$p \models G\phi$	iff for any $i \geq 0$ , $s_i \models \phi$ and $s_{i+1}$ exists
$p \models \phi_1 U \phi_2$	iff $s_i \models \phi_2$ for some $i \geq 0$ , and $s_j \models \phi_1$ for any $j$ such that $0 \leq j < i$
$p \models \phi_1 W \phi_2$	iff $p \models \phi_1 U \phi_2$ or $p \models G\phi_1$

表 3.5 CTL の意味定義

### CTL の演繹体系

CTL でも LTL と同様, 構文規則に表れる演算子以外に, よく使われる演算子が表 3.6 のように他の演算子から定義される。

<sup>\*3</sup> LTL と同様,  $M$  を省略することもある。

$\phi_1 \vee \phi_2$	$\equiv$	$\neg(\neg\phi_1 \wedge \neg\phi_2)$
$\phi_1 \rightarrow \phi_2$	$\equiv$	$\neg\phi_1 \vee \phi_2$
$F\phi$	$\equiv$	$true U \phi$

表 3.6 CTL の演繹体系

### CTL 式の例

図 3.2 は、Kripke 構造上で CTL 式が成り立つ状態の集合を表したものである。

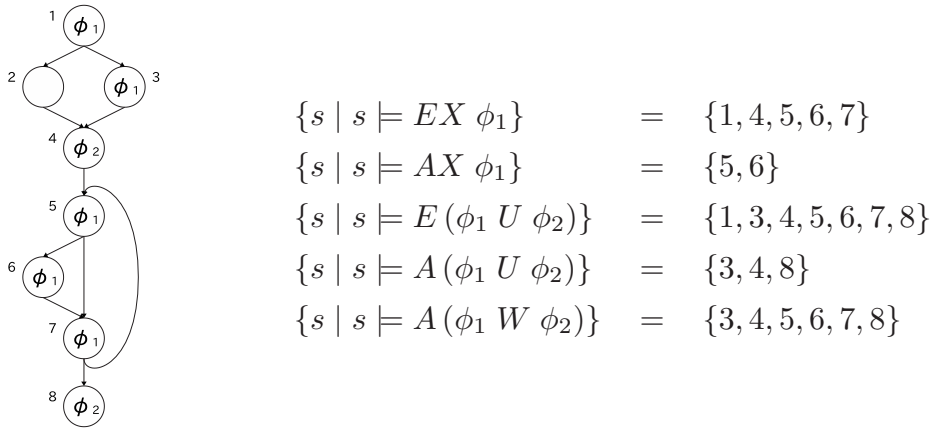


図 3.2 CTL 式の例

### 3.2.3 CTL\*

CTL\*とは、LTL に経路限量子  $E$  と  $A$  を直接導入したものであり、CTL の「経路限量子と時相演算子は組になる必要がある」という制限がなくなった論理である。CTL\*は分岐時間論理である。

CTL\*の記述力は LTL と CTL を完全に包含する。簡単に言ってしまえば、CTL\*とは LTL と CTL を合わせたような論理と言える。

#### CTL\*の構文

CTL\*の構文は表 3.7 の通りである。CTL と同様、*state-formula* は状態式、*path-formula* は経路式である。

$CTL^*$ -formula	::=	state-formula
state-formula	::=	proposition
		$\neg$ state-formula
		state-formula $\wedge$ state-formula
		$E$ path-formula
		$A$ path-formula
path-formula	::=	$CTL^*$ -formula
		$\neg$ path-formula
		path-formula $\wedge$ path-formula
		$X$ path-formula
		$G$ path-formula
		path-formula $U$ path-formula

表 3.7  $CTL^*$ の構文規則

### CTL\*の意味

$CTL^*$ の正確な意味は、Kripke 構造  $M = (S, R, L)$  に対して、表 3.8 のように定義される。ただし、 $s$  は状態、 $p$  は経路とする。状態式の意味は CTL と同様であり、経路式の意味は、LTL と同様である。

### CTL\*の演繹体系

$CTL^*$ も、LTL や CTL と同様、構文規則に表れる演算子以外に、よく使われる演算子が表 3.9 のように他の演算子から定義される。

## 3.3 モデル検査

モデル検査とは、形式的検証手法の一つであり、状態遷移システムの状態空間を網羅的に探索することにより、システムの状態がある性質を満たすことを検証する手法である。

Kripke 構造  $M = (S, R, L)$  と満たすべき時相論理式  $\phi$  が与えられたとき、モデル検査を行うことで、ある状態  $s \in S$  が  $\phi$  を満たすかどうか、つまり

$$M, s \models \phi$$

### state formula

$s \models \text{proposition}$	iff	$\text{proposition} \in L(s)$
$s \models \neg\phi$	iff	not $s \models \phi$
$s \models \phi_1 \wedge \phi_2$	iff	$s \models \phi_1$ and $s \models \phi_2$
$s \models E\psi$	iff	$p \models \psi$ for some path $p = s \rightarrow s_1 \rightarrow \dots$
$s \models A\psi$	iff	$p \models \psi$ for any path $p = s \rightarrow s_1 \rightarrow \dots$

### path formula ( $p = s_0 \rightarrow s_1 \rightarrow \dots$ )

$p \models \phi$	iff	$s_0 \models \phi$
$p \models \neg\psi$	iff	not $p \models \psi$
$p \models \psi_1 \wedge \psi_2$	iff	$p \models \psi_1$ and $p \models \psi_2$
$p \models X\psi$	iff	$p_1 \models \psi$
$p \models G\psi$	iff	for any $i \geq 0$ , $p_i \models \psi$ and $p_{i+1}$ exists
$p \models \psi_1 U \psi_2$	iff	$p_i \models \psi_2$ for some $i \geq 0$ , and $p_j \models \psi_1$ for any $j$ such that $0 \leq j < i$

表 3.8 CTL\*の意味定義

$\phi_1 \vee \phi_2$	$\equiv$	$\neg(\neg\phi_1 \wedge \neg\phi_2)$
$\phi_1 \rightarrow \phi_2$	$\equiv$	$\neg\phi_1 \vee \phi_2$
$F\phi$	$\equiv$	$\text{true} U \phi$
$\phi_1 W \phi_2$	$\equiv$	$(\phi_1 U \phi_2) \vee (G\phi_1)$

表 3.9 CTL\*の演繹体系

が成り立つかどうかを検証できる。実際には、一つの状態  $s$  を固定して検査するのではなく、 $\phi$  を満たすような状態の集合

$$\{s \in S \mid M, s \models \phi\}$$

を求めて、この集合にある状態が入っているかどうかを調べる方法が一般的である。

### 3.3.1 モデル検査の計算量

モデル検査のアルゴリズムは、それぞれの時相論理についていくつかのアルゴリズムが提案されているようである [7]。現在の最もよいとされるアルゴリズムの計算量は、モデ

ル  $(S, R, L)$ , 式  $f$  について, それぞれ次の通りである. ただし,  $|f|$  は  $f$  の部分式の個数とする.

$$\begin{aligned} \text{LTL} & : O((|S| + |R|) \times 2^{O(|f|)}) \\ \text{CTL} & : O(|f| \times (|S| + |R|)) \\ \text{CTL}^* & : O((|S| + |R|) \times 2^{O(|f|)}) \end{aligned}$$

CTL は, モデルと式のサイズについて線形であるが, LTL と CTL\* は, モデルのサイズについては線形であるが, 式のサイズについては指数的である.

### 3.3.2 $\mu$ 計算への変換

$\mu$  計算とは, 遷移システムの性質を, 最小不動点および最大不動点の演算を使用することによって表現する強力な言語である. 多くの時相論理と様相論理に対する検証手続きは,  $\mu$  計算に変換して記述することができる.  $\mu$  計算による検証は, 順序付き二分決定グラフを併せて用いることで, 効率よく行うことができる.

例えば,  $AU$  に対するモデル検査は,

$$A(\phi_1 U \phi_2) \equiv \phi_2 \vee (\phi_1 \wedge AX A(\phi_1 U \phi_2))$$

という等価性を用いて, 次の  $\mu$  計算により記述できる.

$$\{s \mid s \models A(\phi_1 U \phi_2)\} = \mu.Z(\phi_2 \vee (\phi_1 \wedge \square Z))$$

ここで,  $\square$  は,  $AX$  に相当する様相記号である.

文献 [31] によると,  $\mu$  計算による検証は, データフロー方程式と高い類似性がある.

## 第III部

# 提案手法

## 第 4 章

# 提案手法で採用する時相論理

結論からいうと，本手法では Lacey らと同様，CTL-FV[18] という時相論理を採用する．これは，3.2.2 節で説明した CTL を拡張した論理である．

この章では，本手法で CTL-FV を採用した理由を順を追って説明する．

### 4.1 プログラムのモデル化

採用する時相論理の説明をする前に，時相論理で性質を記述する対象であるプログラムのモデル化について，詳細を述べる．

3.1 節でも述べたが，プログラムの実行は状態遷移システムとしてモデル化できる．最も単純には，命令 1 ステップの実行ごとに変わっていく機械の内部状態を，システムの 1 つの状態としてモデル化することが考えられる．しかし，このような具体的なモデル化では状態の数が膨大となり，すぐにモデル検査が不可能となってしまう．また，コンパイル時にはこのような具体的な実行に関する情報は得られていないので，このようなモデル化ではコンパイル時の検査は不可能である．

そこで，プログラムの実行をなんらかの方法で抽象化して，それを状態遷移システムとしてモデル化する必要がある．抽象化には様々な方法が考えられるが，最適化は 2.2 節で述べた制御フローグラフ上で行われることが多いので，プログラムの制御フローグラフを状態遷移システムと見るのが自然である．この制御フローグラフを元に，時相論理で性質を記述する対象のモデルを定義する．

### 4.1.1 制御フローモデル

制御フローグラフは、次の制御フローモデルにモデル化される。

定義 4.1 (制御フローモデル)

一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。原始命題全体の集合を  $AP$  とし、ノード  $n \in N$  で成り立つ原始命題の集合  $L(n)$  を与える写像を  $L: N \rightarrow 2^{AP}$  とする。三つ組  $M = (N, E, L)$  を制御フローモデルという。これは Kripke 構造である。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。

### 4.1.2 原始命題の種類

本手法で用いる論理では、最適化に関する性質を記述するのに有用と思われる原始命題を採用した。主なものは、ノード番号や基本ブロック名、変数や式の使用や定義などで、文献 [18] や [12] と共通のものが多い。

本手法で特徴的なのは、マークが付いていることを表す原始命題  $mark$  である。マークについては 5 章で詳しく述べるが、簡単にいうと、最適化による変形箇所を覚えておくために付けられるものである。マーク  $M$  が付いているノードでは  $mark(M)$  という原始命題が成り立つ、という意味として解釈される。

ここで、本手法で用いる原始命題を列挙し、その直観的な意味を与える。制御フローモデル  $M = (N, E, L)$  のノード  $n \in N$  について、 $n$  で成り立つ原始命題の集合  $L(n)$  は表 4.1 のように定義される。

$L(n)$	=	{	$node(N)$		$n = N$	}
		∪	{	$block(B)$		$n$ は基本ブロック $B$ の文のノード }
		∪	{	$use(X)$		$X$ は $n$ で「使用される」変数 }
		∪	{	$def(X)$		$X$ は $n$ で「定義される」変数 }
		∪	{	$comp(E)$		$E$ は $n$ で「使用される」式 }
		∪	{	$trans(E)$		$E$ は $n$ で「変更されない」式、 つまり $E$ の全てのオペランドは $n$ で定義されない }
		∪	{	$mark(M)$		$n$ にはマーク $M$ が付いている }

表 4.1  $L(n)$  の直観的な意味の定義

表 4.1 に現れる「使用される」「定義される」といった言葉は、対象のプログラム言語の意味が定まってはじめてきちんと定義されるものである。本研究では、COINS コンパイラの LIR という中間表現に対して提案手法を適用した実装を行った。LIR についての原始命題の意味定義は 6.4 節で述べる。

## 4.2 CTL の採用

章の冒頭でも述べたが、本手法で採用する時相論理は CTL を拡張した論理である CTL-FV である。この節では、3.2 節で説明したいいくつかの時相論理の中から何故 CTL を選んだのかを説明する。

3.2 節で説明した論理を大きく分けると、

- 線形時間論理
- 分岐時間論理

の二つに分類できる。LTL が線形時間論理、CTL、CTL\*、 $\mu$  計算が分岐時間論理である。制御フローモデルはプログラムの実行を抽象化したモデルであることは既に述べたが、コンパイラの最適化をこのモデル上で定式化するには、2.3 節での例のように複数の経路を考える必要がある。よって、本手法には線形時間論理よりも分岐時間論理の方が適しているといえる。

複数の分岐時間論理の中から CTL を選ぶ理由は次の通りである。

- 経路に沿った論理で直観的に分かりやすく、証明もしやすい。
- モデル検査が簡単で、効率よく行うことができる。
- CTL の記述力でもあまり問題にならないと思われる。

本手法の概要は 1.2 節でも少し述べたが、正しさの厳密な証明は別途行う必要がある。 $\mu$  計算のような不動点を扱う論理を用いるよりも、CTL や CTL\* のような経路を扱う論理を用いる方が、Lacey らと同様の方法で証明が行えて簡単である。

CTL と CTL\* は直観的には似たような論理で、CTL\* の方が記述力が高いが、CTL の方がモデル検査が簡単で、検査の計算量や実装の際のバグの混入の可能性の低さの面で優れている。結局、記述力と効率のトレードオフとなるが、CTL の記述力でもあまり問題がないと思われる。

以上より、本手法では基本として CTL を選択した。

### 4.3 双方向経路を扱う CTL

プログラムの最適化では、例えば 2.3.1 節で説明した共通部分式除去のように、逆向きのパスを考えることがしばしばある。

時系列を扱う時相論理にとって、通常の経路が「将来」に向かう経路だとすれば、逆向きの経路は「過去」に向かう経路であるといえる。3.2.2 節で説明した CTL の定義では、「過去」つまり逆向きの経路を扱うことができないので、「過去」を扱えるように拡張する必要がある。

「将来」について、線形時間と見るか分岐時間と見るかの二通りがあったように、「過去」についても線形時間か分岐時間かの二通りが考えられる。最適化の観点から制御フローモデルの「過去」を考えると、「将来」と同様、分岐時間であることが自然であるといえる。なぜなら、ある時点での最適化変形が「過去」に依存する場合、その点に至る「過去」の全ての可能性を考える必要があるからである。よって「過去」すなわち逆向きの経路は分岐時間とする。

逆向きの経路の正確な定義は次で与えられる。

定義 4.2 (逆向きの遷移, 逆向きの経路)

3.1 節で定義した状態遷移システム  $T = (S, R)$  について、 $s \rightarrow s'$  という遷移関係があるとき、この遷移関係を逆向きに辿ることを逆向きの遷移と言い、 $s' \leftarrow s$  または  $s' \rightarrow^\circ s$  と表す。逆向きの遷移に対して、定義 3.2 と同様に逆向きの経路が定義される。

通常の経路だけでなく、定義 4.2 で定義した逆向きの経路も扱えるように、CTL に逆向きの経路限量子  $\overleftarrow{E}$ ,  $\overleftarrow{A}$  を加えて拡張する。この双方向の経路を扱える論理を便宜上  $CTL_{bd}$  と呼ぶことにする。 $CTL_{bd}$  の構文や意味は、 $\overleftarrow{E}$ ,  $\overleftarrow{A}$  以外は CTL と全く同じであり、 $\overleftarrow{E}$ ,  $\overleftarrow{A}$  については、経路が逆向きであるという点以外では  $E$ ,  $A$  と全く同様に扱われる。 $CTL_{bd}$  の構文規則を表 4.2 に、意味定義を表 4.3 に載せる。これらは、3.2.2 節で説明した CTL の構文や意味と、 $\overleftarrow{E}$ ,  $\overleftarrow{A}$  が増えている以外にほとんど違いがない。

### 4.4 自由変数の導入

例えば、「状態  $s$  から見て、変数  $x$  が使用されることがあるような経路が存在する」という性質は、 $CTL_{bd}$  を用いて

$$s \models EF use(x) \quad (4.1)$$

$CTL_{bd}\text{-formula}$	$::=$	$state\text{-formula}$
$state\text{-formula}$	$::=$	$proposition$ $  \neg state\text{-formula}$ $  state\text{-formula} \wedge state\text{-formula}$ $  E path\text{-formula}$ $  A path\text{-formula}$ $  \overleftarrow{E} path\text{-formula}$ $  \overleftarrow{A} path\text{-formula}$
$path\text{-formula}$	$::=$	$X state\text{-formula}$ $  G state\text{-formula}$ $  state\text{-formula} U state\text{-formula}$ $  state\text{-formula} W state\text{-formula}$

表 4.2  $CTL_{bd}$  の構文規則

### state formula

$s \models proposition$	iff	$proposition \in L(s)$
$s \models \neg\phi$	iff	not $s \models \phi$
$s \models \phi_1 \wedge \phi_2$	iff	$s \models \phi_1$ and $s \models \phi_2$
$s \models E\psi$	iff	$p \models \psi$ for some path $p = s \rightarrow s_1 \rightarrow \dots$
$s \models A\psi$	iff	$p \models \psi$ for any path $p = s \rightarrow s_1 \rightarrow \dots$
$s \models \overleftarrow{E}\psi$	iff	$p \models \psi$ for some path $p = s \rightarrow^\circ s_1 \rightarrow \dots$
$s \models \overleftarrow{A}\psi$	iff	$p \models \psi$ for any path $p = s \rightarrow^\circ s_1 \rightarrow \dots$
<b>path formula</b>		$(p = s_0 \rightarrow' s_1 \rightarrow' \dots$ in which $\rightarrow'$ is $\rightarrow$ or $\rightarrow^\circ)$
$p \models X\phi$	iff	$s_1$ exists and $s_1 \models \phi$
$p \models G\phi$	iff	for any $i \geq 0$ , $s_i \models \phi$ and $s_{i+1}$ exists
$p \models \phi_1 U \phi_2$	iff	$s_i \models \phi_2$ for some $i \geq 0$ , and $s_j \models \phi_1$ for any $j$ such that $0 \leq j < i$
$p \models \phi_1 W \phi_2$	iff	$p \models \phi_1 U \phi_2$ or $p \models G\phi_1$

表 4.3  $CTL_{bd}$  の意味定義

と記述できる．ここで， $x$  は実際にプログラム中に現れる変数であるとする．この式が成り立てば， $s$  において  $x$  は生存変数であるといえる．

しかし，この式中の  $x$  はあくまで実際にプログラムに現れる変数  $x$  であり，別の変数  $y$  の性質を検査することができない．このような記述のままでは，検査したい変数が確定してから論理式を記述しなければならないので，コンパイル前に変形箇所が満たすべき性質を記述するのは不可能である．

この状況を解決するために，論理式の原始命題を自由変数を引数にとれる述語として拡張する必要がある．こうすることで，「特定の変数  $x$ 」ではなく「抽象化した変数  $x$ 」としてコンパイル前に論理式を記述することができる．

式 (4.1) 中の変数  $x$  を，自由変数  $x$  として置き換えると次のようになる．

$$s \models EF\ use(x) \quad (4.2)$$

この式の  $x$  は  $x$  かもしれないし  $y$  かもしれない．この式の  $x$  を実際の変数  $x$  に結び付けることを  $x$  を  $x$  で束縛するという．また，全ての自由変数を束縛し，式 (4.1) のような自由変数のない式にすることを具体化するというにすることにする．

CTL<sub>bd</sub> の原始命題を，このように自由変数を引数にとれる述語として拡張した論理を，Lacey らは CTL-FV と呼んだ [18]．本論文では，以後，自由変数を含む CTL<sub>bd</sub> 式を CTL-FV 式，自由変数を全く含まない CTL<sub>bd</sub> 式を単に CTL 式と，区別して呼ぶことにする\*1．

---

\*1 この区別はモデル検査を実際に行う際に必要となる．

## 第 5 章

# 提案手法

この章では，提案する手法の詳細について説明する．

### 5.1 概要

本論文で提案する手法は，プログラムに対する最適化の実行を行った後に，プログラムに対する最適化が正しく行われたかどうか，つまり観測的意味を保存する最適化であったかどうかを検査する手法である．

図 5.1 に，本手法の概略図を示す．最適化が正しく行われたかどうかは，最適化による全ての変形が観測的意味を保存する変形であったかどうかによる．そこで本手法では，最適化中に全ての変形箇所に変形の種類に応じたマークを付ける．そして，マークを付けた箇所が観測的意味を保存するために満たすべき性質を本当に満たすかどうかを最適化後に

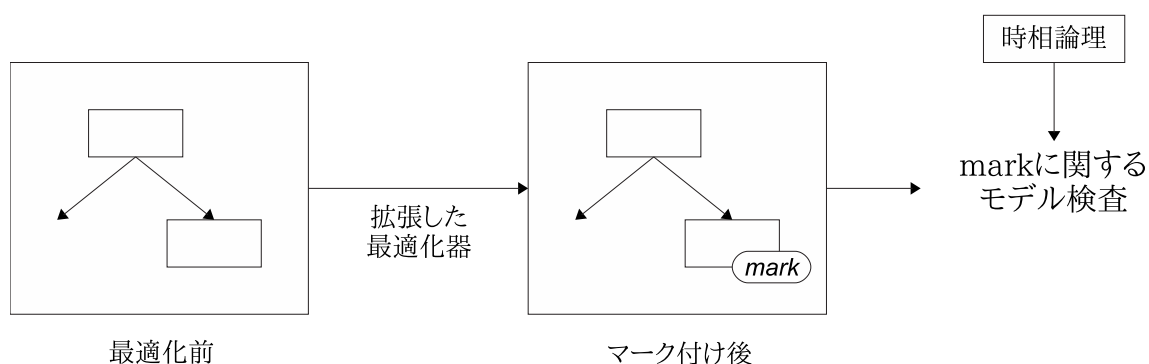


図 5.1 提案手法の概要

検査する方法をとる．この検査は，個々の最適化の実行ごとに行う．

マークをつけた箇所の満たすべき性質は，最適化ごとにあらかじめ時相論理 CTL-FV を用いて検査仕様として記述しておき，全ての変形後にモデル検査を実施して，式を満たすかどうか検査する．

提案手法は，次の 6 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. モデル検査のための CTL 式を，検査仕様と等価の CTL-FV 式を具体化することで生成する．
6. モデル検査を行う．

各ステップの詳細は，5.3 節以降で順次説明していく．説明のために，次節で説明する部分冗長除去を行う Lazy Code Motion という最適化を例に用いる．

## 5.2 Lazy Code Motion

以降の説明のため，部分冗長性を除去するアルゴリズムのひとつである Lazy Code Motion[16, 17] という最適化を例として用いる．

制御フローグラフのあるパス上で，同じ値を計算する式が複数現れるとき，式の 2 回目以降の出現は冗長な計算である．式が全てのパスで冗長である場合，これを全冗長といい，全冗長ではないが，冗長であるパスが存在するとき部分冗長という．部分冗長を除去し，実行時の計算回数を少なくする最適化を一般に部分冗長除去という．

図 5.2 は，部分冗長除去の例である．図 5.2(a) で，左側のパスを通ると  $a + b$  が 2 回計算されるが，右側のパスでは  $a + b$  は 1 回しか計算されず， $a + b$  は部分冗長となっている．この部分冗長を Lazy Code Motion によって除去すると，図 5.2(b) のようになる．

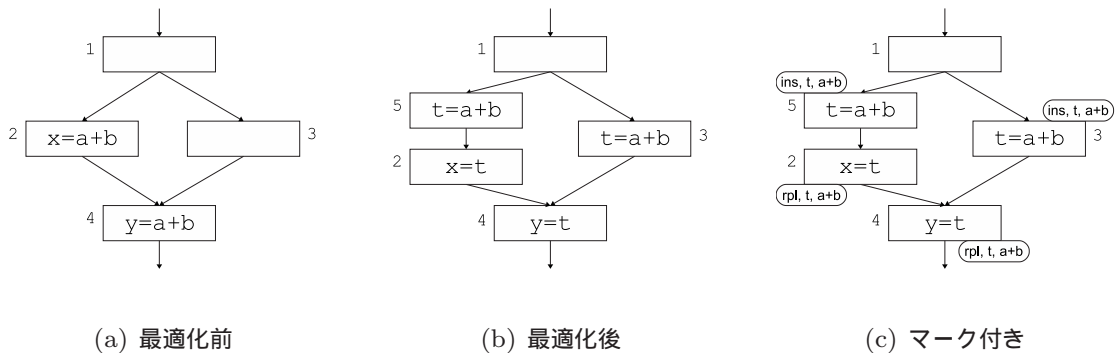


図 5.2 部分冗長除去の例

図 5.2(b) では、左と右どちらのパスを通っても  $a + b$  は 1 回しか計算されておらず、部分冗長性が除去されたのが分かる。

Lazy Code Motion における変形箇所は次の 2 箇所となる。

- $t = a + b$  を、最初の  $a + b$  の出現する前に挿入した。
- $a + b$  の元の計算を一時変数  $t$  で置換した。

それぞれを挿入点、置換点と呼ぶ。

図 5.2(c) 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  $(ins, t, a + b)$ 、置換点には  $(rpl, t, a + b)$  というマークを付けて、どのような変形をしたかが分かるようにしている。このマークに基づいて、最適化後に検査を行う。

### 5.3 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性質を、あらかじめ記述しておく必要がある。この記述を、検査仕様、または単に仕様と呼ぶことにする。変形の仕方は当然最適化によって異なるので、仕様は最適化ごとに個別に記述する必要がある。

最適化の正しさを検証するためには、5.1 節でも述べたように、変形箇所全てが意味を保存することを言えばいいので、仕様はマークの種類ごとに記述するのが自然である。本手法では、変形箇所が満たすべき性質を CTL-FV で記述する。よって、マークを  $m$ 、それに対応する CTL-FV 式を  $\phi$  とすれば、仕様はこれらの組  $\langle m, \phi \rangle$  と表される。また、便

宜上、次のように表すこともある。

MARK	$m$
FORMULA	$\phi$

### 5.3.1 仕様の構文

仕様の構文の定義は表 5.1 の通りである。 *specification* が仕様を、 *formula* が CTL-FV 式を表す。また、 *predicate* は述語を表す。 *symbol* は文字列記号である。 CTL-FV の構文は、 4.3 節の表 4.2 で定義した  $CTL_{bd}$  の構文とほとんど同じである。

### 5.3.2 仕様の意味

仕様  $\langle m, \phi \rangle$  は、「マーク  $m$  の付いたノードでは  $\phi$  が成り立つ」という意味である。制御フローモデル  $M = (N, E, L)$  に対して、仕様の意味を正確に定義すると次のようになる。

$$\forall n \in N, \quad mark(m) \in L(n) \Rightarrow M, n \models \phi \quad (5.1)$$

これは次のように表すこともできる。

$$\forall n \in N, \quad M, n \models mark(m) \rightarrow \phi \quad (5.2)$$

この場合、  $n$  は省略しても意味は同じである。

CTL-FV 式の意味は 4.3 節の表 4.3 で定義した通りである。述語の意味は、 4.1.2 節でも述べたが正確な定義は対象のプログラム言語に依存する。直観的な意味についても 4.1.2 節の表 4.1 ですでに示したので、ここでは省略する。

### 5.3.3 Lazy Code Motion の検査仕様の記述

検査仕様の記述の例として、図 5.2 の Lazy Code Motion の変形の正しさについて簡単に述べ<sup>\*1</sup>、検査仕様の記述を考えていく。

Lazy Code Motion における変形箇所は次の 2 箇所であった。

- $t = a + b$  を、最初の  $a + b$  の出現する前に挿入した。
- $a + b$  の元の使用を一時変数  $t$  で置換した。

---

<sup>\*1</sup> 詳細については文献 [17] を参照されたい。

<i>specification</i>	$::=$	$\langle \textit{mark}, \textit{CTL-FV-formula} \rangle$
<i>CTL-FV-formula</i>	$::=$	<i>state-formula</i>
<i>state-formula</i>	$::=$	<i>true</i>
		<i>false</i>
		<i>predicate</i>
		$\neg$ <i>state-formula</i>
		<i>state-formula</i> $\wedge$ <i>state-formula</i>
		<i>E path-formula</i>
		<i>A path-formula</i>
		$\overleftarrow{E}$ <i>path-formula</i>
		$\overleftarrow{A}$ <i>path-formula</i>
<i>path-formula</i>	$::=$	<i>X state-formula</i>
		<i>G state-formula</i>
		<i>state-formula U state-formula</i>
		<i>state-formula W state-formula</i>
<i>predicate</i>	$::=$	<i>node(symbol)</i>
		<i>block(symbol)</i>
		<i>use(symbol)</i>
		<i>def(symbol)</i>
		<i>comp(symbol)</i>
		<i>trans(symbol)</i>
		<i>mark</i>
<i>mark</i>	$::=$	<i>mark(symbol</i> [, <i>symbol</i> ]* <i>)</i>

表 5.1 仕様の構文規則

以降の議論を簡単にするため，一時変数  $t$  は最適化前のプログラムには出現しない変数である，と仮定する．

#### 命令を挿入した箇所

文献 [17] によると， $t = a + b$  の挿入点  $n_1$  は次のいずれかの条件を満たすとき正しい．

1.  $n_1$  の全ての先行パス上に，同じ値を計算する式  $a + b$  が存在する．
2.  $n_1$  の全ての後続パス上に，同じ値を計算する式  $a + b$  が存在する．

ただし，Lazy Code Motion は先行パスに同じ値を計算する式があるような場所には挿入しないので，実際には条件は厳しくなるが条件 2 だけで十分である．

ところで，後続パス上の同じ値を計算する  $a + b$  は，最適化により一時変数  $t$  に置き換えられるはずである．この点は置換点ということになる．この置換点には，最適化中のマーク付けにより  $(rpl, t, a + b)$  というマークが付いているはずである．つまり，挿入点からは  $a + b$  の値が変わることなくマーク  $(rpl, t, a + b)$  の付いたノードに到達するということになる．

以上から，一時変数  $t$  を  $t$ ，式  $a + b$  を  $e$  という自由変数で表すと，満たすべき仕様は，

$$\begin{aligned} \text{MARK} & \quad (\text{ins}, t, e) \\ \text{FORMULA} & \quad A(\text{trans}(e) \ W \ \text{mark}(rpl, t, e)) \end{aligned} \tag{5.3}$$

となる\*2．

$U$  演算子ではなく  $W$  演算子である理由は，ループを回り続ける経路のような，終端に至ることのない無限経路を考慮しないためである．マーク  $(\text{ins}, t, e)$  の付いたノードとマーク  $(rpl, t, e)$  の付いたノードの間のパスにループが存在すると，このループを回り続ける経路が存在することになる．この経路上では  $\text{mark}(rpl, t, e)$  が成立することはないので， $A(\text{trans}(e) \ U \ \text{mark}(rpl, t, e))$  という式では，この経路の存在により成立しないことになる．最適化では通常，そのような終端に至ることのない経路を考慮しないので， $U$  ではなく  $W$  を用いる必要がある．

---

\*2 「 $t$  は最適化前のプログラムには出現しない」という仮定ができない場合，FORMULA に加えて，

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

という式も必要である．この式は，直観的には「 $t$  が再定義されることなく置換点以外で使用されるようなパスは存在しない」という意味である．これにより，他で使用される  $t$  の値が  $t = a + b$  の挿入により変更されることがないことを保証できる．

式の計算を置換した箇所

文献 [17] によると,  $a + b$  の  $t$  による置換点  $n_2$  は次の条件を満たすとき正しい.

- $n_2$  の全ての先行パス上に挿入点が存在し, その点まで  $a + b$  の値は変更されない.

つまり, 挿入点で  $t$  により利用可能になった  $a + b$  の値が, 利用可能なまま  $n_2$  に到達するということである.  $a + b$  の値が  $t$  により利用可能であるためには, 逆向きの経路上で挿入点に至るまでに  $a + b$  の値の変更, および  $t$  の定義がないことが条件となる.

以上から, 一時変数  $t$  を  $t$ , 式  $a + b$  を  $e$  という自由変数で表すと, 満たすべき条件式  $\phi_2$  は,

$$\begin{array}{ll} \text{MARK} & (\text{rpl}, t, e) \\ \text{FORMULA} & \overleftarrow{A} ((-def(t) \wedge trans(e)) \ W \ mark(\text{ins}, t, e)) \end{array} \quad (5.4)$$

となる.  $U$  演算子ではなく  $W$  演算子である理由は, 命令の挿入箇所と同様, ループを回り続ける経路のような無限経路を考慮しないためである.

## 5.4 ステップ 2 – 既存の最適化器の拡張

本手法では, 最適化実行時のマーク付けを実現するために, 最適化器のコードを拡張する. 具体的には, 最適化器のコード中のプログラムの変形を行う部分のコードの前後に, マークを付けるためのコードを挿入する. 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.

<pre> 1: (<math>P_{ins}, P_{rpl}</math>) <math>\leftarrow</math> analyze() 2: <b>for all</b> <math>p \in P_{ins}</math> <b>do</b> 3:   insert <math>t = e</math> at <math>p</math> 4: <b>for all</b> <math>q \in P_{rpl}</math> <b>do</b> 5:   replace <math>e</math> with <math>t</math> at <math>q</math> </pre>	<pre> 1: (<math>P_{ins}, P_{rpl}</math>) <math>\leftarrow</math> analyze() 2: <b>for all</b> <math>p \in P_{ins}</math> <b>do</b> 3:   insert <math>t = e</math> at <math>p</math> 4:   <math>marking_{ins}(p)</math> 5: <b>for all</b> <math>q \in P_{rpl}</math> <b>do</b> 6:   replace <math>e</math> with <math>t</math> at <math>q</math> 7:   <math>marking_{rpl}(q)</math> </pre>
(a) 拡張前の最適化器コード	(b) 拡張後の最適化器コード

図 5.3 最適化器の拡張の例

図 5.3 は最適化器の拡張の例である．図 5.2 の Lazy Code Motion の例では，最適化による変形は，命令を挿入した箇所と，式の使用を置換した箇所の 2 箇所であった．Lazy Code Motion を行う最適化器が図 5.3(a) のように実装されていたとする．*analyze()* はデータフロー解析を表し，挿入箇所の集合  $P_{ins}$  と置換箇所の集合  $P_{rpl}$  を求める関数とする．この場合，

- 命令を挿入する箇所  
マーク  $(ins, t, e)$  を付ける関数  $marking_{ins}(p)$  を挿入する．
- 式を置換する箇所  
マーク  $(rpl, t, e)$  を付ける関数  $marking_{rpl}(q)$  を挿入する．

とすることにより，最適化中にマークを付けられるようになる．

#### 5.4.1 アスペクト指向プログラミングの利用

このコード挿入による最適化器の拡張は，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を直接改変することをほとんど行う必要なく拡張することができる．詳細は，7.4 で述べる．

### 5.5 ステップ 3 – モデルの生成

本手法では，4.1 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．制御フローグラフは，そのままでも制御フローモデルと考えることもできるので，最適化後の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，次のような問題がある．

- 文を削除するような最適化<sup>\*3</sup>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処理系もあるだろうが，一般にはノードごと削除する処理系が多いと思われるので，この問題に対処する必要がある．

---

\*3 無用命令除去など．

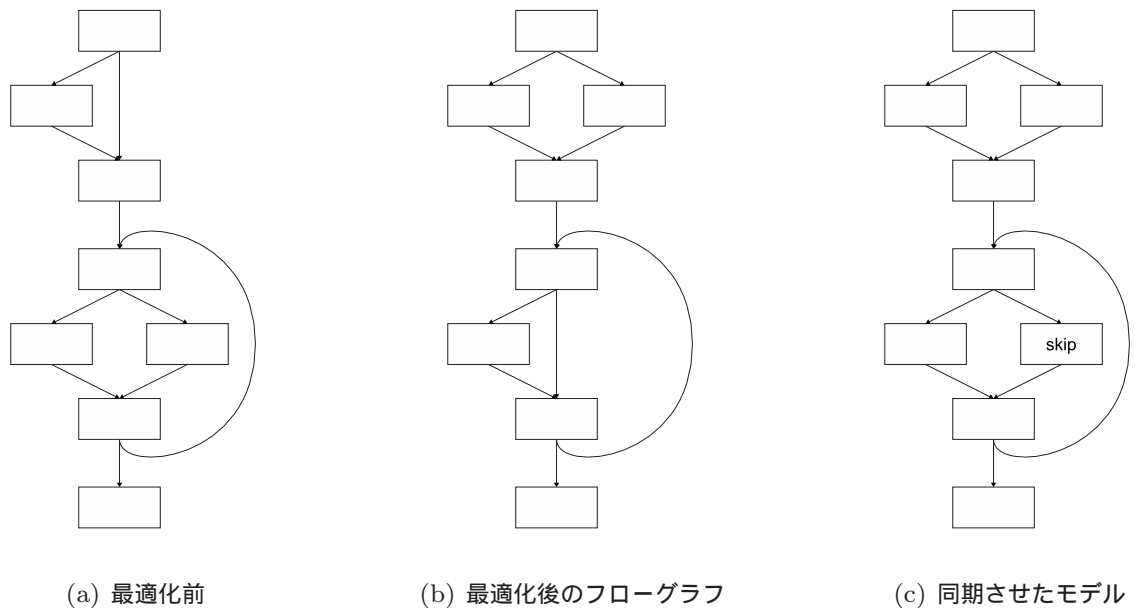


図 5.4 制御フローグラフの変更にモデルを同期させる例

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。

図 5.4 は、元のプログラムの変形に同期してモデルにも変形を加える例である。図 5.4(b) に追加されたノードは図 5.4(c) にも追加され、図 5.4(b) で削除されたノードは図 5.4(c) では「skip」となっている。このように、コピーしたモデルでは、ノードの削除を行わないので、最適化後に文を削除した箇所の正しさの検査が可能である。

モデルの変形は、マーク付けを行う際に同時に行う。このためのコードも最適化器に挿入する必要があるが、5.4 節で、既に変形箇所にマーク付けのコードを挿入しているので、このマーク付けのコードとともに挿入することになる。このコード挿入の詳細は、7.4 節で述べる。

本実装に用いた COINS コンパイラ [8] では、文の削除の際はノードごと削除する最適化器が実装されていたので、上記の方法を用いて問題に対処した。

## 5.6 ステップ 4 – 変形箇所のマーク付け

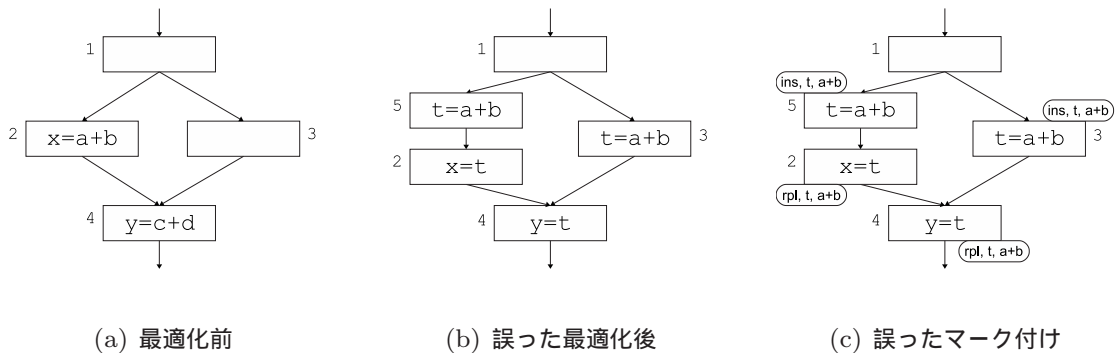


図 5.5 誤ったマーク付けの例

5.4 節で述べた最適化器の拡張により，最適化中に，図 5.2(c) のように変形に応じたマークを付けていく．

### 5.6.1 マーク付けの際の注意点

マークを付ける際は，本当にそのマークで正しいのか注意する必要がある．図 5.5 は，Lazy Code Motion の誤った最適化の例である．5.5(b) では， $c + d$  を  $a + b$  と間違えて  $t$  に置き換えてしまっている．これは当然間違いで，最適化器が  $c + d$  を  $a + b$  と誤って認識してしまったバグである．しかし「 $c + d$  を  $t$  に置き換えた」という事実ではなく，「 $a + b$  を  $t$  に置き換えた」という最適化器の誤った情報を信じてマークを付けてしてしまうと，図 5.5(c) のようにマークが付いてしまう．これでは，図 5.2(c) と区別がつかず，本来誤った最適化であったはずなのに，検査を通過してしまう．

この事態を避けるために，マーク付けは慎重に行う必要がある．マーク  $(rpl, t, a + b)$  を付ける場合なら，

- 置換する前の式は確かに  $a + b$  であった．
- 置換した後の変数は確かに  $t$  であった．

ということをして，マーク付けの際にきちんと調べる必要がある．

### 5.6.2 解析箇所のマーク付け

これまでの説明で，変形箇所には全てマークを付けると述べたが，マークを付けるのは変形箇所に限定する必要はない．例えば，ループ解析モジュールにより「このノードはループのヘッダである」という答えが得られた場合，この答えが本当に正しいか検査する

ために、該当ノードにマークを付け、後で検査することも可能である。

本手法を実際に適用した際には、このように解析箇所マークを付ける場合がいくつかあった。実際の例は、付録 A で簡単に説明する。

## 5.7 ステップ 5 - CTL 式の生成

検査仕様は、式 (5.2) の式と意味が等しいことは 5.3.2 節ですでに述べた。よって、仕様を満たすかどうか検証するには、仕様から式 (5.2) の形の式を生成し、この生成した式でモデル検査を行えばよい。

しかし、仕様から式 (5.2) の形の式を生成しても、これは自由変数を含んだ CTL-FV の式であり、直接モデル検査することはできない。従って、式を具体化し、自由変数を含まない CTL 式とする必要がある。

具体化するためには、最適化中に付けたマークを利用するのが効率がよい。最適化中に実際に付けたマークの一つを  $m$  とすると、それに対応する仕様  $\langle m_{fv}, \phi \rangle$  が存在し、 $m$  と  $m_{fv}$  の引数の同じ位置にあるシンボルはそれぞれ対応していることになる。この対応関係で束縛することで、具体化された CTL 式が得られ検査可能になる。詳しくは、以下の 5.7.1 節で具体例を挙げて説明する。

対応関係で束縛しきれなかったシンボルは、プログラム中に現れるシンボルを順に選び、束縛していくことになる。この際、束縛しきれなかったシンボルがあれば、検査する CTL 式が組み合わせ級数的に増大する。実際に、文献 [12] では、CTL-FV 式中で束縛しきれなかった自由変数が一つ増えるだけで、モデル検査にかかる時間が大きく増加すると報告されている。

しかし本手法では、付けるマークを工夫することにより、束縛漏れをほぼ 0 に抑えることが可能である。これについては、7.1 節で例を挙げて説明する。

### 5.7.1 Lazy Code Motion での例

図 5.2(c) の例では、二つのマークを付けた。

マーク (ins, t, a + b) に対応する仕様は、式 (5.3) であり、次の通りであった。

MARK (ins, t, e)  
FORMULA  $A(trans(e) W mark(rpl, t, e))$

仕様のマーク  $(ins, t, e)$  と実際につけたマーク  $(ins, t, a + b)$  の対応を取ると,  $t$  が  $t$  に,  $e$  が  $a + b$  に対応しているのが分かる. よって, この対応に従って 5.3 中の自由変数を束縛し具体化すればよい. 具体化した仕様と等価な CTL 式は, 式 (5.2) より式 (5.5) となる.

$$\begin{aligned} M \models & mark(ins, t, a + b) \\ & \rightarrow A(trans(a + b) W mark(rpl, t, a + b)) \end{aligned} \quad (5.5)$$

マーク  $(rpl, t, a + b)$  についても同様にして CTL 式を得る. 対応する仕様は式 (5.4) であり, 次の通りであった.

$$\begin{array}{ll} \text{MARK} & (rpl, t, e) \\ \text{FORMULA} & \overleftarrow{A} ((\neg def(t) \wedge trans(e)) W mark(ins, t, e)) \end{array}$$

この仕様を具体化したものと等価な CTL 式は式 (5.6) となる.

$$\begin{aligned} M \models & mark(rpl, t, a + b) \\ & \rightarrow \overleftarrow{A} ((\neg def(t) \wedge trans(a + b)) W mark(ins, t, a + b)) \end{aligned} \quad (5.6)$$

#### 複数箇所を变形する例

図 5.2 の例は, 变形は二種類の仕様それぞれに対して一つずつであった. しかし, 仕様に対する变形の箇所が複数になっても基本は変わらない.

図 5.6 は, 複数の式の部分冗長性を除去する Lazy Code Motion の例である. この例では,  $a + b$  と  $c + d$  の二つの式に対する部分冗長性を除去していて, 变形箇所に付けたマークは, 次の四種類である.

- $(ins, t, a + b)$
- $(ins, u, c + d)$
- $(rpl, t, a + b)$
- $(rpl, u, c + d)$

この場合でも 5.7.1 節の方法と同様に, 仕様を具体化し, 検査する CTL 式を生成する. すると次の四つの式が得られる.

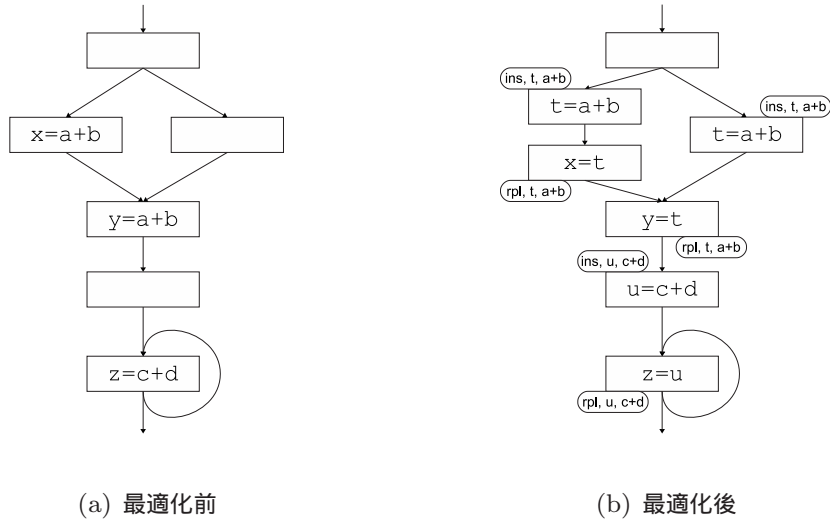


図 5.6 複数箇所を最適化する例

$$\begin{aligned}
 M &\models \text{mark}(\text{ins}, t, a + b) \\
 &\rightarrow A(\text{trans}(a + b) \ W \ \text{mark}(\text{rpl}, t, a + b))
 \end{aligned}
 \tag{5.7}$$

$$\begin{aligned}
 M &\models \text{mark}(\text{rpl}, t, a + b) \\
 &\rightarrow \overleftarrow{A}((\neg \text{def}(t) \wedge \text{trans}(a + b)) \ W \ \text{mark}(\text{ins}, t, a + b))
 \end{aligned}
 \tag{5.8}$$

$$\begin{aligned}
 M &\models \text{mark}(\text{ins}, u, c + d) \\
 &\rightarrow A(\text{trans}(c + d) \ W \ \text{mark}(\text{rpl}, u, c + d))
 \end{aligned}
 \tag{5.9}$$

$$\begin{aligned}
 M &\models \text{mark}(\text{rpl}, u, c + d) \\
 &\rightarrow \overleftarrow{A}((\neg \text{def}(u) \wedge \text{trans}(c + d)) \ W \ \text{mark}(\text{ins}, u, c + d))
 \end{aligned}
 \tag{5.10}$$

## 5.8 ステップ 6 – モデル検査

この時点で、検査対象であるモデルと、仕様から生成した自由変数を含まない CTL 式が得られているので、後はモデル検査により仕様を満たすかどうか検証するだけである\*4。モデル検査の結果、CTL 式を満たさないノードがあった場合、そのノードに施さ

\*4 繰り返しになるが「仕様を満たせば正しい」ことが別途保証されていなくてはならない。

れた変形は誤りであったと判断し，その旨を報告する．

CTL のモデル検査のアルゴリズムはいくつか存在する [7]．よいアルゴリズムは，モデルと式の規模に対して線形であり，実際にもかなり速いといわれている．

本手法では，モデル検査のアルゴリズムの詳細は重要ではない．しかし，CTL の意味のより深い理解のためにはアルゴリズムの説明は有用であると考えられる．この節では，本論文で使用している演算子を持つ CTL 式のモデル検査アルゴリズムの説明を行う．説明するアルゴリズムは，文献 [6] で提案された CTL モデル検査の初期のアルゴリズムを基にしたものであり，本実装でも実際に使用しているものである．

### 5.8.1 アルゴリズムの概要

説明するアルゴリズムは，対象の CTL 式の部分式から順に，その部分式が成り立つ状態の集合を求めていき，最終的に式全体の成り立つ状態の集合を求める，といったものである．

式 (5.5) の CTL 式における部分式の間を明示すると，図 5.7 のようになる．これは CTL の構文木を表す．

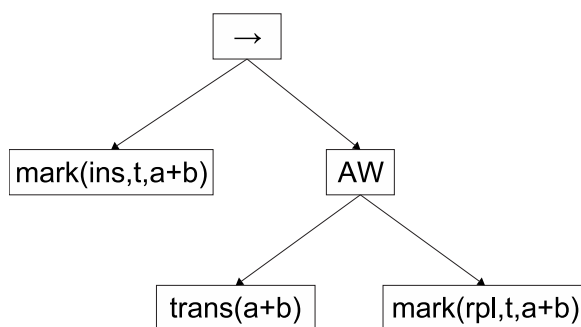


図 5.7 CTL 式の構文木の例

「部分式から順に成り立つ状態の集合を求める」とはつまり，この構文木の葉の要素の式から順に求めていくということである．図 5.7 では，例えば  $AW$  を頂点とする部分式について計算したければ，まず  $trans(a+b)$  と  $mark(rpl,t,a+b)$  の成り立つ状態の集合を先に求めておく必要がある．

## 5.8.2 各演算子ごとのアルゴリズム

以下、各演算子ごとに、CTL 式  $\phi$  を満たす状態の集合を求めるアルゴリズムを示す。モデルを  $M = (N, E, L)$  とする。

ここで、 $\{n \in N \mid n \models \phi\}$  という集合を求めるために、各状態に  $\phi$  を満たすかどうかのラベルを付けることにする。つまり、 $n$  が  $\phi$  を満たすなら「 $n \models \phi$ 」、満たさないなら「 $n \not\models \phi$ 」というラベルを  $n$  に付ける。

### 原始命題

$\phi$  が原始命題のとき、 $\phi$  を満たす状態の集合は明らかで、 $\{n \in N \mid \phi \in L(n)\}$  である。この集合の要素にラベルを付ける。

### 時相演算子でないもの ( $\neg, \wedge$ など)

$\phi = \neg\phi_1$  のとき、 $\phi$  を満たす状態の集合は  $\neg$  の定義から明らかで、 $\{n \in N \mid n \not\models \phi_1\}$  である。ただし、 $\phi_1$  のラベル付けは既に済んでいるものとする。

$\phi = \phi_1 \wedge \phi_2$  のときも同様で、 $\{n \in N \mid n \models \phi_1 \wedge n \models \phi_2\}$  である。ただし、 $\phi_1, \phi_2$  のラベル付けは既に済んでいるものとする。

### EX

$\phi = EX \phi_1$  のとき、状態  $n$  が  $\phi$  を満たすとは、定義より「 $n \rightarrow n'$  かつ  $n' \models \phi_1$  を満たす  $n'$  が存在する」ことであった。これより、 $\phi$  のラベル付けは、アルゴリズム 5.1 となる。ただし、 $\text{succ}(n) = \{n' \mid n \rightarrow n'\}$  とする。

$\phi = \overleftarrow{EX} \phi_1$  についても経路を逆にすることで、同様のアルゴリズムで計算できる。すなわち、アルゴリズム 5.1 の 3 行目の  $\text{succ}(n)$  を  $\text{pred}(n)$  とすればよい。ただし、 $\text{pred}(n) = \{n' \mid n \rightarrow^\circ n'\}$  とする。

### AX

$\phi = AX \phi_1$  のとき、状態  $n$  が  $\phi$  を満たすとは、定義より「 $n \rightarrow n'$  となる  $n'$  が存在し、かつそのような全ての  $n'$  が  $n' \models \phi_1$  を満たす」ことであった。これより、 $\phi$  のラベル付けは、アルゴリズム 5.2 となる。

$\phi = \overleftarrow{AX} \phi_1$  についても経路を逆にすることで、同様のアルゴリズムで計算できる。すなわち、アルゴリズム 5.2 の 2, 6 行目の  $\text{succ}(n)$  を  $\text{pred}(n)$  とすればよい。

---

**アルゴリズム 5.1**  $EX \phi_1$  のラベル付け

---

**labelEX()**

```
1: for all  $n \in N$  do
2:    $val \leftarrow \text{false}$ 
3:   for all  $n' \in succ(n)$  do
4:      $val \leftarrow val \vee n' \models \phi$ 
5:   if  $val = \text{true}$  then
6:     label  $n \models \phi$ 
7:   else
8:     label  $n \not\models \phi$ 
```

---

---

**アルゴリズム 5.2**  $AX \phi$  のラベル付け

---

**labelAX()**

```
1: for all  $n \in N$  do
2:   if  $succ(n)$  is empty then
3:     label  $n \not\models \phi$ 
4:   else
5:      $val \leftarrow \text{true}$ 
6:     for all  $n' \in succ(n)$  do
7:        $val \leftarrow val \wedge n' \models \phi$ 
8:     if  $val = \text{true}$  then
9:       label  $n \models \phi$ 
10:    else
11:      label  $n \not\models \phi$ 
```

---

***EU***

$\phi = E(\phi_1 U \phi_2)$  のとき, 状態  $n$  が  $\phi$  を満たすとは,  $n$  から始まる次のような経路  $p$  が存在する場合であった.

- 経路上には  $\phi_2$  を満たす状態が必ずあり, その状態に至るまでの経路上の全ての状態は  $\phi_1$  を満たす.

つまり、次のようにラベル付けすればよい。

- $\phi_2$  を満たす状態  $n$  には、ラベル  $n \models \phi$  を付ける。
- $\phi_1$  を満たす状態  $n$  が「 $n$  の次の状態  $n'(n \rightarrow n')$  のうちの一つにラベル  $n' \models \phi$  が付いている」という条件を満たすとき、 $n$  にラベル  $n \models \phi$  を付ける。
- 最終的に状態  $n$  にラベルが付いていない場合、ラベル  $n \not\models \phi$  を付ける。

ラベル付けのアルゴリズムは、

1.  $\phi_2$  を満たす状態  $n$  に、ラベル  $n \models \phi$  を付ける。
2.  $\phi_2$  を満たす状態  $n$  に遷移可能な全ての状態  $n'$  に、ラベル  $n' \models \phi$  を付ける。
3. ラベル  $n \models \phi$  の付いている状態  $n$  に遷移可能な全ての状態  $n'$  にもラベル  $n' \models \phi$  を付ける。
4. 3 を繰り返す。

というように、 $\phi_2$  を満たす状態から逆向きに深さ優先で経路を辿ってラベルを付けていく。アルゴリズムの全貌は、アルゴリズム 5.3 である。

$\phi = \overleftarrow{E}(\phi_1 U \phi_2)$  についても、 $EX$  や  $AX$  のときと同じく経路を逆にするだけで、同様のアルゴリズムが適用できる。すなわち、アルゴリズム 5.3 の 9, 19 行目の  $pred(n)$  を  $succ(n)$  とすればよい。

## AU

$\phi = A(\phi_1 U \phi_2)$  のとき、状態  $n$  が  $\phi$  を満たすとは、 $n$  から始まる全ての経路が次の条件を満たす場合であった。

- 経路上には  $\phi_2$  を満たす状態が必ずあり、その状態に至るまでの経路上の全ての状態は  $\phi_1$  を満たす。

つまり、次のようにラベル付けすればよい。

- $\phi_1$  も  $\phi_2$  も満たさない状態  $n$  には、ラベル  $n \not\models \phi$  を付ける。
- $\phi_2$  を満たす状態  $n$  には、ラベル  $n \models \phi$  を付ける。
- $\phi_1$  を満たす状態  $n$  が「 $n$  の次の状態  $n'(n \rightarrow n')$  全てにラベル  $n' \models \phi$  が付いている」という条件を満たすとき、 $n$  にラベル  $n \models \phi$  を付ける。

---

**アルゴリズム 5.3**  $E(\phi_1 \cup \phi_2)$  のラベル付け

---

**labelEU()**

```
1: for all  $n \in N$  do
2:   unlabel  $n$ 
3: for all  $n \in N$  do
4:   if  $n$  is not labeled then
5:     if  $n \not\models \phi_1 \wedge n \not\models \phi_2$  then
6:       label  $n \not\models \phi$ 
7:     else if  $n \models \phi_2$  then
8:       label  $n \models \phi$ 
9:     for all  $n' \in \text{pred}(n)$  do
10:       $\text{subEU}(n')$ 
11: for all  $n \in N$  do
12:   if  $n$  is not labeled then
13:     label  $n \not\models \phi$ 
```

**subEU( $n$ )**

```
14: if  $n$  is not labeled then
15:   if  $n \not\models \phi_1 \wedge n \not\models \phi_2$  then
16:     label  $n \not\models \phi$ 
17:   else
18:     label  $n \models \phi$ 
19:     for all  $n' \in \text{pred}(n)$  do
20:       $\text{subEU}(n')$ 
```

---

上記のラベル付けは、経路を深さ優先探索で辿っていくことで行う。一度訪れた状態にはマーク\*<sup>5</sup>を付けておく。すでにマークを付けた状態を再び訪れた場合、その状態にラベルが付いていなければ、その状態を含む経路には、 $\phi_1$  が成立し続けるが  $\phi_2$  が成立することはないものが存在することになり、そのような状態  $n$  には、 $A(\phi_1 \cup \phi_2)$  の定義よりラベル  $n \not\models \phi$  が付くことになる。アルゴリズムの全貌は、アルゴリズム 5.4 である。

---

\*<sup>5</sup> 本手法における変形箇所につけるマークとは当然別の意味である。

---

**アルゴリズム 5.4**  $A(\phi_1 \cup \phi_2)$  のラベル付け

---

**labelAU()**

- 1: **for all**  $n \in N$  **do**
- 2:   unlabel  $n$
- 3:   unmark  $n$
- 4: **for all**  $n \in N$  **do**
- 5:    $subAU(n)$

**subAU( $n$ )**

- 6: **if**  $n$  is not marked **then**
  - 7:   mark  $n$
  - 8:   **if**  $n \not\models \phi_1 \wedge n \not\models \phi_2$  **then**
  - 9:     label  $n \not\models \phi$ ; **return false**
  - 10:   **else if**  $n \models \phi_2$  **then**
  - 11:     label  $n \models \phi$ ; **return true**
  - 12:   **else if**  $succ(n)$  is empty **then**
  - 13:     label  $n \not\models \phi$ ; **return false**
  - 14:   **else**
  - 15:     **for all**  $n' \in succ(n)$  **do**
  - 16:       **if**  $\neg subAU(n')$  **then**
  - 17:         label  $n \not\models \phi$ ; **return false**
  - 18:       label  $n \models \phi$ ; **return true**
  - 19:   **else**
  - 20:     **if**  $n \not\models \phi \vee n$  is not labeled **then**
  - 21:       **return false**
  - 22:     **else** //  $n \models \phi$
  - 23:       **return true**
-

$\phi = \overleftarrow{A}(\phi_1 U \phi_2)$  についても,  $EU$  のときと同じく経路を逆にすることで, 同様のアルゴリズムが適用できる. すなわち, アルゴリズム 5.4 の 12, 15 行目の  $\text{succ}(n)$  を  $\text{pred}(n)$  とすればよい.

## AW

$\phi = A(\phi_1 W \phi_2)$  のとき, 状態  $n$  が  $\phi$  を満たすとは,  $n$  から始まる全ての経路が次のいずれかの条件を満たす場合であった.

- 経路上には  $\phi_2$  を満たす状態が必ずあり, その状態に至るまでの経路上の全ての状態は  $\phi_1$  を満たす.
- 経路は無限経路で, 経路の全ての状態が  $\phi_1$  を満たす.

つまり, 次のようにラベル付けすればよい.

- $\phi_2$  を満たす状態  $n$  には, ラベル  $n \models \phi$  を付ける.
- $\phi_2$  を満たさない状態  $n$  が,  $\phi_1$  を満たさないか次の状態がない場合には,  $n$  にラベル  $n \not\models \phi$  を付ける.
- $\phi_2$  を満たさない状態  $n$  が「 $n$  の次の状態  $n'(n \rightarrow n')$  のうちの一つにラベル  $n' \not\models \phi$  が付いている」という条件を満たすとき,  $n$  にラベル  $n \not\models \phi$  を付ける.
- 最終的に状態  $n$  にラベルが付いていない場合, ラベル  $n \models \phi$  を付ける.

ラベル付けアルゴリズムの全貌は, アルゴリズム 5.5 である. このラベル付けは  $EU$  のラベル付けに似ているが,  $EU$  のラベル付けが, ラベル  $n \models \phi$  の付いた状態  $n$  に遷移可能な全ての状態  $n'$  にラベル  $n' \models \phi$  を付けるアルゴリズムであるのに対し,  $AW$  のラベル付けは, ラベル  $n \not\models \phi$  の付いた状態  $n$  に遷移可能な全ての状態  $n'$  にラベル  $n' \not\models \phi$  を付けるアルゴリズムであり, 逆の関係といえる.

$\phi = \overleftarrow{A}(\phi_1 W \phi_2)$  についても,  $EU$  のときと同じく経路を逆にすることで, 同様のアルゴリズムが適用できる. すなわち, アルゴリズム 5.5 の 8 行目の  $\text{succ}(n)$  を  $\text{pred}(n)$  に, 11, 22 行目の  $\text{pred}(n)$  を  $\text{succ}(n)$  とすればよい.

---

**アルゴリズム 5.5**  $A(\phi_1 \ W \ \phi_2)$  のラベル付け

---

**labelAW()**

```
1: for all  $n \in N$  do
2:   unlabel  $n$ 
3: for all  $n \in N$  do
4:   if  $n$  is not marked then
5:     if  $n \models \phi_2$  then
6:       label  $n \models \phi$ 
7:       mark  $n$ 
8:     else if  $n \not\models \phi_1 \vee \text{succ}(n)$  is empty then
9:       label  $n \not\models \phi$ 
10:      mark  $n$ 
11:     for all  $n' \in \text{pred}(n)$  do
12:       subAW( $n'$ )
13: for all  $n \in N$  do
14:   if  $n$  is not marked then
15:     label  $n \models \phi$ 
```

**subAW( $n$ )**

```
16: if not marked  $n$  then
17:   mark  $n$ 
18:   if  $n \models \phi_2$  then
19:     label  $n \models \phi$ 
20:   else
21:     label  $n \not\models \phi$ 
22:     for all  $n' \in \text{pred}(n)$  do
23:       subAW( $n'$ )
```

---

## 第Ⅳ部

# 実装

## 第 6 章

# コンパイラインフラストラクチャ COINS と本手法との関係

本章では，本手法の適用を行う際に利用した並列化コンパイラ向け共通インフラストラクチャ（COmpiler INfra Structure, COINS）について説明を行う．

### 6.1 概要

COINS は，コンパイラ研究の基盤となる共通のコンパイラインフラストラクチャの作成をテーマに 2000 年度より研究が進められているプロジェクトである [8]．

COINS には高水準と低水準の二つの中間表現があり，低水準中間表現 LIR では，多くの SSA 最適化を行っている．現在，多くの最適化コンパイラで SSA 最適化の実験や実装が行われているが [13, 14, 15, 30]，本研究室の佐々政孝教授が COINS の研究プロジェクトに携わっているため，本研究室では COINS をインフラとして利用した研究が多く行われており，データの蓄積が行いやすいため，本手法の実装を行うインフラに COINS を選択した．

### 6.2 構成

図 6.1 に COINS の構成図を示す．COINS は，フロントエンドとバックエンドから構成されている．フロントエンドではまず，ソースプログラムを読み込んで構文・意味解析を行い，高水準中間表現 HIR に変換する．HIR 上ではいくつかの最適化を行うことができる．HIR は最終的には低水準中間表現 LIR に変換される．バックエンドでは LIR を扱

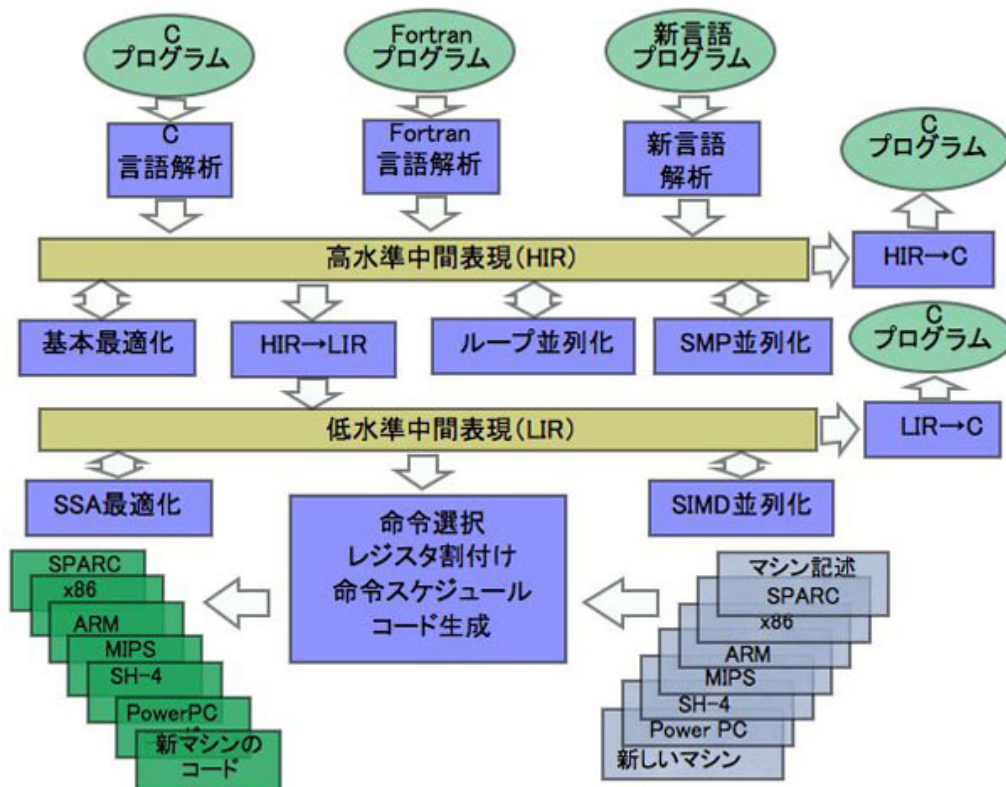


図 6.1 COINS の構成図

う．バックエンドには SSA 最適化をはじめとする多くの最適化が実装されており，これらを行うことができる．LIR は最終的に，命令選択やレジスタ割付けを経て，対象機械の目的コードに変換される

### 6.3 COINS バックエンドの構造

本研究では，提案手法の適用実装を LIR 上での最適化（SSA 最適化を含む）を対象に行った．この節では，LIR の構成について説明する [22]．

COINS では，一つのソースファイルをコンパイルの一単位としている．LIR ではコンパイル単位を Module と呼ぶ．Module の構成を図 6.2 に示す．Module は，大域変数の記号表である `globalSymTab` と，関数を表す `Function` のリストから構成されている．LIR の関数は `L 関数` と呼ばれる．



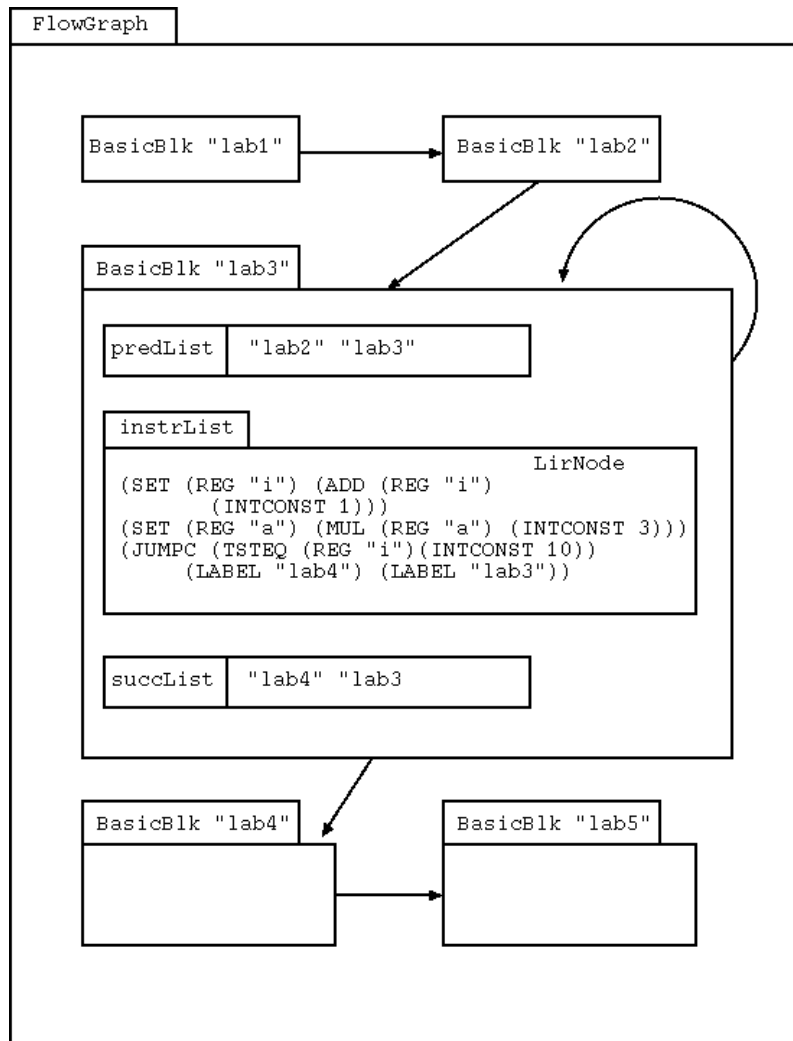


図 6.3 FlowGraph の構成

データ構造が保持している .instrList は、この BiLink のリストである。

LIR の文は Lisp などでも用いられている S 式の形をしている。この LIR の式は L 式と呼ばれる。図 6.4 の LirNode が L 式を表している。

## 6.4 LIR におけるモデルの定義

COINS バックエンドの最適化の多くは L 関数単位で行っており、特に SSA 最適化の対象は全て L 関数である。よって、本実装の適用対象も L 関数とし、L 関数の制御フローグラフから制御フローモデルを生成した。

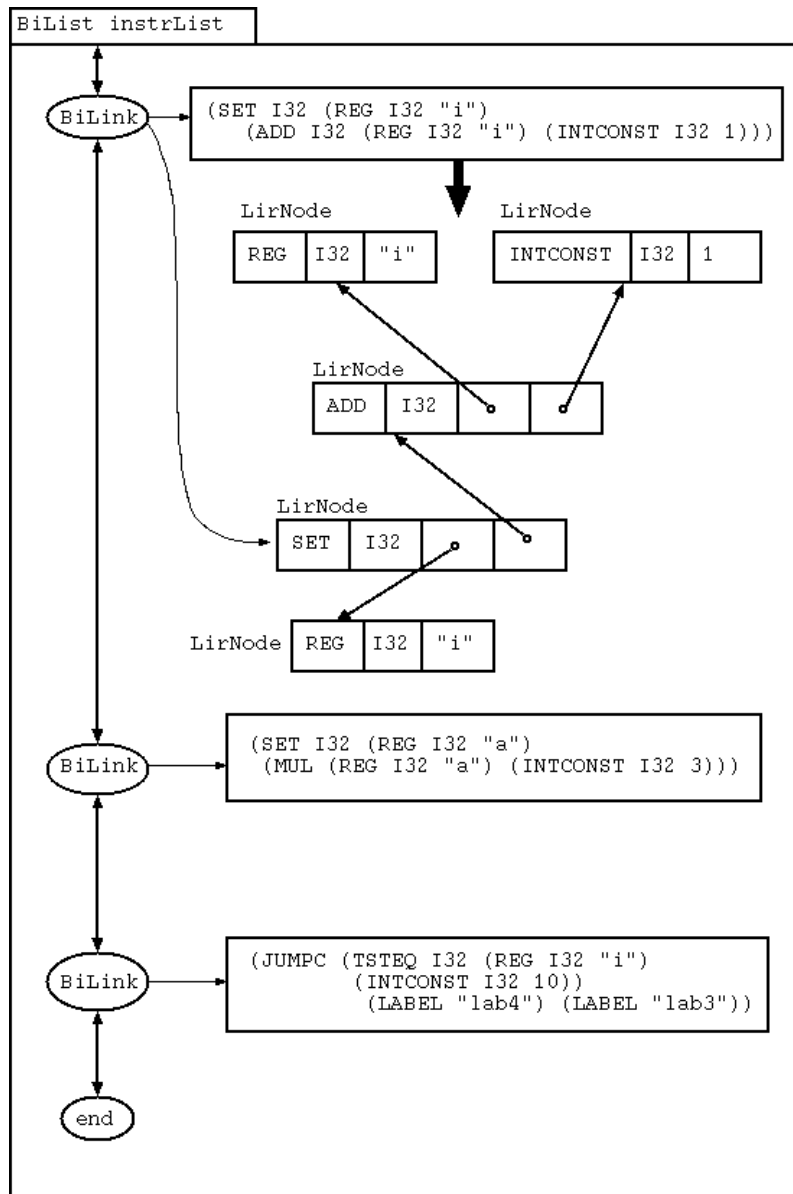


図 6.4 `instrList` の構成

この節では、`L` 関数上で制御フローモデルを定義する<sup>\*1</sup>。以下、制御フローモデルを  $M = (N, E, L)$  とする。

<sup>\*1</sup> LIR の正確な言語仕様は、文献 [9] で定義されているので、ここでは直観的な意味から定義することにする。

### 6.4.1 モデルの状態

状態の集合  $N$  の要素は、LIR の文とする。すなわち、図 6.4 の BiLink 一つが一つの状態に対応することになる。

基本ブロックを状態することも考えられるが、その場合、4.1.2 節で挙げた原始命題では十分ではなく、例えば「基本ブロックで、変数が再定義前に使用されるならば真」といったものが必要となってくる。これは、基本ブロック内だけとはいえ、データフロー解析を行っていることに相当する。しかし、データフロー解析に誤りが混入しやすいために本手法のような別の角度からの検証を行うのであって、基本ブロックでのデータフロー解析を行うのでは本末転倒になってしまう。よって、状態の単位は文とするべきであると考えた。

### 6.4.2 状態間の遷移関係

文献 [9] によると、状態間の遷移関係  $n_1 \rightarrow n_2$  があるのは、次のいずれかの条件を満たすときである。

- $n_1$  が Jump 文で、かつジャンプ先の基本ブロックの先頭の文が  $n_2$  である場合。
- $n_1$  が Jump 文でなく、かつ  $n_2$  が  $n_1$  の字面上の次の文である場合。

ただし Jump 文とは、JUMP 式、JUMPC 式、または JUMPN 式のいずれかを頂点とする L 式とする。

実装する上では、COINS の L 関数中の文は 6.3 節で述べたように制御フローグラフの形で保持されているので、遷移関係はそれを直接用いればよい。

### 6.4.3 原始命題の意味

この節では、4.1.2 節で挙げた原始命題について、LIR での意味の説明を行う。ただし、 $node(N)$ 、 $block(B)$ 、および  $mark(M)$  については表 4.1 の定義で明らかであるので、ここではその他の原始命題について述べる。

#### 変数と式

各原始命題について述べる前に、変数、式とは何かを定義しておく必要がある。

変数とは、ここでは最適化の対象となる記号のことをいう。LIR 上での多くの最適化

が、仮想レジスタの REG 式を最適化対象の記号としているので\*<sup>2</sup>、本実装でもそれに合わせて、変数とは仮想レジスタの REG 式のこととした\*<sup>3</sup>。

式とは、ここでは算術式や論理演算式など、レジスタ・メモリへの書き込みやジャンプなどではないものをいう。LIR では、このような式は PURE 式と呼ばれている。本実装では、式とはこの PURE 式のこととした。また、REG 式単体や INTCONST 式、FLOATCONST 式も式とした。

## 各原始命題

### 定義 6.1 (使用される変数)

$use(x) \in L(n)$  ならば、 $n$  の文は次のいずれかを満たす。

- SET 式であり、第二引数に  $x$  が出現する。
- SET 式かつ第一引数が MEM 式であり、その MEM 式中に  $x$  が出現する。
- PHI 式であり、第二引数以降に  $x$  が出現する。
- CALL 式であり、第一、第二引数のいずれかに  $x$  が出現する。
- EPILOGUE 式であり、第二引数以降に  $x$  が出現する。
- 上記の L 式、PROLOGUE 式、CLOBBER 式、INFO 式ではなく、かつ  $x$  が出現する。

この場合に、 $n$  で  $x$  が使用されるということにする。

### 定義 6.2 (定義される変数)

$def(x) \in L(n)$  ならば、 $n$  の文は次のいずれかを満たす。

- SET 式かつ第一引数が MEM 式でなく、第一引数が  $x$  である。
- PHI 式であり、第一引数が  $x$  である。
- CALL 式であり、第三引数に  $x$  が出現する。
- PROLOGUE 式であり、第二引数以降に  $x$  が出現する。
- CLOBBER 式であり、 $x$  が出現する。

この場合に、 $n$  で  $x$  が定義されるということにする。

---

\*<sup>2</sup> LIR には素朴な別名解析が実装されており、別名解析を行った場合はメモリの一部も最適化対象とするものがある [29]。その場合はメモリ全体も一つの変数として扱うことにする。

\*<sup>3</sup> LIR には部分レジスタを表す SUBREG 式というものもあるが、SUBREG 式の現れる L 関数は最適化対象としていないので、ここでは考慮する必要はない。

**定義 6.3 (式の計算)**

$comp(e) \in L(n)$  ならば,  $n$  の文には  $e$  が出現する.  $e$  は PURE 式, REG 式, INTCONST 式, FLOATCONST 式のいずれかである. ただし,  $e$  が REG 式  $x$  の場合,  $use(x) \in L(n)$  の場合に限り  $comp(e) \in L(n)$  とする. また,  $e$  が MEM 式  $m$  の場合,  $m$  が SET 式の第一引数でない場合に限り  $comp(e) \in L(n)$  とする.

$comp(e) \in L(n)$  の場合に,  $n$  で  $e$  が計算されるということにする.

**定義 6.4 (式の変更)**

$trans(e) \in L(n)$  ならば,  $n$  の文では  $e$  で使用されている変数を定義しない. この場合に,  $n$  で  $e$  が変更されないということにする.

## 第 7 章

# 提案手法の適用

本研究では，次の最適化について本手法による検査を実現した．

- 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<sup>\*1</sup>
- Lazy Code Motion

以下，それらについて述べる．

### 7.1 ループ不変式移動への適用

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．

図 7.1 は SSA 形式上でのループ不変式移動の例である．図 7.1(a) で， $a + b$  はループ内で値の変わらない式であるので，図 7.1(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  $a + b$  の計算は，一時変数  $t$  に置き換えられる．

---

<sup>\*1</sup> 検査の一部は実装が完了していない．詳細は付録 A.3 で説明する．

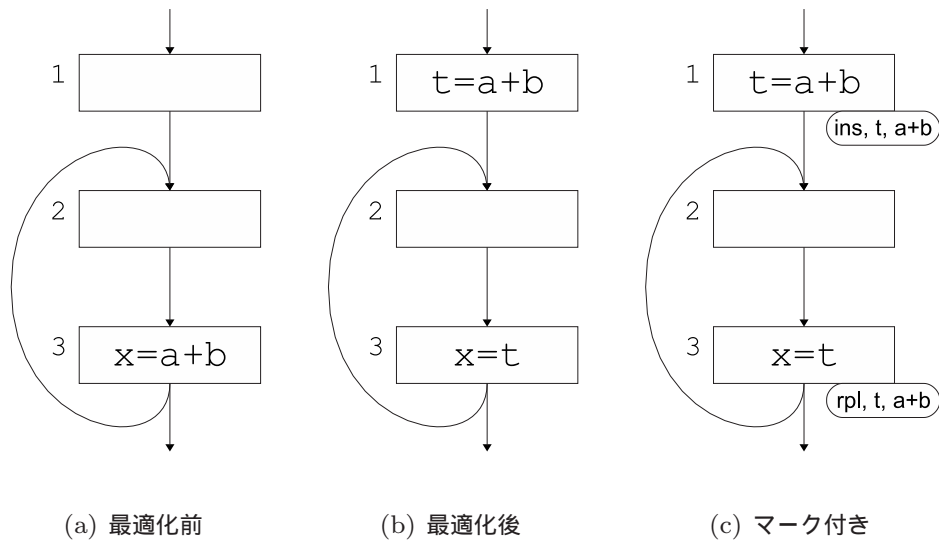


図 7.1 SSA 形式上でのループ不変式移動の例

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。

### 7.1.1 ループ不変式移動の正しさ

図 7.1 のループ不変式移動の例では、最適化による変形は次の二箇所であった。

- $a + b$  を巻き上げて、ループの前に  $t = a + b$  を挿入した。
- $a + b$  の元の計算を一時変数  $t$  で置き換えた。

それぞれの変形箇所には、図 7.1(c) のようにマーク  $(ins, t, a + b)$  ,  $(rpl, t, a + b)$  を最適化中の変形と同時に付ける。

一般的には、ループ不変式移動は「巻き上げた式の挿入」と「式の使用の置換」の二つの変形からなるといえる。以下、それぞれの変形箇所を満たすべき仕様を説明する。

#### 巻き上げた式を挿入した箇所

図 7.1 の例において、 $a + b$  の計算は例外を発生するなどの副作用を伴わない。よって、 $a$  か  $b$  の定義文を越えなければ、いくら巻き上げて計算してもプログラムの意味は変わらない<sup>\*2</sup>。SSA 形式では変数の二重定義が許されないので、 $t$  が  $(ins, t, a + b)$  の付いた

<sup>\*2</sup> 挿入する場所によっては計算する回数が増えて効率が落ちるかもしれないが、プログラムの意味自体は変わらない。

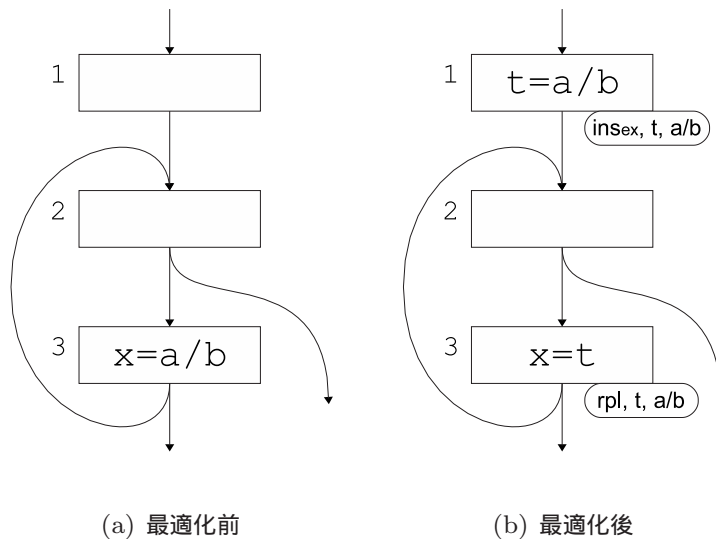


図 7.2 ループ不変式移動の誤った最適化の例

ノード 1 以外で定義されていないならば， $t = a + b$  の挿入はプログラムの意味を変えないといえる．

以上から，一時変数  $t$  を  $t$ ，式  $a + b$  を  $e$  という自由変数で表し， $(ins, t, e)$  のマークの付いているノードを自由変数  $n$  で表すと，このマーク  $(ins, t, e)$  の付いたノードの満たすべき仕様は，

$$\begin{aligned}
 \text{MARK} & \quad (ins, t, e) \\
 \text{FORMULA} & \quad \neg EF (\neg node(n) \wedge def(t)) \wedge \overleftarrow{EF} (\neg node(n) \wedge def(t)) \quad (7.1)
 \end{aligned}$$

となる．この仕様の直観的な意味は「 $n$  以外に  $t$  を定義するノードがある経路は存在しない」である．

式 (7.1) は，副作用を伴わない式の巻き上げ先が満たすべき式であった．しかし，0 による除算の可能性があるので，副作用を伴う可能性のある式ではこれでは十分でない．

例えば，図 7.2(a) の  $a/b$  は，ループ不変式であるので一見巻き上げ可能に見える．しかし，ノード 3 を通らずにループを出る経路が存在するため，例えば「 $b$  の値は 0 であるが，ノード 3 は通らず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 7.2(b) のように  $a/b$  を巻き上げてしまうと， $a/b$  は必ず実行されることになり，元々起こらなかった例外が起こるようになってしまう．これは正しくない．

最適化が正しくなかった理由は，元々計算のなかった経路に例外を発生しうる計算を巻

き上げて挿入してしまったのが原因である。文献 [1] や [3] によると、例外を発生しうる式は、元々計算のなかった経路に巻き上げることがないよう気を付ける必要がある、とある。従って、例外を発生しうる式の巻き上げ先の満たすべき仕様は式 (7.1) では十分でない。例外を発生する式を巻き上げて挿入した箇所に特別に  $(ins_{ex}, t, e)$  というマークを付けることにすると、仕様は、

$$\begin{array}{ll} \text{MARK} & (ins_{ex}, t, e) \\ \text{FORMULA} & A(trans(e) W mark(rpl, t, e)) \wedge \phi \end{array} \quad (7.2)$$

となる。ただし、式 (7.1) の FORMULA を  $\phi$  としている。この式の直感的な意味は「全ての経路で、 $e$  の値が変更されないまま元々計算のあったノードに必ず到達する」である。なお、 $U$  演算子ではなく  $W$  演算子を使っている理由は、終端に至らない経路<sup>\*3</sup>を考慮しないためである。最適化では通常、終端に至らない経路は考慮しない。

式の計算を置換した箇所

図 7.1 の例で、 $a + b$  の計算を  $t$  で置換した箇所では、すでに  $a + b$  の値が  $t$  によって利用可能となっている必要がある。言い換えると、全ての逆向きのパス上で  $t = a + b$  を挿入した点に式  $a + b$  の値が変わることなく到達する必要がある<sup>\*4</sup>。これは、 $a/b$  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同様である。

以上から、一時変数  $t$  を  $t$ 、式  $a + b$  を  $e$  という自由変数で表すと、このマーク  $(rpl, t, e)$  の付いたノードの満たすべき仕様は、

$$\begin{array}{ll} \text{MARK} & (rpl, t, e) \\ \text{FORMULA} & \overleftarrow{A}(trans(e) W (mark(ins, t, e) \vee mark(ins_{ex}, t, e))) \end{array} \quad (7.3)$$

となる。

### 7.1.2 効率向上のための工夫

ループ不変式移動の仕様は、式 (7.1)、式 (7.2)、式 (7.3) と記述することができたので、モデル検査のために 5.7 節で述べた方法により仕様を具体化し、CTL 式を生成する必要がある。図 7.1(c) のようにマークを付けたとすると、仕様の具体化を試みると次のよう

<sup>\*3</sup> ずっとループを繰り返す経路など。

<sup>\*4</sup> ここでも終端に至らない経路は考慮しない。理由は同様である。

になる\*5 .

$$\begin{aligned} M \models \text{mark}(\text{ins}, t, a + b) \\ \rightarrow \neg EF (\neg \text{node}(n) \wedge \text{def}(t)) \wedge \overleftarrow{EF} (\neg \text{node}(n) \wedge \text{def}(t)) \end{aligned} \quad (7.4)$$

$$\begin{aligned} M \models \text{mark}(\text{rpl}, t, a + b) \\ \rightarrow \overleftarrow{A} (\text{trans}(a + b) W \text{mark}(\text{ins}, t, a + b)) \end{aligned} \quad (7.5)$$

式 (7.5) に関しては完全に具体化された CTL 式なのでそのままモデル検査を行えるが、式 (7.4) に関しては、まだ  $n$  を束縛していないので、このままではモデル検査できない。よって、 $n$  を束縛する必要がある。

最も単純には、モデルの全てのノードを順に選び、その番号で束縛すればよい。しかし、この方法ではモデルの状態数の数だけ CTL 式が生成され、検査の回数が非常に多くなってしまい、効率が悪い。

そこで、次のような工夫を行うことで効率の改善を図る。

- 巻き上げ先に付けるマークを  $(\text{ins}, t, e)$  から  $(\text{ins}, t, e, n)$  に変更する。ただし、 $n$  は巻き上げ先のノードの番号とする。図 7.1(c) の例では、実際に付けるマークは  $(\text{ins}, t, a + b, 1)$  となる。
- マークの変更に合わせて仕様も変更する。つまり、式 (7.1) と式 (7.3) の中に現れる  $(\text{ins}, t, e)$  を  $(\text{ins}, t, e, n)$  で置き換える。 $(\text{ins}_{\text{ex}}, t, e)$  についても同様に置き換える。

こうして変更した仕様に、式 (7.4) と式 (7.5) を対応させると次のようになる。

$$\begin{aligned} M \models \text{mark}(\text{ins}, t, a + b, 1) \\ \rightarrow \neg EF (\neg \text{node}(1) \wedge \text{def}(t)) \wedge \overleftarrow{EF} (\neg \text{node}(1) \wedge \text{def}(t)) \end{aligned} \quad (7.6)$$

$$\begin{aligned} M \models \text{mark}(\text{rpl}, t, a + b) \\ \rightarrow \overleftarrow{A} (\text{trans}(a + b) W \text{mark}(\text{ins}, t, a + b, n)) \end{aligned} \quad (7.7)$$

これで、式 (7.6) の自由変数をなくし、具体化することができたが、今度は式 (7.7) 中に束縛しきれない  $n$  が出現してしまっている。しかし、式 (7.3) の仕様の意味を考えると、

---

\*5 簡単のため、マーク  $(\text{ins}_{\text{ex}}, t, e)$  に関する部分は除いてある

置換点から見て挿入点が存在することが重要なのであって、どのノードに挿入したかは重要でない。つまり、 $n$  はどんな値でもよい。

そこで、任意の値を表す記号  $*$  を導入し、式 (7.7) の  $n$  を  $*$  で書き換える\*<sup>6</sup>。ここで、 $mark(ins, t, a + b, *)$  は、マーク  $(ins, t, a + b, 1)$  のついたノードでも  $(ins, t, a + b, 2)$  のついたノードでも真になる原始命題と解釈することにする。この拡張により、 $n$  を束縛する必要がなくなり、効率よい検査を行うことができる。

## 7.2 条件分岐を考慮した定数伝播への適用

COINS には、条件分岐を考慮した定数伝播 [34] 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。Lacey らの手法などでは、記述能力の限界からこ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も検査することができる。

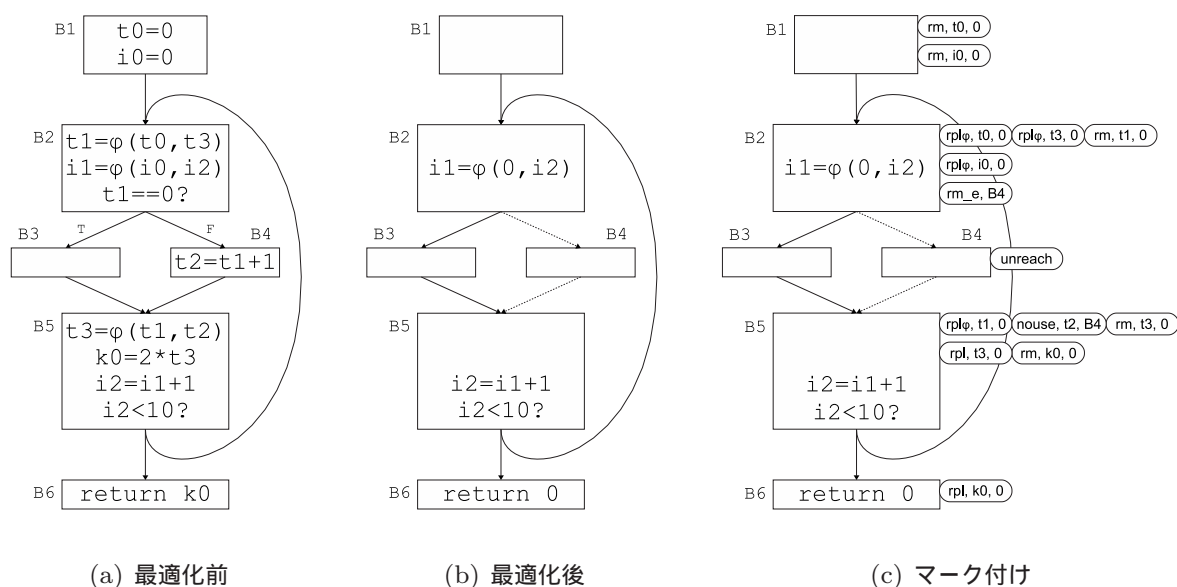


図 7.3 条件分岐を考慮した定数伝播の例

図 7.3 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 7.3(a) のグラフを入り口から順に辿ると、ブロック B4 は到達不能であり、 $t1$  は常に 0 とな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

\*<sup>6</sup> いわゆるワイルドカード文字に相当する。論理的には  $\exists$  限量子に相当するともいえる。

いう枠組みの中で行われる最適化である。

最適化後は図 7.3(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。

### 7.2.1 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 (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 (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 (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.  $\phi$  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 (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  $\phi$  関数かどうかにより異なる。

- $e$  が  $\phi$  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  $\phi$  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<sup>\*7</sup>、その式の計算結果が  $c$  となる場合、 $e$  は  $c$  と評価される。

---

<sup>\*7</sup>  $0 \times x$  のような場合の  $x$  は、この限りではない。

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  $(rpl, x, c)$  が付いており, また,  $\phi$  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  $(nouse, x, p)$  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,  $\perp$  は値が不定であることを表す記号とする [34]. また,  $c, c_1, c_2$  は定数を,  $x, y$  など変数を,  $op$  は算術演算を表す.

$$\begin{aligned}
eval(\top) &= \top \\
eval(\perp) &= \perp \\
eval(c) &= c \\
eval(x) &= \begin{cases} \top & \text{マーク (nouse, } x) \text{ が付いている} \\ c & \text{マーク (rpl, } x, c) \text{ が付いている} \\ \perp & \text{otherwise} \end{cases} \\
eval(x \text{ op } y) &= \begin{cases} c & eval(x) = c_1 \wedge eval(y) = c_2 \wedge c_1 \text{ op } c_2 = c \\ \perp & \text{otherwise} \end{cases} \\
eval(\phi(x_1, x_2, \dots)) &= \begin{cases} c & \forall x_i, eval(x_i) = c \vee eval(x_i) = \top \\ \perp & \text{otherwise} \end{cases}
\end{aligned}$$

$eval(e) = c$  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  $(rm, x, c)$  が付けられる.

以上から仕様は,

$$\begin{aligned}
\text{MARK} \quad & (rm, x, c) \\
\text{FORMULA} \quad & eval(e) = c
\end{aligned} \tag{7.8}$$

となる. FORMULA の式は厳密には CTL-FV 式ではないが, この仕様は, 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るので, モデル検査による検証を必要としない. よって, 5.6.1 節で説明したマーク付けの際の検査として実装してしまうことにする.

#### 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  $(rm, x, c)$  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  $m$  が到達不能ブロックが存在すればよいので, 仕様は,

$$\begin{array}{ll} \text{MARK} & (\text{rpl}, x, c) \\ \text{FORMULA} & \overleftarrow{A} (\text{true } W (\text{mark}(\text{rm}, x, c) \vee \text{mark}(\text{unreach}))) \end{array} \quad (7.9)$$

となる .

$\phi$  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 7.3 では , ブロック B2 の  $t1 = \phi(t0, t3)$  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  $\phi$  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  $\phi$  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 (7.9) 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 .

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

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 ,  $\mu$  計算と同等の計算力があるといわれている [31] .  $\mu$  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 7.3(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 7.2.1 節で示したように記述可能である .

## 7.3 他の最適化器への適用

本研究では，SSA 形式上でのループ不変式移動や条件分岐を考慮した定数伝播を含め，5.2 節で取り上げた Lazy Code Motion を含め，様々な COINS の最適化器について本手法を適用した実装を行った．それをまとめると以下の通りである．

SSA 最適化 [29]

- 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\*<sup>8</sup>

通常形式上の最適化

- Lazy Code Motion[16, 17] \*<sup>9</sup>

## 7.4 アスペクト指向プログラミングによる最適化器の拡張

5.4 節や 5.5 節で述べたように，本手法では，変形箇所のマーク付けや，モデルの生成・変形などの目的で，最適化器自体の拡張を行う必要がある．これは，アスペクト指向プログラミング (AOP) を利用することで，最適化器のコードそのものをほとんど改変することなく行える．

本手法で必要とする主な拡張箇所をまとめると，次の 3 種類となる．

- 最適化の実行直前の箇所に，モデル生成のためのコードを挿入する．
- 変形を行うコードの箇所に，マーク付けのためのコードを挿入する．
- 変形を行うコードの箇所に，モデル変形のためのコードを挿入する．

それぞれ，5.4 節と 5.5 節で簡単に述べているが，ここで例を挙げて詳細を述べる．

---

\*<sup>8</sup> 仕様の大まかな記述までは行ったが完全な実装には至っていない．

\*<sup>9</sup> これは COINS 標準付属モジュールではなく，自分で実装したものである．

### 7.4.1 アスペクト指向プログラミングシステム GluonJ

本実装では、Java のアスペクト指向プログラミングシステム GluonJ[5] を用いて最適化器の拡張を行った。

GluonJ は、Java 5.0 の文法でアスペクト指向プログラミングを提供しているという特徴がある。GluonJ では、AOP の部品を `Glue` クラスという特殊なクラス内にまとめて定義する。

GluonJ による拡張には、次の 2 種類がある。

- クラスの改良
- ポイントカットとアドバイスによるコードの挿入・上書き。

クラスの改良は、文字通り Java のクラスの改良を行う機能であり、メソッドの書き換えやクラスのメンバの追加などが行える。コードの挿入・上書きは、ポイントカットにより挿入・上書き箇所であるジョインポイントを指定し、そのジョインポイントにアドバイスのコード（アドバイス・ボディ）を挿入・上書きすることができる。

本実装では、この GluonJ のポイントカットとアドバイスによるコード挿入を用いて、上記のマーク付けのためのコードの挿入や、モデルの生成・変形のためのコードの挿入を行った。

### 7.4.2 ループ不変式移動を行う最適化器への適用例

図 7.4 は、ループ不変式移動を行う最適化器の Java 風の擬似コードである。doIt() が最適化を行うメソッド本体であり、insert()、replace() メソッドがそれぞれ命令の挿入と式の置換を行うメソッドである。

この最適化器を拡張するための Glue クラスは、図 7.5 のようになる<sup>\*10</sup>。

簡単に説明すると、@Before や @After といったアノテーションの引数がアドバイス・ボディであり、最適化器のコードに挿入されるコードである。Pointcut クラスのフィールドに代入されているものがポイントカットであり、これにより挿入箇所を指定する。例えば、9-10 行目の記述は、HLI クラスの insert() メソッドが呼ばれる箇所をジョインポイントとして、その直後に HLI`Glue.mark_ins()` というメソッド呼び出しのコードを挿入することを表している。詳細は GluonJ の Web サイト [5] を参照されたい。

---

\*10 詳細は省略している。

```

1: class HLI {
2:   void doIt(Function f) {
3:     Set<Loop> loops = getLoops(f); // ループ構造の収集
4:     for (Loop l : loops) {
5:       Set<LirNode> exps = getExps(f); // ループ不変式の収集
6:       for (LirNode e : exps) {
7:         if (isHoistable(e)) { // 巻き上げ可能か?
8:           LirNode t = makeNewTemporary();
9:           insert(t, e, l.phead); // ループのプリヘッドに挿入
10:            replace(t, e);          // 式を置換
11:        }
12:      }
13:    }
14:  }
15: }

```

図 7.4 ループ不変式移動を行う最適化器の擬似コード

### 7.4.3 ポイントカット指定が困難な場合

最適化器の実装によっては、変形を行うコードの箇所の指定が困難なものも存在する。そのような箇所をポイントカットで指定するために、本実装では最適化器の該当箇所に、空のメソッドの呼び出しを手で直接挿入し、そのメソッドをジョインポイントとするようなポイントカットを記述するように実装した。

図 7.6 は、空のメソッドの呼び出しを挿入した例である。変形を行う `someTransform()` メソッドの直後に、中身が空の `dummyMethod()` メソッドの呼び出しを挿入し、それに対するポイントカット指定をしている。

```

1: @Glue class HLI glue {
2:   @Before("HLI glue.makeModel($1)")
3:   Pointcut point_make = Pcd.call("HLI#doIt(..");
4:
5:   static void makeModel(Function f) {
6:     // モデルを生成するメソッド
7:   }
8:
9:   @After("HLI glue.mark_ins(...);")
10:  Pointcut point_ins = Pcd.call("HLI#insert(..");
11:
12:  static void mark_ins(...) {
13:    // マーク (ins,t,e) を付け ,
14:    // モデルの変形をするメソッド
15:  }
16:
17:  @After("HLI glue.mark_rpl(...);")
18:  Pointcut point_rpl = Pcd.call("HLI#replace(..");
19:
20:  static void mark_rpl(...) {
21:    // マーク (rpl,t,e) を付け ,
22:    // モデルの変形をするメソッド
23:  }
24: }

```

図 7.5 ループ不変式移動を行う最適化器の擬似コード

```

class Opt {
    void doIt(Function f) {
        ...
        someTransform();
        ...
    }
}

```

(a) 拡張前の最適化器

```

class Opt {
    void doIt(Function f) {
        ...
        someTransform();
        OptGlue.dummyMethod();
        ...
    }
}

@Glue class OptGlue {
    @Before("someAdviceBody();")
    Pointcut point_make
        = Pcd.call("Opt#doIt(..)");

    static void dummyMethod() {}
}

```

(b) 拡張後の最適化器

図 7.6 空のメソッドの挿入の例

## 第 8 章

# 実験と考察

### 8.1 未知のバグの発見

本研究では，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．

COINS の SSA 形式上でのループ不変式移動は，COINS SSA 最適化仕様書 [29] によると，文献 [3] のアルゴリズムを採用している．この文献によると，7.1.1 節でも述べたように，例外が発生する可能性のある式を巻き上げる際は，元々計算のなかった経路に挿入しないように注意する必要がある，とある．

ところが，COINS の実装ではこのことが考慮されていないので，まさに図 7.2 のような誤った最適化が行われ，実行時例外が発生する可能性がある．これは未知のバグであったが，本手法を用いて仕様に基づいた検査式を定式化し検査を行ったところ，このバグを発見することができた．

検査に用いたコンパイル対象のプログラムは，SPEC CPU2000[32] のベンチマークプログラムの一つである 254.gap であった．SSA 最適化を有効にした状態で，このベンチマークのコンパイル・実行を行っても，0 による除算の実行時例外は発生せず，ベンチマークは正常に終了していたため，今までこのバグは発見されなかった．このバグは発見しづらかった潜在的なバグといえるが，本手法を用いて検査を行ったところ，このようなバグも発見することができた．これは本手法の特筆すべき点の一つである．

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  $(ins_{ex}, t, e)$  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ことを示す。つまり、本手法では最適化のどの変形にバグの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。

## 8.2 最適化器の拡張箇所の個数

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* <sup>1</sup>	480	3	0
Lazy Code Motion	1259	2	0

表 8.1 最適化器ごとの拡張箇所の個数

表 8.1 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

最適化器のソースコードの書き換えとは、ポイントカットの指定が困難であったため、7.4.3 節で述べたような、空のメソッドの挿入を指している。

表 8.1 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\*<sup>2</sup>、発見するのは容易であった。

---

\*<sup>1</sup> 完全な実装ではないが、これ以上拡張箇所も改変箇所も増えることはない。

\*<sup>2</sup> 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。

## 8.3 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、マークの種類数はすなわち表 8.1 の拡張箇所の数に当たる。既に述べたように、拡張箇所数は少ない。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。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 8.1 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。

## 8.4 検査効率の実験

5.1 節で述べたように、本手法は各最適化パスを実行するたびに検査を行うため、検査時間は現実的な時間内であることが望まれる。この節では、現時点で検査を実装できている最適化について、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 8.2 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

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
GluonJ	1.2
Benchmark	SPEC CPU2000 version 1.2

表 8.2 実験環境

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 最適化の実行にかかった時間の合計
- B 本手法の検査付き最適化の実行にかかった時間の合計
- C 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，COINS SSA 最適化を次の順に行った．

```
prun/divex/cse/cstp/hli/osr/hli/cstp/cpyp/preqp/cstp/rpe/dce/srd3
```

また，最適化実行中のガーベッジコレクションの発生の影響をなくすため，最適化実行直前に `System.gc()` メソッドを呼んだ．時間は，`System.nanoTime()` メソッドにより計測した．

計測結果を表 8.3 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<sup>\*3</sup>．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 の 27 分から 43 分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．

本実装での CTL モデル検査器は，文献 [6] のアルゴリズムに基づいて実装したが，実装における効率の重視は行わなかったため，検査はさらに高速化できると思われる．

また，本手法では，仕様を具体化して検査する CTL 式を生成したが，5.7.1 節の例のよ

---

<sup>\*3</sup>  $B - A = D - C$  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，ガーベッジコレクションの実行時間などの影響が考えられる．

	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 8.3 検査の有無によるコンパイル時間の比較（単位：秒）

うに，同じ仕様から複数の CTL 式が生成されることが多くある．これらの CTL 式を個別に検査するよりも，まとめて検査した方が効率がよい．まとめて検査を行うのは，記号モデル検査[7]のアルゴリズムとデータフロー方程式のビットベクトル法を利用することで可能であると思われ，検査の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．

## 第 V 部

### 結論

## 第 9 章

# 関連研究

この章では，本研究に関連の深い研究について述べる．

### 9.1 Lacey らの研究

本研究に関連の深い研究に，Lacey らの研究が挙げられる [18, 19]．Lacey らは，時相論理 CTL-FV を提唱し，それを用いた書き換え規則  $I \rightarrow I'$  if  $\phi$  により最適化を行う枠組みを提案した．

書き換え規則は次の意味を持つ．

文  $I$  を持つノードが，CTL-FV 式  $\phi$  を満たすとき， $I$  を  $I'$  に書き換える．

Lacey らは，この書き換え規則により，従来の最適化の多くが記述できると主張し，実際にいくつかを記述している．

また Lacey らは，提案した枠組みの上で，最適化の正しさを証明する手法を提案し，いくつかを実際に証明した．

#### 9.1.1 本手法と Lacey らの手法の比較

本手法では，プログラムを解析し変形するには，既存の最適化器を用いた．Lacey らの手法では，書き換え規則  $I \rightarrow I'$  if  $\phi$  を記述し， $\phi$  のモデル検査によりプログラムの解析を行い，解析結果の箇所を  $I \rightarrow I'$  と変形して最適化した．

また本手法では，最適化の変形が正しかったかを調べるために，変形箇所が満たすべき仕様を CTL-FV 式で記述し，モデル検査により検証した．Lacey らの手法では，変形箇

所はすでに  $\phi$  を満たしていることが最適化時のモデル検査により分かっている。

以上の考察から，非常に大まかにいえば，次のような関係があるといえる。

最適化器 + 本手法  $\equiv$  Lacey らの手法

これは，一見 Lacey らの手法で十分であるように思えるが，Lacey らの手法には実用上の問題点がいくつかある。

### 扱える最適化の限界

Lacey らの手法は，最適化と検証が同時に行えるという利点がある。しかし，この同時に行えるという利点が欠点にもなりうる。

例えば，7.2 節で述べた条件分岐を考慮した定数伝播の最適化は，Lacey らの手法では行うことができない。Lacey らの手法では，制御フローグラフ上でのデータフロー方程式を解く以上のことはできないと予想されるが，条件分岐を考慮した定数伝播の最適化はフローグラフ上の抽象実行に基づいた最適化であり，単純にデータフロー方程式で記述できるものではない。

他にも，現実のコンパイラ最適化器は，対象の中間表現特有の問題の回避のため，または目的コードのさらなる効率向上のために細かい工夫をすることがあると思われるが，それらを CTL-FV で表現できるかどうかは疑問である。

一方本手法は，最適化自体は既存の最適化器に任せてしまい，変形の正しさの検証にのみ CTL モデル検査を用いた，と見ることができる。条件分岐を考慮した定数伝播のように，最適化の正しさの検証だけなら CTL モデル検査で行える場合もあり，一概に Lacey らの手法で十分とはいえない。

### 既存の最適化器の検査

Lacey らの手法では，一つの記述により最適化も検査も実装でき，最適化を新規実装するのが容易であるという利点があるが，逆にいうと，既存の最適化器の検査は行えない。一方本手法は，既存の手書きの最適化器の検査を行うことができる。これは本手法の大きな利点であるといえる。

## 9.2 Lerner らの研究

Lerner らは，時相論理を応用した独自のドメイン記述言語を用いて最適化を行うシステムを提案した [20, 21]。どちらも，システムで記述した最適化の正しさを証明する手順

を，定理証明器を利用することでほぼ自動化するという点に大きな特徴がある．

はじめに提案したのは Cobalt という言語であった．これは CTL の記述力を最適化向けに限定したものに近いと思われ，その意味で Lacey らの手法に非常に近いといえる．しかし，Lacey らの論文 [18] では手で証明していた最適化の正しさが，Lerner らの Cobalt を用いたシステムではほぼ自動化されており，大きな特徴となっている．

次に提案したのが Rhodium という言語であった．これは Cobalt の記述力を改善し，データフロー情報を明示的に表現できるようにしたものである．この改善により，より多くの最適化を記述できるようになった．この記述は，データフロー方程式に非常に似ており， $\mu$  計算の記述力に近いと思われる．このシステムでも正しさの証明がほぼ自動で行われると主張している．

しかし，Lerner らの提案したいずれのシステムも，制御フローグラフ上でのデータフロー方程式を超える解析能力はないと思われる．よって Lacey らと同様，条件分岐を考慮した定数伝播など，行うことのできない最適化が存在すると考えられる．

### 9.3 Necula の研究

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 [24]．この手法は，Lacey らや Lerner らのような，記述から最適化を実行する手法とは大きく異なっている．

この手法は，原理的には最適化に依存しない検査手法であるので，最適化ごとに検査仕様を作成する必要のある本手法より実現コストが低いといえる．また，最適化にかかる時間の 4 倍程度の時間で検査可能とあり，かなり高速なようである．これらから，Necula の手法は非常に実用的であるといえる．実際に，GCC や Linux カーネルのソースコードのコンパイルに対して検査を行い，既知のものではあるがバグも見つけている．

Necula の手法は，条件分岐を考慮した定数伝播など，SSA 最適化を含む多くの最適化に適用できると考えられ，適用範囲は非常に広いといえる．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．

## 9.4 Rinard らの研究

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 [27, 26]。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。

## 第 10 章

# まとめ

我々は、CTL モデル検査を利用して既存の最適化器の実行の正しさを検証する手法を提案した。この手法は次のような特徴がある。

- 最適化後に検査を行うことで、Lacey ら、Lerner らより広い範囲の最適化の実行の検証を行うことができる。
- 既存の手書きの最適化器に適用できる。
- アスペクト指向の考え方を適用し、既存の最適化器のソースの変更を最小限に抑えながら検査できる。
- 正しくない最適化に提案手法を用いることで、最適化器のバグを発見できる。バグの箇所の特定も容易である。

提案手法を COINS の最適化器に対して適用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。これは、通常のコンパイル・実行を行っても発見されなかった潜在的なバグであった。また、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。

## 第 11 章

# 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。

### 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 [10] や質問伝播に基づく大域値番号付けと部分冗長除去 [29] 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、8.4 節で述べたような手法で、大幅な検査時間の短縮が望められると思われる。

# 謝辞

本研究を進めるにあたり多大なる御指導御鞭撻を頂いた，東京工業大学 数理・計算科学専攻の佐々政孝先生に深く感謝の意を表します．

また，研究に多くの助言を頂いた佐々研究室の中谷俊晴さん，方玲さん，アスペクト指向システム GluonJ について様々な助言を頂いた千葉研究室の熊原奈津子さんにも，ここに深くお礼申し上げます．

## 参考文献

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.
- [2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 1–11. ACM Press, 1988.
- [3] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java 2nd ed.* Cambridge University Press, 2002.
- [4] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exper.*, Vol. 28, No. 8, pp. 859–881, 1998.
- [5] Shigeru Chiba, Muga Nishizawa, and Natsuko Kumahara. GluonJ home page. <http://www.csg.is.titech.ac.jp/projects/gluonj/>.
- [6] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, Vol. 8, No. 2, pp. 244–263, 1986.
- [7] E. M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.
- [8] COINS Project. COINS home page. <http://www.coins-project.org/>.
- [9] COINS Project. COINS プロジェクト LIR 仕様書, 2002. <http://www.coins-project.org/spec/lir.pdf>.
- [10] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. Operator strength reduction. *ACM Trans. Program. Lang. Syst.*, Vol. 23, No. 5, pp. 603–625, 2001.
- [11] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control

- dependence graph. *ACM Trans. Program. Lang. Syst.*, Vol. 13, No. 4, pp. 451–490, 1991.
- [12] 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.
- [13] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an optimizing compiler for Java. *Software - Practice and Experience*, Vol. 30, No. 3, pp. 199–232, 2000.
- [14] GNU Project. GCC homepage. <http://gcc.gnu.org/>.
- [15] IBM. Jikes Research Virtual Machine home page. <http://jikesrvm.sourceforge.net/>.
- [16] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pp. 224–234. ACM Press, 1992.
- [17] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: theory and practice. *ACM Trans. Program. Lang. Syst.*, Vol. 16, No. 4, pp. 1117–1155, 1994.
- [18] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 283–294. ACM Press, 2002.
- [19] 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, pp. 173–206, 2004.
- [20] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pp. 220–231. ACM Press, 2003.
- [21] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 364–377. ACM Press, 2005.
- [22] 森公一郎. COINS 新 LIR 内部構造, 2003. <http://www.coins-project.org/050303/base/BackEnd/NewLir.html>.

- [23] 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.
- [24] George C. Necula. Translation validation for an optimizing compiler. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pp. 83–94. ACM Press, 2000.
- [25] Amir Pnueli. The temporal logic of programs. In *In Proceeding of the 18th IEEE Symposium on Foundations of Computer Science*, pp. 46–77, 1977.
- [26] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, Jul 1999.
- [27] Martin Rinard. Credible compilers. Technical Report MIT/LCS/TR-776, Massachusetts Institute of Technology, 1999.
- [28] 佐々政孝. プログラミング言語処理系. 岩波書店, 1989.
- [29] 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>.
- [30] Scale Compiler Group. Scale home page. <http://www-ali.cs.umass.edu/Scale/>.
- [31] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 38–48. ACM Press, 1998.
- [32] Standard Performance Evaluation Corporation. SPEC home page. <http://www.spec.org/>.
- [33] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *SAS '99: Proceedings of the 6th International Symposium on Static Analysis, Lecture Notes in Computer Science*, Vol. 1694, pp. 194–210. Springer-Verlag, 1999.
- [34] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, Vol. 13, No. 2, pp. 181–210, 1991.

## 付録 A

# COINS SSA 最適化モジュールの 検査仕様の記述

本研究で実装した、いくつかの最適化の検査仕様について簡単に述べる。以下の最適化は全て SSA 形式上の最適化であることに注意する。

### A.1 コピー伝播

マークを付ける変形箇所および解析箇所は以下の通りである。

1. コピー文  $x = y$  を削除した箇所。  
マーク  $(\text{rm}, x, y)$  を付ける。
2. 変数  $x$  の使用を等価な変数  $y$  で置換した箇所。  
マーク  $(\text{rpl}, x, y)$  を付ける。
3. 変数  $x$  が変数  $y$  と等しいことを表から仮定した箇所。  
マーク  $(\text{look}, x, y)$  を付ける。
4. ブロック  $b$  中の  $\phi$  関数において、ブロック  $p$  からの変数  $x$  が変数  $y$  と等しいことを表から仮定した箇所。  
マーク  $(\text{look}_\phi, x, y, b, p)$  を付ける。
5.  $x = t$  と  $t = y$  から  $x = y$  を推測した箇所。  
マーク  $(\text{guess}, x, y, t)$  を付ける。

コピー文を削除した箇所

コピー文  $x = y$  があると, 以後の  $x$  の使用は全て  $y$  または  $y$  に等しい変数に置換される. よって,  $x$  が使用されなくなっていれば正しい.

$$\begin{array}{ll} \text{MARK} & (\text{rm}, x, y) \\ \text{FORMULA} & \neg EF \text{ use}(x) \end{array} \quad (\text{A.1})$$

変数の使用を等価な変数で置換した箇所

$x$  を  $y$  に置換した箇所は,  $x$  と  $y$  が等しいという解析結果があれば正しい.

$$\begin{array}{ll} \text{MARK} & (\text{rpl}, x, y) \\ \text{FORMULA} & \text{look-equal}(x, y) \vee \text{mark}(\text{guess}, x, y, *) \end{array} \quad (\text{A.2})$$

ただし,  $\text{look-equal}(x, y)$  は次のように定義される.

$$\text{look-equal}(x, y) \equiv \text{mark}(\text{look}, x, y) \vee \text{mark}(\text{look}_\phi, x, y, *, *)$$

変数が別の変数と等しいことを表から仮定した箇所

$x$  と  $y$  が等しいことを表から引いた箇所は, 全ての先行パスで,  $x = y$  というコピー文があったノードか, もしくは  $x$  と  $y$  が等しいと推論したノードがあり, その後  $x$  も  $y$  も変更されていないならば正しい.

$$\begin{array}{ll} \text{MARK} & (\text{look}, x, y) \\ \text{FORMULA} & \overleftarrow{A} ((\neg \text{def}(x) \wedge \neg \text{def}(y)) \ W \ \text{guess-equal}(x, y)) \end{array} \quad (\text{A.3})$$

ただし,  $\text{guess-equal}(x, y)$  は次のように定義される.

$$\text{guess-equal}(x, y) \equiv \text{mark}(\text{rm}, x, y) \vee \text{mark}(\text{guess}, x, y, *)$$

$\phi$  関数中の変数が別の変数と等しいことを表から仮定した箇所

$\phi$  関数中の変数の場合, その変数の関連するブロック  $p$  から考える必要がある.

$$\begin{array}{ll}
\text{MARK} & (\text{look}_\phi, x, y, b, p) \\
\text{FORMULA} & \overleftarrow{A} (\text{block}(b) \cup (\text{block}(p) \rightarrow \\
& \overleftarrow{A} ((\neg \text{def}(x) \wedge \neg \text{def}(y)) \cup \text{guess-equal}(x, y))))
\end{array} \tag{A.4}$$

変数の等価性を推測した箇所

$x$  と  $y$  が等しいと推論した箇所は,  $x$  と  $t$ ,  $t$  と  $y$  の等価性が既にある場合に正しい.

$$\begin{array}{ll}
\text{MARK} & (\text{guess}, x, y, t) \\
\text{FORMULA} & \text{equal}(x, t) \wedge \text{equal}(t, y)
\end{array} \tag{A.5}$$

ただし,  $\text{equal}(x, y)$  は次のように定義される.

$$\text{equal}(x, y) \equiv \text{look-equal}(x, y) \vee \text{guess-equal}(x, y)$$

## A.2 共通部分式除去

マークを付ける変形箇所および解析箇所は以下の通りである.

1. コピー文  $x = y$  を削除した箇所.  
マーク  $(\text{rm\_cp}, x, y)$  を付ける.
2. 変数  $x$  の使用をコピー文の左辺の変数  $y$  で置換した箇所.  
マーク  $(\text{rpl\_cp}, x, y)$  を付ける.
3. ブロック  $b$  中の  $\phi$  関数において, ブロック  $p$  からの変数  $x$  の使用を等価な変数  $y$  で置換した箇所.  
マーク  $(\text{rpl\_cp}_\phi, x, y, b, p)$  を付ける.
4. 式  $e$  の値が変数  $v$  によって利用可能になる箇所.  
マーク  $(\text{avl}, v, e)$  を付ける.
5. 式  $e$  の使用を変数  $v$  で置換した箇所.  
マーク  $(\text{rpl}, v, e)$  を付ける.
6.  $\phi$  関数  $e$  の値が変数  $v$  によって利用可能になる箇所.  
マーク  $(\text{avl}_\phi, v, e)$  を付ける.
7.  $\phi$  関数  $e$  を変数  $v$  で置換した箇所.  
マーク  $(\text{rpl}_\phi, v, e)$  を付ける.
8. 変数  $x$  に代入する  $\phi$  関数の評価値が常に定数  $c$  となる文を削除した箇所.

マーク  $(\text{const}, x, c)$  を付ける .

9. 8 で削除した文と等価な定数代入文を挿入した箇所 .

マーク  $(\text{ins}, x, c)$  を付ける .

コピー文を削除した箇所

コピー文  $x = y$  を削除した箇所は , コピー伝播のときと同様 , 後続パスで  $x$  が使用されていなければ正しい .

$$\begin{array}{ll} \text{MARK} & (\text{rm\_cp}, x, y) \\ \text{FORMULA} & \neg EF \text{ use}(x) \end{array} \quad (\text{A.6})$$

変数の使用をコピー文の左辺の変数で置換した箇所

$x$  を  $y$  に置換した箇所は , 全ての先行パスに  $x = y$  というコピー文があれば正しい . コピー文  $x = y$  は削除され , マーク  $(\text{rm}, x, y)$  が付いていることに注意する .

$$\begin{array}{ll} \text{MARK} & (\text{rpl\_cp}, x, y) \\ \text{FORMULA} & \overleftarrow{A} ((\neg def(x) \wedge \neg def(y)) W \text{ mark}(\text{rm\_cp}, x, y)) \end{array} \quad (\text{A.7})$$

$\phi$  関数中の変数の使用をコピー文の左辺の変数で置換した箇所

$\phi$  関数中の変数を置換する場合は , コピー伝播のときと同様 , 関連するブロックから考える .

$$\begin{array}{ll} \text{MARK} & (\text{rpl\_cp}_\phi, x, y, b, p) \\ \text{FORMULA} & \overleftarrow{A} (\text{block}(b) U (\text{block}(p) \rightarrow \\ & \overleftarrow{A} ((\neg def(x) \wedge \neg def(y)) W \text{ mark}(\text{rm\_cp}, x, y)))) \end{array} \quad (\text{A.8})$$

式が利用可能になる箇所

式  $e$  の値が  $v$  によって利用可能になる箇所は ,  $v = e$  に相当する文のあった箇所である . この箇所では正しさのために要求される性質はない . このマークは他のマークの正しさの検証のために付ける . よって仕様は必要ない .

式の使用を変数で置換した箇所

式  $e$  の使用を変数  $v$  で置換した箇所は、全ての先行パスで  $e$  の値が  $v$  によって利用可能となっていればよい。

$$\begin{array}{ll} \text{MARK} & (\text{rpl}, v, e) \\ \text{FORMULA} & \overleftarrow{A} (\text{trans}(e) \ W \ \text{mark}(\text{avl}, v, e)) \end{array} \quad (\text{A.9})$$

$\phi$  関数が利用可能になる箇所

式が利用可能になる箇所と同様、これも他のマークの正しさの検証のために付けるマークである。よって仕様は必要ない。

$\phi$  関数を変数で置換した箇所

$\phi$  関数  $e$  を変数  $v$  で置換した箇所は、全ての先行パスで  $e$  が  $v$  によって利用可能となっていればよい。

$$\begin{array}{ll} \text{MARK} & (\text{rpl}_\phi, v, e) \\ \text{FORMULA} & \overleftarrow{A} (\text{trans}(e) \ W \ \text{mark}(\text{avl}, v, e)) \end{array} \quad (\text{A.10})$$

定数となる  $\phi$  関数を削除した箇所

評価値を変数  $x$  へ代入する  $\phi$  関数は、引数が全て同じ定数  $c$  でならば、この  $\phi$  関数を削除し、等価な定数代入文  $x = c$  に置き換えることができる。 $\phi$  関数の引数全てが同じ定数になることは、マーク付けの際に検査すればよく、時相論理により検査する必要はない。

削除した  $\phi$  関数と等価な定数代入文を挿入した箇所

$x = c$  を挿入する箇所は、全ての先行パスに削除した  $\phi$  関数があったノードがあればよい。そのパス上には他のノードでの  $x$  の使用も定義もあってはならない。これは、間にループを挟むことがないので、 $W$  演算子ではなく  $U$  演算子となる。

$$\begin{array}{ll} \text{MARK} & (\text{ins}, x, c) \\ \text{FORMULA} & \overleftarrow{A} X \overleftarrow{A} ((\neg \text{dex}(x) \wedge \neg \text{use}(x)) \ U \ \text{mark}(\text{const}, x, c)) \end{array} \quad (\text{A.11})$$

### A.3 無用命令除去

マークを付ける変形箇所および解析箇所は以下の通りである。

1. 死んでいる変数  $x$  への代入文を削除した箇所。  
マーク  $(\text{rm}, x)$  を付ける。
2. 無用な分岐を削除した箇所。  
マーク  $(\text{rm\_edge}, l)$  を付ける。
3. 到達不能ブロック  $b$  の文を削除した箇所。  
マーク  $(\text{unreach}, b)$  を付ける。

#### 代入文を削除した箇所

変数  $x$  への代入文を削除した箇所は、以後  $x$  が使用されなければ正しい。

$$\begin{array}{ll} \text{MARK} & (\text{rm}, x) \\ \text{FORMULA} & \neg EF \text{ use}(x) \end{array} \quad (\text{A.12})$$

#### 分岐を削除した箇所

仕様書 [29] によると、次の条件が全て成り立つとき、分岐を削除し JUMP 式に書き換えてよい。

- ループの出口ブロックの文ではない。
- この分岐文に制御依存するブロックに生きている文がない。
- この分岐文に制御依存するブロックの後続ブロックに  $\phi$  関数がない。

この分岐文のノードを  $n$  とすれば、満たすべき条件は CTL-FV でおおよそ以下のよう  
に書ける。

$$n \models \neg \text{exit-loop} \quad (\text{A.13})$$

$$n \models \neg EF (c\text{-dep}(m, n) \wedge \text{live}) \quad (\text{A.14})$$

$$n \models \neg EF (c\text{-dep}(m, n) \wedge EX \text{ exist-phi}) \quad (\text{A.15})$$

ただし,  $c\text{-dep}(m, n)$  は「 $m$  は  $n$  に制御依存している」という意味の式で,

$$c\text{-dep}(m, n) \equiv \overleftarrow{E} ((\neg E(\text{node}(m) \cup \text{node}(\text{end}))) \cup \text{node}(n))$$

と表される.  $\text{node}(\text{end})$  はプログラムの出口のノードである.

また,  $\text{live}$  は生きている文, つまり削除しなかった文を表す式として書け,  $\text{exist-phi}$  は,  $\phi$  関数のある文を表す式として書ける.

ところが, ループの出口を表す  $\text{exit-loop}$  は, 一般に記述ができないと考えられる. 構造化したループしかないことが保証されていれば記述可能であるが, LIR では厳密にはそういう仮定ができない. よって, 現在ではこのマークに対する正確な仕様を記述するに至っていない.

到達不能ブロック中の文を削除した箇所

ブロック  $b$  が到達不能なら, このブロックの文は全て削除できる.  $b$  が到達不能であるためには, 次のいずれかを満たす必要がある.

- 入口ノードから  $b$  に至るパスがない. つまり,  $b$  から入口ノードに到達できない.
- 一つ前のノードは, 同じく到達不能か, もしくはこのブロックへの分岐の辺を削除したノードである.

以上から, 仕様は次のようになる.

$$\begin{array}{ll} \text{MARK} & (\text{unreach}, b) \\ \text{FORMULA} & \neg \overleftarrow{E} F \text{ node}(\text{start}) \vee \\ & \overleftarrow{A} X (\text{mark}(\text{unreach}, b) \vee \text{mark}(\text{rm\_edge}, b)) \end{array} \quad (\text{A.16})$$